Eradicating Bearer Tokens for Session Management

Philippe De Ryck, Lieven Desmet, Frank Piessens, Wouter Joosen
iMinds-DistriNet, KU Leuven, 3001 Leuven, Belgium
{firstname.lastname}@cs.kuleuven.be

***Abstract*— Session management is a crucial component in every modern web application. It links multiple requests and temporary stateful information together, enabling a rich and interactive user experience. The de facto cookie-based session management mechanism is however flawed by design, enabling the theft of the session cookie through simple eavesdropping or script injection attacks. Possession of the session cookie gives an adversary full control over the user's session, allowing him to impersonate the user to the target application and perform transactions in the user's name. While several alternatives for secure session management exist, they fail to be adopted due to the introduction of additional roundtrips and overhead, as well as incompatibility with current Web technologies, such as third-party authentication providers, or widely deployed middleboxes, such as web caches.**

We identify four key objectives for a secure session management mechanism, aiming to be compatible with the current and future Web. We propose *SecSess*, a lightweight session management mechanism based on a shared secret between client and server, used to authenticate each request. SecSess ensures that a session remains under control of the parties that established it, and only introduces limited overhead. During session establishment, SecSess introduces no additional roundtrips and only adds 4.3 milliseconds to client-side and server-side processing. Once a session is established, the overhead becomes negligible (< 0.1ms), and the average size of the request headers is even smaller than with common session cookies. Additionally, SecSess works well with currently deployed systems, such as web caches and third-party services. SecSess also supports a gradual migration path, while remaining compatible with currently existing applications.

I. Introduction

Session management is a corner stone of any modern web application, since it enables the temporary storage of stateful information, crucial for widely-used features such as user authentication, authorization, transaction processing, etc. Considering that HTTP, the de facto communication protocol of the Web, is stateless by design, additional technology supporting session management has been added afterwards. Unfortunately, current session management techniques' dependence on a bearer token is a prevailing source of security problems, allowing an attacker to gain full access to a user's account.

Generally, a unique, random session identifier, generated by the server-side application or underlying framework and subsequently shared with the browser is used to identify the session. The session identifier is a bearer token, making its presence the only verification of the legitimacy of the request within this particular session. Consequently, an adversary obtaining the session identifier can easily impersonate the user by sending requests with the stolen session identifier attached, a so-called session hijacking attack [11]. Two common attack vectors for stealing the cookie containing the session identifier are through injected JavaScript code, or eavesdropping on the network traffic, where the session cookie is attached to the plaintext HTTP requests.

The first attack vector can be countered by adding the *HttpOnly* flag to the session cookie, which makes the cookie inaccessible from JavaScript. Similarly, the *Secure* flag is introduced to counter the second attack vector, as it prevents the browser from sending the cookie on insecure, plaintext channels. However, the use of the *Secure* flag is closely connected to deploying the web application over a secure HTTPS channel, where the HTTP protocol is run over a TLS-secured connection. While the benefits of HTTPS deployments are evident, its adoption on the Web is rather limited. A 2010 study by Qualys [12] 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, of which a large percentage (approx. 25%) does not even validate correctly. Similarly, last month's data from the SSL Pulse project [15] shows that only 51% of approximately 160,000 sites that voluntarily tested their TLS security are actually secure, with 6.2% of sites with an incomplete certificate chain. While a scientific survey of the culprit for the slow and potentially insecure adoption is lacking, explanations based on conjecture often refer to (a) the potential performance impact, of which the relevance has diminished with modern hardware [10], (b) the difficulty of presenting the correct certificate on shared hosting systems, addressed by the Server Name Indication extension of TLS [6], which is not yet supported on older operating systems, such as Windows XP, (c) the impact of encrypted traffic on middleboxes installed by ISPs and network administrators, such as web caches, traffic inspection systems and malware detection systems, and (d) the complexity of creating keys, signing requests, installing certificates and dealing with browser-approved Certificate Authorities in general.

Without a doubt, wide-scale deployment of HTTPS would entail a significant security upgrade. However, an objective, non-utopian view on the evolution of the state of practice shows that full-scale TLS adoption will be an uphill battle, leaving a transition phase with mixed usage of HTTP and HTTPS. Additionally, while cookie-based session management with the appropriate flags over an HTTPS channel is currently adequately secure, the session cookie still remains a bearer token. Therefore, we think it is crucial to invest in the security of session management mechanisms used with the HTTP(S) protocol, effectively eradicating attacks aiming to steal a user's session.

In this work, we give an overview of the currently available session management mechanisms, and propose an alternative lightweight session management mechanism, *SecSess*. SecSess

ensures that an established session remains under control of the browser and server that created it, thereby effectively mitigating attacks depending on the bearer token property of the session identifier. SecSess can be implemented in an application-agnostic way, enabling a gradual migration path, and is compatible with the current infrastructure of the Web, such as intermediate caches and traffic inspection boxes. SecSess introduces no significant performance overhead and fits well within the current flows of requests and responses observed on the Web, thereby avoiding the introduction of additional requests or roundtrips.

In the remainder of this work, we define the relevant threat model for session management, identify four key objectives for secure session management and discuss related work (Section II). We present *SecSess*, a secure session management mechanism (Section III), and conclude with a discussion of several practical compatibility scenarios and deployment strategies (Section IV).

II. Session Management on the Web

Currently deployed cookie-based session management mechanisms often lack adequate protection, allowing the session identifier to be stolen by an adversary, for example by eavesdropping on the network or by injecting malicious JavaScript within the target application. Even within the Alexa top 100, session management vulnerabilities can be found, with the popular image sharing site Flickr as an example.

In this section, we define the threat model relevant for session management, and determine four key objectives for a secure session management mechanism. Finally, we discuss related work in light of the proposed threat model and objectives.

A. Threat Model for Session Management

Currently deployed session management mechanisms are threatened and exploited in various ways. Two common threats against current session management mechanisms are eavesdropping attacks and script injection attacks, both part of the threat model we consider relevant for a secure session management mechanism to be deployed on an insecure channel. Two inherently more powerful attacker types, where either the network or client machine are under full control of the attacker, are considered to be out of scope. Below, we cover both in-scope attacker types and both out-of-scope attacker types in more detail.

First, an attacker can gain some control over the network, resulting in a *passive network attack*, also known as an eavesdropping attack, where he can steal a session identifier transmitted over an insecure channel. In the modern Web, with ubiquitous publicly available wifi networks, numerous wifi enabled devices, such as smartphones, and widely available, easy-to-use attacker tools, this type of attack is a common threat. Additionally, we consider the attacker to have access to a second, high-grade network connection, which allows him to eavesdrop on a request being sent, crafting a new request using certain properties and values from the original request, and have it reach the target server first.

Second, an attacker has the possibility to *exploit script injection vulnerabilities* in the target application, enabling the attacker to run custom scripts in the victim's browser, within the context of the target application. This attack vector is often used to steal a user's cookies, typically including the session cookie, and sending them to an attacker-controlled server, which can now impersonate the user to the target application. Note that the capability to inject custom scripts is sufficient to make the user's browser send attacker-controlled requests, but that session stealing attacks through script injection are extremely common on the Web, making this threat relevant and in-scope.

Two other attacker types in the Web context have significantly stronger capabilities, and are typically protected against on layers below session management, which is why we consider them to be out-of-scope in this context. One attacker type is the *active network attacker*, who is capable of performing full man-in-the-middle attacks on the user's connection. While being highly relevant, this type of attacker is typically countered by deploying TLS, which provides entity authentication, confidentiality and integrity. Explicitly protecting against active network attackers in our secure session management mechanism would lead to similar characteristics as TLS currently provides. Instead, we have opted to achieve compatibility with current and future TLS deployment scenarios, thereby further strengthening session management on the Web. Secondly, we do not consider *attacks on the client machine* to be in scope. Adversaries able to compromise a client machine, for example through malware installation or a man-in-the-browser attack, have unlimited control over the browser, its data and its requests.

B. Objectives for Secure Session Management

Based on the discussion of the current cookie-based session management mechanism and its associated threats, we identify four design objectives for a new session management mechanism. A first objective strongly focuses on security in light of the proposed threat model. The three remaining objectives are of a more practical nature, covering performance overhead, compatibility with the current infrastructure and a gradual migration path.

Secure Session Management: Within the proposed threat model, the state-of-practice session management mechanisms are vulnerable by design, due to their reliance on the session identifier as a bearer token. The key to achieve a secure session management mechanism is therefore to get rid of these bearer token properties, by anchoring the session to the parties involved in establishing it. By preventing the unauthorized transfer of an established session, eavesdropping and script injection attacks are effectively mitigated.

Minimal Overhead: A crucial requirement for a new session management mechanism in the Web is a minimal overhead, well illustrated by browser vendors and web application developers focusing on minimizing page load times. Overhead in the Web is twofold, with on one hand performance overhead, such as additional computations, and on the other hand networking overhead, with increased message sizes

and additional roundtrips. Especially the latter is considered problematic, since it delays the processing of the page and the loading of sub-resources, such as style sheets, images, etc.

Compatibility: A newly proposed session management mechanism should be compatible with the current Web and its peculiar deployment scenarios. Examples are the integration of third-party content in Web sites, and the redirection towards third-party service providers, such as a centralized authentication provider. On the infrastructure level, the Web deploys numerous middleboxes, such as web caches at various levels, content inspection systems at network perimeters, etc., which typically conflict with security measures offering confidentiality.

Gradual Migration: Finally, a new mechanism looking for adoption in the Web should support a gradual migration path, starting with early adopters on the client and server side, followed by a gradual increase of coverage in the Web. Key in this process is to aim for an application-agnostic session management mechanism, supporting implementation in current clients and server software or application development frameworks, thereby preventing the need for each individual application to incorporate the new mechanism. Additionally, backwards compatibility with parts of the Web that will not quickly adopt the new mechanism is also important, since the Web can not be updated in a single step.

C. Related work

The importance of securing session management is widely recognized, with several proposals offering different approaches to tackle the problem. While all of these approaches offer a secure session management mechanism, none of them meets all of the objectives for running a secure session management mechanism on an insecure channel. For example, several solutions require extensive changes to legacy applications before they can be deployed, or integrate tightly with the authentication process, making them them incompatible with several common web scenarios, such as third-party authentication providers. Additionally, most solutions depend on the mandatory use of TLS, which makes them unsuitable for deployment during a transition phase.

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 a non-TLS scenario, where the client performs an out-of-band Diffie-Hellman key exchange with the server. 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 [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 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, and additionally causes incompatibilities with current third-party authentication services.

One-Time Cookies: 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. Establishing the initial credential is done during the authentication phase, which is assumed to take place over a secure channel.

TLS Origin-Bound Certificates: Origin-Bound Certificates (OBC) [5] 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 difficult to deploy in a transition phase from HTTP to HTTPS.

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 over an insecure channel is the use of 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 protocol does not support web caches or out-of-order requests.

III. SECSESS SESSION MANAGEMENT

The essence of improving upon the current session management mechanism, is effectively binding the session to the initiating parties, abolishing the bearer token. Doing so ensures that the session can not be transferred without authorization of one of the initiating parties, protecting the session integrity, even when used on insecure channels.

A. Design

Our secure session management mechanism is illustrated in Figure 1. All session management instructions are contained in the newly introduced *Session* header, which keeps track of a session identifier (ID), as well as the parameters needed to provide the necessary security guarantees. SecSess 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 a message authentication code (HMAC) for a request, which is sent to the server in the *Session* header. Using the shared secret associated with the session, the server can calculate the

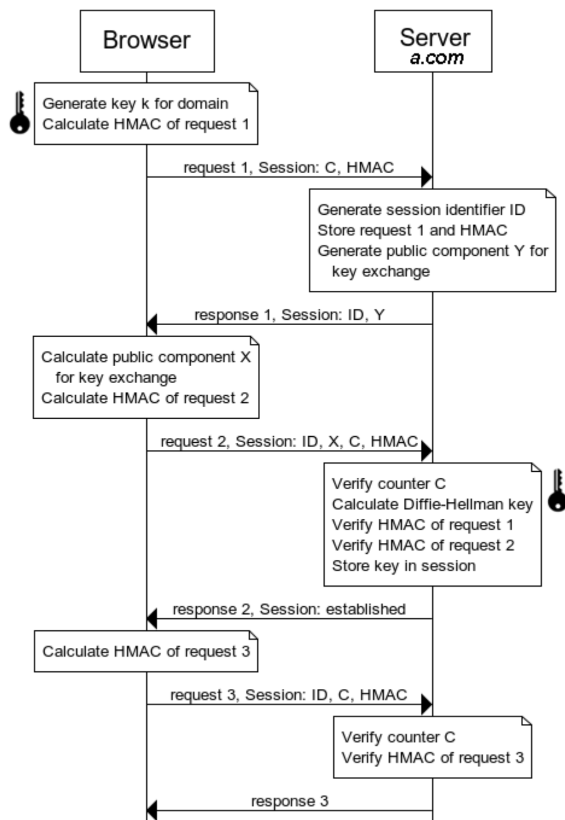

Fig. 1: The detailed flow of requests and responses used by SecSess.

same HMAC in order to verify whether the received request is actually valid within the session.

Note that the session identifier is no longer a bearer token, making its secrecy obsolete. Using a simple incremental counter as an identifier is sufficient. An attacker attempting to use a stolen session identifier also needs the session secret to generate valid HMACs for crafted requests. Since this shared secret is safely contained within the browser, it can not be obtained by an attacker.

We further detail each aspect of SecSess in four steps: (a) the actual session management mechanism, (b) how we establish the shared session secret, (c) generating and verifying request HMACs, and (d) preventing replay attacks. Figure 1 shows a detailed view of SecSess's session establishment.

a) 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). 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 the shared secret

b) Shared Session Secret: The shared session secret, needed to compute and verify HMACs on requests, is established using the Hughes variant [8], [13] of the Diffie-Hellman key exchange algorithm. The advantage of the Hughes variant is that one party can pre-calculate a key, which is then securely transferred to the second party. After running this protocol, the server shares a secret with the browser, without any information being disclosed to an eavesdropper.

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.

c) Request HMACs: When both parties possess a shared key, authentication codes can be attached to messages and verified afterwards 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 compute the correct values, indicating the authenticity of the request within the session, since it can only have been generated by the browser holding the shared secret. Incoming requests with an invalid HMAC are simply discarded.

Note that the input for the HMAC should be chosen carefully. Technically, a passive 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 the request headers and body (e.g. the destination account of a wire transfer). Therefore, the HMAC also covers the request headers, 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.

d) Replay Protection: Adding an HMAC to each request prevents an attacker from modifying request data, but does not protect against replaying 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) is part of the *Session* header, which is in turn protected by the request HMAC. In case verification of the replay protection token fails, the request is discarded.

B. Prototype Implementation

To show the feasibility of SecSess on the Web, we created a proof-of-concept implementation. The client-side component is implemented as a browser add-on, making it easy to integrate in an existing Firefox browser, and the server-side is implemented as a middleware component for the Express framework, running on top of Node.js.

The browser-addon is implemented in 706 source lines of code, of which 339 cover the protocol implementation itself.

We use the OpenSSL crypto library to perform the required cryptographic calculations and operations. The recently introduced *js-ctypes* allows to load a native library, thereby exposing the required native functions to the JavaScript environment.

The server-side middleware for the Express framework is implemented in 658 source lines of code, with a binary module interfacing towards the OpenSSL crypto library. The SecSess middleware can easily be added to a web server, enabling secure session management with a single line of code.

C. Evaluation

We performed a thorough evaluation of SecSess, of which we highlight some results in this section. The performance overhead is very limited[1], with an average 4.3 milliseconds overhead for establishing a new session, shared between client and server. There is one time-consuming step, where the server generates his protocol parameters, which includes the modular inverse of the private exponent, resulting in an average processing time of 212 ms. Fortunately, this step only needs to be done once, and can be precomputed during idle times, or loaded from a precomputed file.

Next to performance overhead, we also investigated the induced network overhead. By design, SecSess follows the same sequence of requests and responses used in existing applications, which deploy cookie-based session management mechanisms, so no additional requests or round trips are required. Furthermore, SecSess leads to a small reduction in average message sizes compared to typical cookie-based session management mechanisms (10 bytes on average), but requires the transmission of the public components, causing increased request sizes during establishment, and one response with an increased message size (both approx. 390 bytes).

IV. Deploying SecSess

Compared to the ubiquitous cookie-based session management mechanism, SecSess offers integrity-protected session management on the current Web, without incurring a significant overhead. In line with the final design objective, SecSess is application-agnostic, and can be deployed as part of the Web's infrastructure or development frameworks. By gradually implementing support in clients, servers and development frameworks, as our Express middleware does, SecSess can be enabled step by step, without loosing backward compatibility. Legacy applications can be protected by means of a reverse proxy, which translates between SecSess and cookie-based session management.

A. Web Caching

Web caches play an important role in the Web's infrastructure, boosting performance and limiting the required bandwidth for large networks. Contrary to most secure session management mechanisms, the design of SecSess explicitly takes caching 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 cached responses, which is fairly trivial once a session is established, but challenging during establishment. If the client's public component 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, effectively preventing the public component from getting lost in a request served from an intermediate cache.

The presence of middleboxes on the Web also sparks the discussion on integrity protection for responses. Technically, adding an HMAC to the response is straightforward, and based on our evaluation results, would not lead to a noticeable performance overhead. However, on the Web, responses are often modified in transit, for example for advertisement or censorship purposes. While we do not support these practices, they do occur, leading to a violation of the integrity, which in turn leads to client-side warnings and errors [3]. For these reasons, we chose to focus on session management, leaving response integrity protection a property for secure channels, such as HTTPS deployments.

B. Ongoing Changes to the Web

As the Web is continuously evolving, the underlying technology often changes, or generally accepted practices become frowned upon. Two such examples we highlight below are the pending upgrade of the HTTP protocol to HTTP/2.0, and the browser behavior in light of third-party cookies.

An upgrade of the HTTP/1.1 protocol by HTTP/2.0 aims to smooth out some rough edges of the HTTP protocol. Being based on the SPDY proposal [2] by Google, HTTP/2.0 keeps the semantics of the familiar HTTP protocol, such as URLs, requests and responses, but changes the representation of this content on the wire, using different types of frames, such as a *HEADERS* frame, a *DATA* frame, and several kinds of control frames. In this light, SecSess is compatible with HTTP/2.0, since it is applied on a higher level, before the request is encoded in HTTP/2.0 frames.

Another reality in the current Web is user tracking, mainly through third-party cookies. In recent discussions, the Firefox team contemplated disabling such third-party cookies, but eventually decided to postpone this decision [14]. Currently, SecSess follows the behavior of cookies, hence also supports sessions initiated by third-party content. Should it be desired to refrain from establishing sessions by means of third-party content, browser implementations can easily decide to limit the session establishment to first-party content, and only use already existing sessions on third-party content. This essentially limits the creation of sessions to sites that users voluntarily visit, but still allows widgets of these sites, embedded in other applications as third-party content, to operate in an authenticated manner.

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

ACKNOWLEDGEMENTS

This research is partially funded by the Research Fund KU Leuven, IWT and the EU-funded FP7 projects NESSoS, WebSand and STREWS, and is supported by the BRAIN-be programme of BELSPO.

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*

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] M. Belshe and R. Peon. Spdy protocol. *IETF Internet Draft*, 2012.

[3] M. Brinkmann. Firefox 4 supports content security policy. *Online at http://www.ghacks.net/2011/05/08/firefox-4-supports-content-security-policy/*, 2011.

[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] 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.

[6] D. Eastlake 3rd. Transport layer security (TLS) extensions: Extension definitions. *RFC 6066*, 2011.

[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] 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] I. Ristic. Internet ssl survey 2010. *Talk at BlackHat*, 2010.

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

[14] J. Temple. Firefox cookie blocking effort delayed again, as mozilla commitment wavers. *Online at http://blog.sfgate.com/techchron/2013/11/06/firefox-cookie-blocking-effort-delayed-again-as-mozilla-commitment-wavers/*, 2013.

[15] Trustworthy Internet Movement. Ssl pulse. *Online at https://www.trustworthyinternet.org/ssl-pulse/*, December 2013.