SecSess: Keeping your Session Tucked Away in your Browser

Philippe De Ryck, Lieven Desmet, Frank Piessens, Wouter Joosen
iMinds-Distrinet, KU Leuven, 3001 Leuven, Belgium
Philippe.DeRyck@cs.kuleuven.be

ABSTRACT

Session management is a crucial component in every modern web application. It links subsequent requests and temporary stateful information together, enabling a rich and interactive user experience. Unfortunately, the de facto standard cookie-based session management mechanism is imperfect, which is why session management vulnerabilities rank second in the OWASP top 10 of web application vulnerabilities [18]. While improved session management mechanisms have been proposed, none of them achieves compatibility with currently deployed applications or infrastructure components such as web caches.

We propose *SecSess*, a lightweight session management mechanism that addresses common session management vulnerabilities by ensuring a session remains under control of the parties that established it. *SecSess* is fully interchangeable with the currently deployed cookie-based session management, and can be gradually deployed to clients and servers through an opt-in mechanism. Evaluation of our proof-of-concept implementation shows that *SecSess* introduces only a minimal performance and networking overhead. Furthermore, we empirically show that *SecSess* is effectively compatible with commonly used web caches, in contrast to alternative approaches.

1. INTRODUCTION

Session management is a fundamental component of every modern web application, and enables stateful features, such as tracking the authentication state or processing multi-step transactions. As HTTP, the workhorse protocol of the Web, is stateless by nature, stateful behavior like session management has been added through the use of cookies. Unfortunately, the security requirements of a session management mechanism clash with the security properties offered by cookies, paving the way for high-impact attacks on session management mechanisms, such as session hijacking and session fixation. Consequently, session security problems are ranked second in the industry-driven OWASP Top Ten of web application security problems [18].

At the heart of a successful attack against session management lies an unauthorized session transfer, where the attacker succeeds in transfering an established session between his and the victim's browser. Such an unauthorized session transfer is enabled by the weak security properties of the uniquely assigned session identifier. This session identifier acts as a *bearer token*, meaning that the merely including this identifier as a cookie in a request suffices for legitimizing the request within the session associated with the identifier. For example in a session hijacking attack, the attacker manages to get hold of the user's session identifier, allowing him to transfer the session to his own browser.

In response to these attacks, the *HttpOnly* and *Secure* cookie attributes have been introduced, allowing an application to strengthen the security of sensitive cookies, such as the one containing the session identifier. While these attributes mitigate several of the currently exploitable attack vectors [5, 11], they do not eliminate the inherent design flaw in current session management mechanisms, namely the bearer token properties of the session identifier. This property is a lurking threat to the security of session management, rearing its head when the session identifier is leaked through a vulnerability in the underlying secure transport system [2], or by a failure to correctly apply these cookie security attributes.

In this work, we propose *SecSess*, a lightweight session management mechanisms that effectively prevents the unauthorized transfer of an established session. *SecSess* eradicates the bearer token properties of the session identifier by agreeing on a shared secret between browser and server, used to guarantee the integrity of the requests. As this shared secret can not be transfered without authorization, an attacker is prevented from carrying out session hijacking or session fixation attacks. *SecSess* improves on previous proposals of alternative session management mechanisms by ensuring full compatibility with currently deployed cookie-based session management mechanisms, allowing a fully interchangeable deployment through an opt-in upgrade path. Additionally, we have extensively evaluated the impact of *SecSess* by means of a prototype implementation, showing that *SecSess* only incurs a minimal computational and network overhead, and does not introduce additional requests and roundtrips. To our knowledge, *SecSess* is the only session management mechanism explicitly designed to be compatible with currently deployed Web infrastructure, such as web caches.

SEC@SAC '15 , April 13 - 17 2015, Salamanca, Spain
ACM 978-1-4503-3196-8/15/04 ...$15.00.
http://dx.doi.org/10.1145/2695664.2695764.

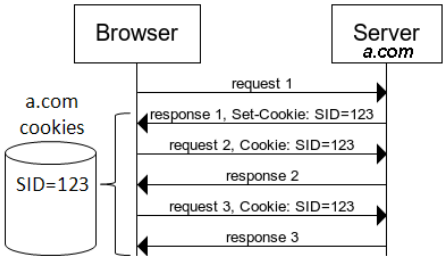

Figure 1: In cookie-based session management mechanisms, the server issues a cookie with a session identifier, which the browser stores in the domain-specific cookie jar and attaches to future requests.

2. BACKGROUND

In a cookie-based session management mechanism, the server generates a random identifier for a session, and sends it to the browser using the *Set-Cookie* header. The browser stores the cookie in the so-called *cookie jar*, and whenever a request is sent to a domain for which cookies are present in the cookie jar, the browser attaches these cookies using the *Cookie* header, as illustrated in Figure 1.

Unfortunately, these session management systems are inherently vulnerable to attacks resulting in the unauthorized transfer of an established session, such as session hijacking and session fixation. In the remainder of this section, we elaborate on these attacks, and argue why current session management mechanisms can not achieve these properties, even when using the additional cookie security attributes.

2.1 Threats against Session Management

One concrete attack that leads to the unauthorized transfer of an established session is a *session hijacking* attack [11], where an adversary is able to steal a user's session identifier. Simply attaching this session identifier to crafted requests is typically sufficient to hijack the user's session, granting the adversary the same level of access as the user. Concrete attack vectors for a session hijacking attack are script-based cookie exfiltration using the *document.cookie* property, or eavesdropping attacks on the network, as aptly demonstrated by the Firesheep addon, which reduces a session hijacking attack to a point-and-click operation.

A second attack is *session fixation* [14], where the adversary forces the user's browser to use a compromised session. The aim of a session fixation attack is to have the user authenticate himself within a session known to the attacker, causing the server to store the user's authentication state in the attacker's session. A session fixation attack is typically carried out by writing to the *document.cookie* property.

Concretely, the essence of this threat against session management mechanisms is the *unauthorized transfer of a session*, where an attacker is able to transfer a session defined between the victim's browser and the target application to his own browser. Transfering the session grants the attacker the same privileges with the target application as the user holds. We deliberately define the threat on the conceptual level, as there are numerous concrete instantiations of a successful session transfer attack. One common example is performing a session hijacking or session fixation attack through attacker-controlled JavaScript. Another example are passive attacks on the network level, where an eavesdropping attacker can steal the session identifier from the network channel, either from the plaintext HTTP message or by performing traffic analysis attacks on an HTTPS channel [2]. In addition to passive network attacks, we also consider active network attacks on an established session to be in scope. In an active network attack, the attacker can manipulate, inject or drop packets on the network. Note that an active network attacker can also manipulate the establishment of a session, which is significantly more difficult to protect against [6]. Therefore, we consider attacks on an established session to be in scope, but man-in-the-middle attacks on the session establishment to be out of scope.

Next to these in-scope attack vectors of a session transfer attack, we consider attack vectors based on a compromise of the client-side or server-side infrastructure to be out of scope. The most common example are machines compromised by malware, both at the client and server side. Concretely, we expect an uncompromised machine and browser codebase at the client, as well as an uncompromised machine and web application codebase at the server.

2.2 Current Mitigation Techniques

Since these attacks on session management mechanisms are well-known and well-documented, several countermeasures are available. Most relevant are the *HttpOnly* and *Secure* cookie flags, which respectively restrict script-based access to cookies, and prevent the transmission of cookies over insecure channels. While these countermeasures offer adequate protection if deployed correctly, they do not fundamentally prevent the unauthorized transfer of an established session, as the session identifier remains a bearer token. Additionally, these countermeasures are often not or incorrectly deployed, even within the Alexa top 100 sites [3], and new attacks that compromise secure deployments have been discovered [2].

Additionally, while the benefits of HTTPS deployments are evident, wide scale adoption on the Web is impeded by several intricacies. One often cited issue is the performance impact, an argument that has lost most of its relevance on modern hardware [10]. Second, HTTPS deployments are disproportionately more complex compared to HTTP deployments, putting a significant burden on system administrators. Examples of such complexities are creating keys, monitoring and renewing certificates, dealing with browser-approved certificate authorities, preventing mixed-content warnings and deploying shared hosting using TLS's Server Name Indication extension [7], if supported by the client. Additional to the complexity of deploying HTTPS, a wide-scale transition to HTTPS severely obstructs the operation of the so-called middleboxes, machines in between the endpoints that cache, inspect or modify traffic. These middleboxes are essential parts of the web infrastructure, for example by bringing the Web to developing nations through extensive caching and enabling efficient video transmission on mobile phone networks.

We acknowledge that wide-scale deployment of HTTPS remains imperative for securing the Web, but also recognize the long and tedious process. This explains why the recent revelations about pervasive monitoring on the Web have sparked multiple proposals looking to transparently upgrade the security properties of the HTTP channel when supported by the endpoints. One prominent proposal is to negotiate an encrypted HTTP channel without verifying the entities' authentication [12], which is even proposed as one

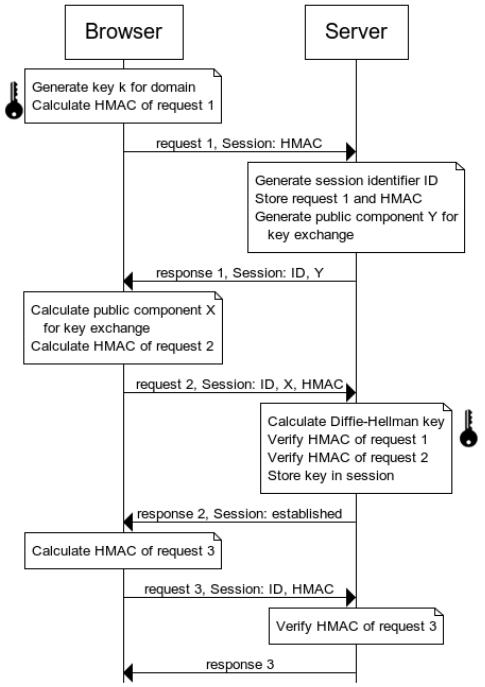

Figure 2: The flow of requests and responses used by *SecSess*, showing all the details.

of the available modes in the upcoming HTTP/2.0 specification. This eagerness to improve the security properties of the HTTP protocol, even by introducing them into the new version, shows that the HTTP protocol will be around for the near future. Therefore, it makes sense to not only upgrade the network-level protocol properties, but also to take the opportunity to improve the security properties of session management on top of the HTTP protocol.

3. SECSESS

In this section, we elaborate on our session management mechanism, *SecSess*, in three steps: (i) the actual session management mechanism, (ii) establishing the shared session secret and (iii) the resulting request flow, which is identical to the flow in cookie-based session management mechanisms. The essence of *SecSess* is establishing a shared secret used for session management between browser and server, which is stored in the browser, where it can not be obtained by an attacker. *SecSess* effectively binds an established session to its initiating parties, thereby preventing the unauthorized transfer of an established session.

Session Management.

A core feature of a session management mechanism is actually keeping track of a session, which *SecSess* achieves by using a plain session identifier. The session identifier is provided by the server using a *Session* response header (response 1 in Figure 2). 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 strongly resembles traditional cookie-based session management, the session identifier is no longer considered to be a bearer token, and is useless without knowledge of the shared secret. Therefore, using a simple incremental counter as an identifier is sufficient.

Instead of using the session identifier as the bearer token, *SecSess* uses the shared secret to add an hash-based message authentication code (HMAC) to the request, thereby legitimizing the request within the session. Since this HMAC takes the request and the shared secret as input, only the browser and the server can compute the correct values. Incoming requests with an invalid HMAC are simply discarded by the server.

Note that the input for the HMAC should be chosen carefully. Technically, a network attacker can steal the valid HMAC from an eavesdropped request and attach it to a crafted request, having the crafted request reach the server first. In order to maintain a valid HMAC on the crafted request, the attacker can only modify the parts of the request that are not part of the input to the HMAC function. Including the URL of the request in this input prevents an attacker from directing the request to a different destination, but still allows him to modify sensitive information in the request headers and body (e.g. the destination account of a wire transfer). Therefore, the HMAC also covers the request headers containing sensitive data[1], and, if present, the request body. Covering the URL, request headers and request body in the HMAC does not prevent an attacker from taking the valid HMAC value and attaching it to a crafted request. However, it does ensure that the attacker can not change the sensitive data, hence limiting the contents of the crafted request to those of the original request, thereby reducing the problem to the common *double submission* problem [16].

Establishing the Shared Secret.

The shared session secret, needed to compute and verify HMACs on requests, is established using the Hughes variant [13] of the Diffie-Hellman key exchange algorithm, which allows to exchange the key even in the presence of eavesdropping attackers. In Figure 2, 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), which 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, effectively establishing the session, as acknowledged in the second response.

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 when the first request already caused some server-side state to be stored in the session.

Preserving the Request Flow.

By design, *SecSess* is an application-agnostic session management mechanism, preserving the same flow of requests and responses as a currently deployed cookie-based session management mechanism. This property supports a grad-

[1]Concretely, we include the following standard HTTP headers: *Authorization*, *Cookie*, *Content-Length*, *Content-MD5*, *Content-Type*, *Date*, *Expect*, *From*, *Host*, *If-Match*, *Max-Forwards*, *Origin*.

ual deployment, where client and server software can be upgraded to opt-in to *SecSess* next to cookie-based session management. If the client does not support *SecSess*, no *Session* header is sent, so the server simply defaults to cookie-based session management. Alternatively, if the client supports *SecSess*, but the server does not, the *Session* header will be ignored by the server, and the default cookie-based session management mechanism will be used.

3.1 Handling Modified Request Flows

Since the Web is a complex distributed system, where multiple simultaneous requests are fired by the browser, request flows often differ from the flows drawn on paper. One example of a modified flow are requests that arrive at the server in a different order than they were sent. A second example are middleboxes changing the request flow, such as a Web cache responding to a request, which will thus not be sent to the server. Since these scenarios are common in the Web, it is important that they are robustly handled by a newly introduced session management mechanism.

The design of *SecSess* explicitly takes modified request flows into account, and effectively achieves full compatibility with currently deployed web caches, both within the browser and on the intermediate network. First, by only adding integrity protection, the caching of content is effectively enabled. Second, *SecSess* is robust enough to deal with out-of-order requests and cached responses, which is fairly trivial once a session is established, but challenging during establishment. If the client's public component (request 2 in Figure 2) would get lost in transit, for example when an intermediate cache responds to a request, the server would not be able to complete the session establishment, effectively breaking the protocol. Concretely, *SecSess* addresses this by continuing to send the public component as long as the server has not confirmed the session establishment (response 2 in Figure 2), effectively preventing it from getting lost in a modified request flow. We discuss the concrete impact of this decision during the performance evaluation.

4. IMPLEMENTATION AND EVALUATION

To show the feasibility of *SecSess* on the Web, as well as to support evaluation, we created a proof-of-concept implementation[2]. At the client side, we have extended the Firefox browser with support for *SecSess*, heavily leveraging the support of OpenSSL's crypto library. At the server side, we have implemented a session management middleware module for the Express framework, which runs on top of Node.js, an event-driven bare metal web server. The middleware amounts to a mere 113 meaningful lines of code, and a binary module linking the OpenSSL library is 178 meaningful lines of code.

4.1 Security

The security evaluation of *SecSess* with regard to the proposed in-scope threat model considers several concrete attack vectors. A first is the capability to run attacker-controlled scripts within the context of the target application. A session hijacking or session fixation attack using this attack vector will no longer succeed, since none of *SecSess*'s data is available to the JavaScript environment. Additionally, the *Session* request and response headers contain only public information, of no use to an attacker.

Session transfer attacks can also be performed on the network level. Eavesdropping attacks on the session management mechanism are effectively mitigated by *SecSess*, since the shared secret used for calculation of the HMACs is never communicated over the wire, and the Hughes variant of the Diffie-Hellman key exchange can withstand passive attacks. Next to passive attacks, an attacker can also try to modify existing requests, or re-attach a valid HMAC to a crafted request. Such attempts will fail as well, because the HMAC is based on the contents of the request, effectively preventing any modifications to go unnoticed.

4.2 Performance and Network Overhead

Figure 3 shows the performance overhead induced by *SecSess* on a session establishment timeline. To get correct measurements, we calculated 100 data points for each step, which contain the average computation time of 100 runs each, executed from within JavaScript code, both on the client-side (browser add-on) as the server side (Node.js)[3].

Most notable results are the very limited overhead at the client-side, especially after the session has been established (from request 3 onwards). At the server side, there is a significant pre-calculation overhead (212ms) for generating the required parameters. This overhead is induced by the Hughes variant of the Diffie-Hellman key exchange, which requires the inverse of the server's private component. Note that these parameters are session-independent, and can be pre-calculated offline in bulk, and read from a file on a per-need basis. After the parameters have been calculated, the additional overhead for actually establishing and maintaining a session is negligible.

In a Web context, network overhead can be caused by increased message sizes, but also by introducing additional requests or round trips in the flow of requests. By design, *SecSess* follows the same sequence of requests and responses used in currently existing applications, which deploy cookie-based session management mechanisms, so no additional requests or round trips are required. For brevity reasons, we can not go into much detail, but we have confirmed that, compared to the sizes of cookies used on the top 5,000 sites, *SecSess* leads to 25.58% reduction in the size of the session management headers in requests, a 9.19% increase in response headers after session establishment, and a 867.44% increase in response headers during the brief stage of session establishment, due to the transmission of the parameters to generate the secret.

4.3 Compatibility with Web Caches

Web caches are widely deployed throughout the Web, enabling faster page loads and limiting the required bandwidth. Caches are often deployed in a transparent way, where they intercept HTTP traffic, and respond when they have the resource in cache. When a cache responds to a request, the request is never forwarded to the target server, resulting in a modified request flow. A crucial property of a

[2]Removed for anonymization. If desired, our prototype can be released to the reviewers through a request from the conference chair.

[3]Experiments have been performed in a VirtualBox VM (Linux Mint 15), which was assigned 1 Intel i7-3770 core and 512 Mb of memory.

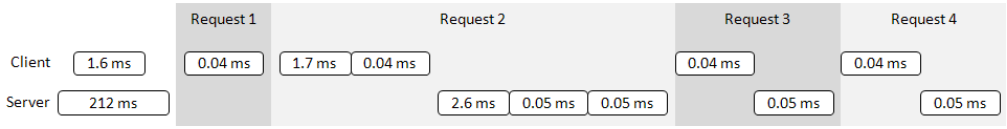

Figure 3: *SecSess* adds an average 4.3 milliseconds to the session establishment. The first step, where the shared secret is generated at the client side and the computation parameters are generated at the server side, takes a bit longer, but can be pre-computed offline or during idle times.

newly proposed session management mechanism is the compatibility with currently deployed infrastructure.

SecSess is robustly designed to be compatible with such modified request flows. We have confirmed this compatibility empirically by running experiments with two popular caches, Squid [15] and Apache Traffic Server [17], configured as a forwarding proxy. In our setup (Figure 4), we add *SecSess* session management on top of the requests sent between the browser and the web servers of the Alexa top 1,000 sites. Since these servers do not know about *SecSess*, we have added a dedicated *SecSess*-proxy in between, which will handle the *SecSess* session management with the browser (full arrows), while forwarding the request to the actual web server (dashed arrows). Finally, we add the cache in between the browser and the *SecSess*-proxy. This setup allows us to test the establishment and maintaining of a session with traffic patterns from the Alexa top 1,000 sites. Additionally, when the cache responds to a request, the *SecSess*-proxy will never see the request. This effectively allows us to verify the robustness of *SecSess* when dealing with modified request flows. The results are shown in Figure 5.

To maximize the potential of the cache, we visited each site in the Alexa top 1,000 twice. For the Squid run, 52,947 requests were sent to 5,167 distinct hosts. Of these requests, 5,008 were cached, of which 830 during session establishment, and 4,178 when an established session was already present. For the Apache Traffic Server run, we observed 44,173 requests to 4,660 hosts in total, of which 4,263 were served from the cache. 1,169 cached responses occurred during session establishment, and 3,094 with an established session. During these requests, *SecSess* robustly handled session management, without losing an established session, or failing to establish a session.

5. RELATED WORK

Related work offers several proposals to tackle the current session management problems. While these approaches offer significant benefits over traditional cookie-based mechanisms, they are typically not interchangeable, thereby hindering deployment on legacy code or within development frameworks. Additionally, many of the proposals depend on the presence of a TLS-channel, and do not withstand passive network attacks when such a channel is unavailable.

SessionLock [1] uses a JavaScript library to augment 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 a non-TLS scenario, where the client performs an out-of-band Diffie-Hellman key exchange with the server. While the idea behind SessionLock is similar to the idea behind *SecSess*, the implementation differs significantly. The implementation as a JavaScript library not only fails to protect against script-based attacks,

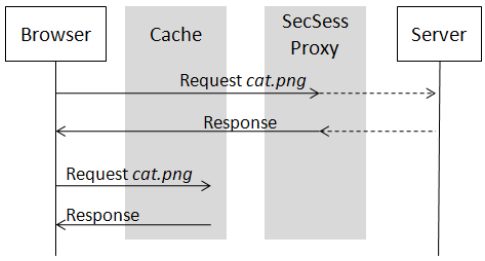

Figure 4: The setup of the cache compatibility experiment, for browsing the Alexa top 1,000.

but also requires significant changes to existing applications, as all requests have to be made through AJAX calls.

BetterAuth [9] 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 used to sign requests, offering authenticity. BetterAuth offers strong security properties, and is even capable of protecting against man-in-the-middle attacks. However, BetterAuth requires TLS for the initial exchange of the password, as well as the modification of existing applications. Additionally, BetterAuth depends on the password, it is incompatible with current third-party authentication services.

The HTTP Integrity Header [8] is an expired draft proposing to add integrity protection to HTTP, which includes a session management mechanism. 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. The HTTP Integrity header actually shares the same idea as *SecSess*, using a shared secret for session management and integrity properties. However, the HTTP Integrity header uses the original Diffie-Hellman protocol, which only establishes a secret at the client after the first request and response have been exchanged. This leaves the setup phase of the session vulnerable to passive network attacks. Additionally, the HTTP Integrity header does not account for the adverse effects of caches or out-of-order requests during session establishment.

One-Time Cookies [4] 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 separately verified and tied to an existing session. To share the initial credential, One-Time Cookies depends on the use of TLS during the authentication phase. One-Time Cookies would be a good replacement for traditional cookie-based session management mechanisms. However, since the initialization must be done over TLS, it loses its security properties when deployed for applications that only use HTTP, making a short-term deployment infeasible.

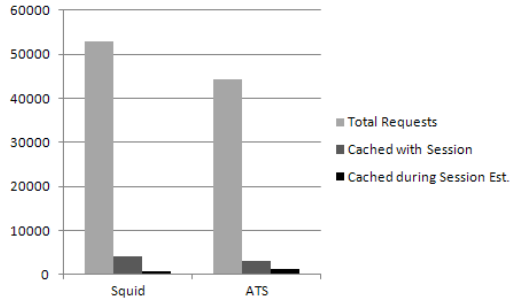

Figure 5: The results of the cache compatibility experiment, for browsing the Alexa top 1,000.

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. TLS-OBC offers strong security guarantees, and is able to eliminate the bearer token-properties of sensitive cookies. However, since TLS-OBC obviously depends on a TLS-only deployment, it is not a feasible solution for securing current and future HTTP deployments.

6. CONCLUSION

The currently deployed cookie-based session management mechanisms are extremely vulnerable to an unauthorized transfer of an established session. Advocated best practices mitigate part of the problem, but fail to eradicate the underlying cause of these attacks, the bearer token properties of the session identifier. We have proposed *SecSess*, a lightweight session management mechanism that prevents unauthorized session transfers, and is explicitly designed to be compatible with the current Web, a feature lacking from alternative proposals. *SecSess* preserves the flow of requests observed today with cookie-based session management mechanisms, hence *SecSess* is compatible both with current infrastructure as with current web applications, as illustrated by an empirical evaluation on the top 1,000 sites. Finally, we have shown that *SecSess* can be easily implemented in browsers, enabling a fast, wide-scale deployment.

Acknowledgements

This research is partially funded by IWT, the Research Fund KU Leuven, the IWT-SBO project SPION, and by the EU FP7 project STREWS. With the financial support from the Prevention of and Fight against Crime Programme of the European Union (B-CCENTRE).

7. 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] N. J. AlFardan and K. G. Paterson. Lucky thirteen: Breaking the tls and dtls record protocols. In *IEEE Symposium on Security and Privacy*, 2013.

[3] S. Calzavara, G. Tolomei, M. Bugliesi, and S. Orlando. Quite a mess in my cookie jar!: leveraging machine learning to protect web authentication. In *Proceedings of the 23rd international conference on World wide web*, pages 189–200. International World Wide Web Conferences Steering Committee, 2014.

[4] 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.

[5] P. De Ryck, N. Nikiforakis, L. Desmet, F. Piessens, and W. Joosen. Serene: self-reliant client-side protection against session fixation. In *Distributed Applications and Interoperable Systems*, pages 59–72. Springer, 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] D. Eastlake 3rd. Transport layer security (TLS) extensions: Extension definitions. *RFC 6066*, 2011.

[8] P. Hallam-Baker. Http integrity header. *Online at http://tools.ietf.org/html/draft-hallambaker-httpintegrity-02*, 2012.

[9] 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.

[10] A. Langley, N. Modadugu, and W. Chang. Overclocking ssl. In *Velocity: Web Performance and Operations Conference*, 2010.

[11] 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.

[12] M. Nottingham. Opportunistic encryption for HTTP URIs. 2013.

[13] B. Schneier. *Applied cryptography: protocols, algorithms, and source code in C*. John Wiley & sons, 2007.

[14] M. Schrank, B. Braun, M. Johns, and J. Posegga. Session fixation–the forgotten vulnerability? *Proceedings of GI Sicherheit*, 2010, 2010.

[15] Squid Project Maintainers. squid: Optimising Web Delivery. *Online at* `http://www.squid-cache.org/`, 2014.

[16] TechNoesis. 4 ways to prevent duplicate form submission. *Online at* `http://technoesis.net/prevent-double-form-submission/`, 2013.

[17] The Apache Software Foundation. Apache Traffic Server. *Online at* `http://trafficserver.apache.org/`, 2014.

[18] J. Williams and D. Wichers. Owasp top 10. *OWASP Foundation*, 2013.