SHPF: Enhancing HTTP(S) Session Security with Browser Fingerprinting

Thomas Unger
FH Campus Wien
Vienna, Austria

Martin Mulazzani, Dominik Frühwirt,
Markus Huber, Sebastian Schrittwieser, Edgar Weippl
SBA Research
Vienna, Austria
Email: (1stletterfirstname)(lastname)@sba-research.org

***Abstract*—Session hijacking has become a major problem in today's Web services, especially with the availability of free off-the-shelf tools. As major websites like Facebook, Youtube and Yahoo still do not use HTTPS for all users by default, new methods are needed to protect the users' sessions if session tokens are transmitted in the clear.**
In this paper we propose the use of browser fingerprinting for enhancing current state-of-the-art HTTP(S) session management. Monitoring a wide set of features of the user's current browser makes session hijacking detectable at the server and raises the bar for attackers considerably. This paper furthermore identifies HTML5 and CSS features that can be used for browser fingerprinting and to identify or verify a browser without the need to rely on the UserAgent string. We implemented our approach in a framework that is highly configurable and can be added to existing Web applications and server-side session management with ease.

***Keywords*-Session Hijacking, Browser Fingerprinting, Security**

I. Introduction

Popular websites like Facebook or Yahoo, along with many others, use HTTPS-secured communication only for user authentication, while the rest of the session is usually transmitted in the clear. This allows an attacker to steal or copy the session cookies, identifiers or tokens, and to take over the session of the victim. Unencrypted Wi-Fi and nation-wide interceptors have used this as an attack vector multiple times recently, proving that session hijacking is indeed a problem for today's Internet security. To protect the session of a user, we implemented a framework that ties the session to the current browser by fingerprinting and monitoring the underlying browser, its capabilities, and detecting browser changes at the server side. Our framework, the *Session Hijacking Prevention Framework (SHPF)*, offers a set of multiple detection mechanisms which can be used independently of each other. SHPF protects especially against session hijacking of local adversaries, as well as against cross-site scripting (XSS). The underlying idea of our novel framework: If the user's browser suddenly changes from, e.g., Firefox on Windows 7 64 bit to an Android 4-based Webkit browser in a totally different IP range, we assume that some form of mischief is happening.

Our framework uses a diverse set of inputs and allows the website administrator to add SHPF with just a few additional lines of code in existing applications. There is no need to change the underlying Web application, and we can use the initial authentication process which is already part of many applications to build further security measurements on top. As part of the authentication process at the beginning of a session, the server asks the browser for an exact set of features and then monitors constantly whether the browser still behaves as expected over the entire session. While an attacker can easily steal unencrypted session information, e.g., on unencrypted Wi-Fi, it is hard to identify the exact responses needed to satisfy the server without access to the same exact browser version. Furthermore, we use a shared secret that is negotiated during the authentication, which is used to sign requests with an HMAC and a timestamp, building and improving on previous work in this direction. Recent attacks against HTTPS in general and the certificate authorities Diginotar and Comodo [8] in particular have shown that even the widespread use of SSL and HTTPS are not sufficient to protect against active adversaries and session hijacking. Previous work in the area of server-side session hijacking prevention relied, e.g., on a shared secret that is only known to the client's browser [1] and never transmitted in the clear. While this is a feasible approach and can protect a session even for unencrypted connections, our system extends this method by employing browser fingerprinting for session security, thus allowing us to incorporate and build upon existing security mechanisms like HTTPS. Furthermore, it offers protection against both passive and active adversaries. While the *OWASP AppSensor project* is a framework that offers similar features as SHPF for Web applications (e.g., detecting anomalies within a session and terminate it if necessary), it only uses a very limited set of checks compared to SHPF, namely the IP and the UserAgent string.

Our contributions in this paper are the following:

- We present a framework to enhance HTTP(S) session management, based on browser fingerprinting.
- We propose new browser fingerprinting methods for reliable browser identification based on CSS3 and HTML5.
- We extend and improve upon existing work on using a shared secret between client and server per session.

II. Background

Many administrators regard introducing SSL by default as too cost-intensive. Anecdotal evidence suggests that

naively enabling SSL without further configuration may incur significant performance degradation up to an order of magnitude. Gmail (with GMail) and Twitter use SSL by default for everyone, while Facebook is currently rolling it out (for North America only, so far). While HTTPS can be found on a large number of popular websites that require some form of authentication [5], only a minority of these binds the sessions to a user's device or IP address to protect the user against session hijacking [2]. Multiple tools have been released that allow automated session hijacking: *FaceNiff*, *DroidSheep*, *Firesheep*, *cookiemonster* and *sslstrip*, just to name a few. They allow an attacker (with multiple methods) to take over the victim's valid session and compromising the account in question. Most of them work by either sniffing for unencrypted packets containing valid session tokens, or forcing the client to send them in the clear.

III. Browser Fingerprinting

This section introduces our new browser fingerprinting methods, namely CSS and HTML5 fingerprinting. Furthermore, we present details on how we monitor HTTP headers in SHPF, which allows website administrators to configure advanced policies such as preventing an HTTP session from roaming between a tightly secured internal network and a public network beyond the control of the administrators. While browser fingerprinting has ambiguous meanings in the literature i.e., identifying the web browser down to the browser family and version number [3] vs. (re-)identifing a given user [9], we use the former.

A. CSS Fingerprinting

The upcoming (not yet fully standardized) CSS3 modules vary in stability and status. While some of them already have recommendation status, others are still candidate recommendations or working drafts. Browser vendors usually start implementing properties early, even long before they become recommendations. We identify three CSS-based methods of browser fingerprinting: CSS properties, CSS selectors and CSS filters. Differences in the layout engine allow us to identify a given browser by the CSS properties it supports. When properties are not yet on "Recommendation" or "Candidate Recommendation" status, browsers prepend a vendor-specific prefix indicating that the property is supported for this browser type only. Once a property moves to Recommendation status, prefixes are dropped by browser vendors and only the property name remains. For example, in Firefox 3.6 the property *border-radius* had a Firefox prefix resulting in *-moz-border-radius*, while in Chrome 4.0 and Safari 4.0 it was *-webkit-border-radius*. Since Firefox 4 as well as Safari 5.0 and Chrome 5.0, this feature is uniformly implemented as *border-radius*. The website https://www.caniuse.com shows a very good overview on how CSS properties are supported in the different browsers and their layout engine. Apart from CSS properties, browsers may differ in supported CSS selectors as well. Selectors are a way of selecting specific elements in an HTML tree. For example, CSS3 introduced new selectors for old properties, and they too are not yet uniformly implemented and can be used for browser fingerprinting. The third method of distinguishing browsers by their behavior is based on CSS filters. CSS filters are used to modify the rendering of e.g., a basic DOM element, image, or video by exploiting bugs or quirks in CSS handling for specific browsers, which again is very suitable for browser fingerprinting. Centricle[1] provides a good comparison of CSS filters across different browsers.

As CSS is used for styling websites it is difficult to compare rendered websites at the server side. Instead of conducting image comparison (as used recently by Mowery et al. [12] to fingerprint browsers based on WebGL-rendering), we use in our implementation JavaScript to test for CSS properties in style objects: in DOM, each element can have a style child object that contains properties for each possible CSS property and its value (if defined). There are now two ways to test CSS support of a property in the style object: the first way is to simply test whether the browser supports a specific property by using the *in* keyword on an arbitrary style object; the returning Boolean value indicates whether the property is supported. The second way to test whether a given CSS property is supported is to look at the value of a property once it has been set. We can set an arbitrary CSS property on an element and query the JavaScript *style* object afterwards. Interpreting the return values shows whether the CSS property is supported by the browser: *undefined (null)* as return value indicates that the property is not supported. If a not-*null* value is returned this means the property is supported and has been parsed successfully by the browser. The value string as such, returned upon querying the style object, also differs between browsers. It can be used as yet another test for fingerprinting based on CSS properties. For example, consider the following CSS3 background definition: *background:hsla(56, 100%, 50%, 0.3)*. While Firefox returns *none repeat scroll 0% 0% rgba(255, 238, 0, 0.3)*, Internet Explorer returns *hsla(56, 100%, 50%, 0.3)*. The order of elements within the return string for composite values may also deviate between browsers e.g., the *box-shadow* property with distance values as well as color definitions.

B. HTML5 Fingerprinting

HTML5, like CSS3, is still under development, but there are already working drafts which have been implemented to a large extend by different browsers. This new standard introduces some new tags, but also a wide range of new attributes. Furthermore HTML5 specifies new APIs (application programming interfaces), enabling the Web designer to use functionalities like drag and drop within websites. Since browser vendors

[1] http://centricle.com/ref/css/filters/

have differing implementation states of the new HTML5 features, support for the various improvements can be tested and used for fingerprinting purposes as well. For identifying the new features and to what extent they are supported by modern browsers, we used the methodology described in [14]. The W3C furthermore has a working draft on differences between HTML5 and HTML4 that was used as input [15].

In total we identified a set of 242 new tags, attributes and features in HTML5 that were suitable for browser identification. While 30 of these are attributed to new HTML tags that are introduced with HTML5[2], the rest of the new features consist of new attributes for existing tags as well as new features. We then created a website using the *Modernizr* library to test for each of these tags and attributes and whether they are supported by a given browser. We collected the output from close to 60 different browser versions on different operating systems. One of our findings from the fingerprint collection was that the operating system apparently has no influence on HTML5 support. We were unable to find any differences between operating systems while using the same browser version, even with different architectures.

C. Basic HTTP Header Monitoring

For each HTTP request, a number of HTTP headers is included and transmitted to the Web server. RFC 2616 defines the HTTP protocol [4] and specifies several HTTP headers that can or should be sent as part of each HTTP request. The number of headers, the contents and especially the order of the header fields, however, are chosen by the browser and are sufficient for identifying a browser. Using this method for browser identification has already been discussed in previous work [3], [16] and is already used to some extend by major websites [2], we will thus only briefly cover the parts which are of particular interest for SHPF. In our implementation we use the following header fields for HTTP session monitoring: *UserAgent string*, *Accept*, *Accept-Language*, *Accept-Encoding*, as well as the client's IP-Address.

The UserAgent contains information about the browser, often the exact browser version and the underlying operating system. It is, however, not a security feature, and can be changed arbitrarily by the user [13]. SHPF is not dependending on the UserAgent, and works with any string value provided by the browser. If the UserAgent changes during a session, however, this is a strong indication for session hijacking especially across different Web browsers. Depending on the configuration of SHPF and the particular security policy in place, it might however be acceptable to allow changes in the browser version e.g., with background updates of the browser while using a persistent session. Apart from the HTTP header values themselves, there is also a significant difference in how the browsers order the HTTP header fields. While Internet Explorer 9 for example sends the UserAgent before the Proxy-Connection and Host header fields, Chrome sends them in the exact opposite order. The content of the header fields is not important in this case.

If any values or a subset of these values change during a session, we assume that the session has been hijacked (in the simplest case). For example, if during a session multiple UserAgents from different IPs use the same session cookie, this implies in our framework that the session has been hijacked (session identifiers ought to be unique). The session would be terminated immediately and the user would need to reauthenticate. While cloning the HTTP headers is rather easy, binding a session to a given IP address considerably raises the bar for adversaries, even if they can obtain a valid session cookie and the full HTTP header.

IV. SHPF Framework

This section describes the implementation of our framework and its architecture, the *Session Hijacking Prevention Framework (SHPF)*. The source code is released under an open source license and can be found on github[3]. Despite the new fingerprinting methods presented in the previous section, we also implemented and improved SessionLock [1] for environments that do not use HTTPS by default for all connections.

A. General Architecture

SHPF is a server-side framework which is written in PHP5 and consists of multiple classes that can be loaded independently. Its general architecture and basic functionality is shown in Figure 1. We designed it to be easily configurable (depending on the context and the security needs of the website), portable and able to handle a possibly large number of concurrent sessions. Our implementation can be easily extended with existing and future fingerprinting methods, e.g., textfont rendering [12] or JavaScript engine fingerprinting [11], [13].

The main parts of the framework are the so-called *features*. A feature is a combination of different checks for detecting and mitigating session hijacking. In our prototype we implemented the following features: HTTP header monitoring, CSS fingerprinting and SecureSession (which implements and extends the SessionLock protocol by Ben Adida). Features are also the means to extend the framework, and we provide base classes for fast feature development. A feature consists of one or more *checkers*, which are used to run certain tests. There are two different types (or classes) of checkers: *Synchronous checkers* can be used if the tests included in the checker can be run solely from existing data and are passive in nature, such as HTTP requests or other website-specific data that is already available. *Asynchronous checkers* are

[2]http://www.w3schools.com/html5/html5_reference.asp

[3]https://github.com/mmulazzani/SHPF

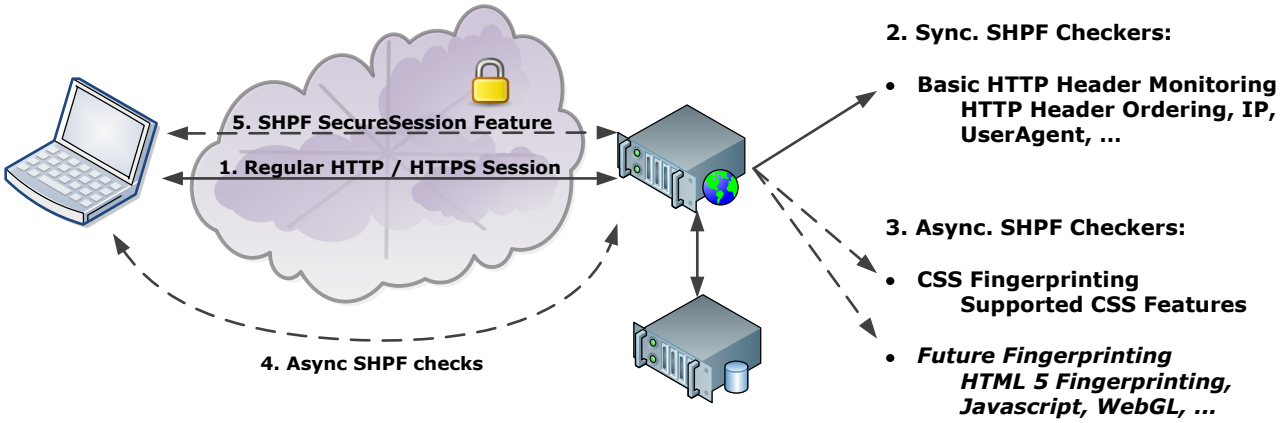

Figure 1. SHPF Architecture

used if the tests have to actively request some additional data from the client and the framework has to process the response. Client responses are sent via asynchronous calls (AJAX) as part of SHPF, thus not blocking the session or requiring to rewrite any existing code.

The distinction between features and checkers gives the website control over which checks to run. Features can be disabled or enabled according to the website's security needs, and can be assigned to different security levels. Different security levels within a webpage are useful, for example, in privacy-heterogeneous sessions - basic checks are performed constantly, while additional checks can be run only when necessary, e.g., when changing sensitive information in a user's profile (much like Amazon does for its custom session management). In order to communicate with a Web application, callbacks can be defined both in PHP and JavaScript. These callbacks are called if a checker fails and thus allow the application to react in an appropriate way, e.g., terminate the session and notify the user. Consider a for example a website e.g., a web store, which uses three different security levels for every session:

- Level 1 is for customers who are logged in and browsing the web store.
- Level 2 is for customers who are in a sensitive part of their session, e.g., ordering something or changing their profile.
- Level 3 is for administrators who are logged into the administrative interface.

Level 1 is a very basic security level. In this example it prevents session hijacking by monitoring the UserAgent string of the user for modifications. As a sole security measure it only protects the user against attacks that can be considered a nuisance, and can possibly be bypassed by an attacker (by cloning the UserAgent string). The Web application is designed in such a way that an attacker cannot actively do any harm to the user, for example browsing only specific products to manipulate the web store's recommendation fields. If the customer decides to buy something, level 2 is entered which uses two additional security measures: the current session is locked to the user's IP address and the order of the HTTP headers is monitored to detect if a different browser uses the same UserAgent string. Once the transaction is complete, the customer returns to level 1. For an administrator, even more checkers are enabled at the start of the session: SecureSession protects the session cryptographically with a shared secret between the particular browsers that started the sessions, and the CSS properties supported by the browser are monitored. Please note that this configuration is given merely by way of an example and must be matched to existing security policies when implemented. Furthermore, note that HTTPS is not mentioned in the example - even though it is strongly advised to use HTTPS during an entire session (besides SHPF), it is not a requirement. SHPF can prevent session hijacking even if session tokens are transmitted in the clear.

B. Basic HTTP Header Monitoring

The HTTP header monitoring feature does the following:

1) On the first request, the framework stores the contents and the order of the HTTP headers as described above.
2) For each subsequent request, the feature compares the headers sent by the client with the stored ones and checks whether their content and/or order match.

Depending on the particular use case, different configurations are possible, e.g., binding a session to a given IP, a certain IP range or a UserAgent string. Another example would be to allow IP address roaming while enforcing that the operating system as claimed by the UserAgent string as well as the browser has to stay the same, allowing the browser version to change, e.g., through background updates in Chrome or Firefox. HTTP header monitoring is implemented as a synchronous checker, as the data needed for processing is sent with every request.

C. CSS Fingerprinting

Using the CSS fingerprinting methods explained above, a SHPF feature has been implemented that does the following:

1) On the first request of the client: Run the complete fingerprinting suite on the client (using 23 CSS properties at the time of writing) and save the values.
2) For each subsequent request of the client: choose a subset of CSS properties and test them on the client.
3) Receive the data and check if it was requested by the framework (anti-replay protection).
4) Compare the values with the saved values.

As this feature needs data from the client, this checker has been implemented as an asynchronous checker. The client is challenged to answer a subset of the previously gathered properties either for each HTTP request or within a configurable interval between CSS checks (say, every 10 or 20 requests). By default, the framework tests three CSS properties and compares the results with the previously collected fingerprint of that browser. The data received asynchronously must match the requested properties and must arrive within a configurable time span. If the received data do not match the expected data, arrive too late or are not requested by the feature, the session is terminated immediately. If no response is received within a configurable time span, the session is terminated as well. SHPF may be also configured to terminate the session if no JavaScript is enabled, thus making CSS fingerprinting mandatory by policy.

In order to reliably identify a given browser, we selected a mix of CSS3 properties that are not uniformly supported by current browsers. In total, 23 properties were identified as suitable for fingerprinting which are shown in Table I. For our implementation of CSS fingerprinting in SHPF we chose to use CSS properties only - CSS selectors were not used because CSS properties are sufficient to reliably identify a given browser. Nonetheless, the framework could be extended by supporting CSS selector and CSS filter fingerprinting in the future.

D. SecureSession

The SecureSession feature implements the SessionLock protocol by Ben Adida [1], but extends and modifies it in certain aspects:

- SessionLock utilizes HTTPS to transfer the session secret to the client. In our SecureSession feature we use a Diffie-Hellman Key Exchange as discussed by Ben Adida in his paper.
- We use the new WebStorage[4] features implemented by modern browsers by using JavaScript and the *localStorage* object to store the session secret.
- We improved patching of URLs in JavaScript compared to the original protocol.

E. Further Fingerprinting Methods

Our framework is especially designed to allow new and possibly more sophisticated fingerprinting methods to be added at a later point in time by implementing them as additional checkers. The presented results on HTML5 fingerprinting above, e.g., have not yet been implemented at the time of writing. We are planning to implement HTML5 fingerprinting as an asynchronous checker in the near future. Other fingerprinting methods e.g., EFF's Panopticlick [3], can be added at ease. See Section VI-A for related work and other fingerprinting methods which could be added to SHPF.

V. Evaluation

A. Threat Model

An attacker in our threat model can be local or remote from the victim's point of view, as well as either active or passive. Figure 2 shows an overview of the different points of attack that were considered while designing SHPF. They are based on the OWASP Top 10[5], which has multiple categories that either directly allow session hijacking, or facilitate it. The most notable categories are "A2 Broken User Authentication and Session Management" and "A3 Cross-Site Scripting". We particularly considered threats that are actively exploited in the wild, with tools available for free.

The following points of attack allow an attacker to hijack a session:

1) Different attacks where the attacker has access to the victim's network connection.
2) The target website is vulnerable to code injection attacks (XSS), pushing malicious code to the client.
3) Local code execution within the victims browser's sandbox, e.g., by tricking the victim into executing Javascript (besides XSS).
4) Attacker has access to 3rd party server with access to the session token, e.g., a proxy, Tor exit node sniffing [10], [6] or via HTTP referrer string.

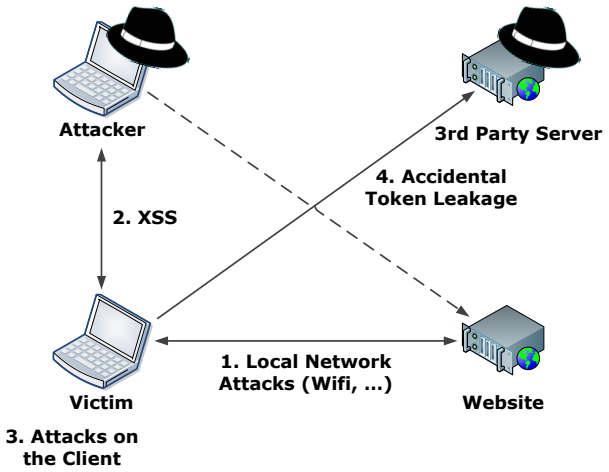

Figure 2. Attack Points for Session Hijacking

B. Discussion

To counter the attacks listed above, SHPF relies on a combination of its features: the shared secret between the server and client using the SecureSession feature, and

[4]http://www.w3.org/TR/webstorage/

[5]https://owasp.org/index.php/Top_10

CSS Status - Recommendation		CSS Status - Working Draft	
Feature	Value	Feature	Value
display	inline-block	transform	rotate(30deg)
min-width	35px	font-size	2rem
position	fixed	text-shadow	4px 4px 14px #969696
display	table-row	background	linear-gradient (left, red, blue 30%, green)
opacity	0.5	transition	background-color 2s linear 0.5s
background	hsla(56, 100%, 50%, 0.3)	animation	name 4s linear 1.5s infinite alternate none
CSS Status - Cand. Recommendation		resize	both
Feature	Value	box-orient	horizontal
box-sizing	border-box	transform-style	preserve-3d
border-radius	9px	font-feature-setting	dlig=1,ss01=1
box-shadow	inset 4px 4px 16px 10px #000	width	calc(25% -1em)
column-count	4	hyphens	auto
		object-fit	contain

Table I
23 CSS PROPERTIES AND VALUES IDENTIFIED FOR CSS FINGERPRINTING

session hijacking detection using browser fingerprinting. An attacker would thus have to find out the secret, share the same IP and copy the behavior of the victim's browser - either by running the same browser version on the same operating system or by collecting the behavior of the browser over time. SHPF does not intend to replace traditional security features for web sessions. While our approach cannot prevent session hijacking entirely it makes it harder for the attacker. For sensitive websites with a high need for security, additional measures like 2-factor authentication or client-side certificates should be employed.

The basic monitoring of HTTP information gives a baseline of protection. Binding a session to, e.g., an IP address makes it considerably harder for a remote attacker to attack, and a local attacker needs to be on the same local area network if the victim is behind NAT. Changes in the UserAgent or the HTTP header ordering are easily detectable, especially if careless attackers use sloppy methods for cloning header information, or only use some parts of the header for their user impersonation: Firesheep and FaceNiff, for example, both parse the header for session tokens instead of cloning the entire header. A recent manual analysis of the Alexa Top100 pages showed that only 8% of these very popular websites use basic monitoring in any form - notably eBay, Amazon and Apple [2]. Even though asynchronous challenges for fingerprinting on the attacker's machine could also simply be forwarded to the victim for the correct responses, the additional delay is detectable by the server.

SHPF has the following impact on the attack vectors: 1) Snooping or redirecting local network traffic can be detected at the server with either browser fingerprinting or using the shared secret, which is never transmitted in clear from SecureSession - both methods are equally suitable. 2) Cross-site scripting prevention relies on browser fingerprinting only, as the attacker could obtain session tokens by executing Javascript code in the victim's browser. The shared secret is not protected against such attacks. 3) Local attacks are also detected by browser fingerprinting only - the session secret is not safe, thus the attacker has to either run the same browser, or answer the asynchronous checks from the framework correctly. 4) Accidental token leakage is again prevented by both aspects, so even if the session is not encrypted by HTTPS the content is encrypted by the SecureSession feature and fingerprinting is used to detect changes in the used browser.

C. Limitations

Even though SHPF makes session hijacking harder, it has limitations: the HTTP headers and their ordering, as well as the UserAgent, are by no means security measures and can be set arbitrarily. However, if enough information specific to a browser is used in combination with ever shorter update intervals for browsers, we believe that fingerprinting is suitable for preventing session hijacking. Secondly, SHPF does not protect against CSRF: An attacker who is able to execute code outside of the browser's sandbox, or has access to the hardware, can bypass our framework. Thus session hijacking is made harder in the arms race with the adversary, but not entirely prevented. Another limitation is the vulnerability to man-in-the-middle attacks: Diffie-Hellman in Javascript for shared secret negotiation is vulnerable to MITM, and either a secure bootstrapping process for session establishment or offline multifactor authentication is needed to protect the session against such adversaries.

VI. RESULTS

In general, the performance impact of running SHPF on the server is negligible, as most of the processing is implemented as simple database lookups. Depending on the complexity of the underlying webapplication, it is expected to be just a few percent and below. A thorough analysis of the additional overhead was not conducted due to the lack of a standardized webapplication or performance benchmark for such a test. Only a few kilobytes of RAM are needed per session and client for all features combined, while the overhead on the network is around 100 kilobytes (mostly for the libraries used by our framework - they need to be transferred only once due to browser caching). A mere 15 lines of code are needed to include SHPF in existing websites, while the features

each consist of a few hundred lines of code on average, with SecureSession being by far the biggest feature (about 4000 lines). Existing Web applications implement far more complicated logic flows and information processing capabilities then SHPF.

A. Related Work

In the area of browser fingerprinting, different approaches have been used to identify a given browser. Panopticlick [3] relies on the feature combination of UserAgent string, screen resolution, installed plugins and more to generate a unique fingerprint that allows the tracking of a given browser even if cookies are disabled. Other recent work in the area of browser fingerprinting identifies a client's browser and its version as well as the underlying operating system by fingerprinting the JavaScript engine [11], [13]. Another recent method uses differences in website rendering [12]. Like SHPF, these methods allow the detection of a modified or spoofed UserAgent string, as it is not possible to change the behavior of core browser components like the rendering or the JavaScript engine within a browser. With regards to privacy, cookies and browser fingerprinting can be employed to track a user and their online activity. A survey on tracking methods in general can be found in [9]. Other work has recently shown that the UserAgent is often sufficient for tracking a user across multiple websites or sessions [16]. Session hijacking has been shown to allow access to sensitive information on social networking sites [7]. Finally, session hijacking is often conducted using cross-site scripting (XSS) attacks that are used to send the session information to an attacker. While this can be employed to protect a user from entering the password at an insecure terminal [2], it is often used maliciously, e.g., to impersonate the victim.

VII. Conclusion

In this paper, we presented our framework SHPF which is able to raise the bar for session hijacking. It detects and prevents attacks and hijacking attemps of various kinds, such as XSS or passive sniffing on the same network (Wi-Fi). We furthermore proposed two new browser fingerprinting methods based on HTML5 and CSS, which can identify a given browser. SHPF uses browser fingerprinting to detect session hijacking. SHPF can be configured to run with different security levels, allowing additional security checks for sensitive sessions or session parts. Future and upcoming fingerprinting methods can be incorporated easily.

Acknowledgements

Thanks to Peter Kieseberg, Manuel Leithner & Sebastian Neuner and for their feedback and helpful comments. This research was funded by the Austrian Research Promotion Agency under COMET K1.

References

[1] B. Adida. Sessionlock: Securing web sessions against eavesdropping. In *Proceeding of the 17th International Conference on World Wide Web (WWW)*, pages 517–524. ACM, 2008.

[2] E. Bursztein, C. Soman, D. Boneh, and J.C. Mitchell. Sessionjuggler: secure web login from an untrusted terminal using session hijacking. In *Proceedings of the 21st international conference on World Wide Web*, pages 321–330. ACM, 2012.

[3] P. Eckersley. How unique is your web browser? In *Proceedings of Privacy Enhancing Technologies (PETS)*, pages 1–18. Springer, 2010.

[4] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Rfc 2616: Hypertext transfer protocol–http/1.1, 1999. *Online at http://www.rfc.net/rfc2616.html*, 1999.

[5] R. Holz, L. Braun, N. Kammenhuber, and G. Carle. The SSL landscape: a thorough analysis of the x. 509 PKI using active and passive measurements. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, pages 427–444. ACM, 2011.

[6] M. Huber, M. Mulazzani, and E. Weippl. Tor HTTP Usage and Information Leakage. In *Communications and Multimedia Security*, pages 245–255. Springer, 2010.

[7] M. Huber, M. Mulazzani, and E. Weippl. Who On Earth Is Mr. Cypher? Automated Friend Injection Attacks on Social Networking Sites. In *Proceedings of the IFIP International Information Security Conference 2010: Security and Privacy (SEC)*, 2010.

[8] N. Leavitt. Internet security under attack: The undermining of digital certificates. *Computer*, 44(12):17–20, 2011.

[9] J.R. Mayer and J.C. Mitchell. Third-party web tracking: Policy and technology. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 413–427. IEEE, 2012.

[10] D. McCoy, K. Bauer, D. Grunwald, T. Kohno, and D. Sicker. Shining light in dark places: Understanding the tor network. In *Privacy Enhancing Technologies*, pages 63–76. Springer, 2008.

[11] K. Mowery, D. Bogenreif, S. Yilek, and H. Shacham. Fingerprinting information in javascript implementations. In *Proceedings of Web 2.0 Security & Privacy Workshop (W2SP)*, 2011.

[12] K. Mowery and H. Shacham. Pixel perfect: Fingerprinting canvas in html5. In *Proceedings of Web 2.0 Security & Privacy Workshop (W2SP)*, 2012.

[13] M. Mulazzani, P. Reschl, M. Huber, M. Leithner, S. Schrittwieser, and E. Weippl. Fast and reliable browser identification with javascript engine fingerprinting. In *Web 2.0 Workshop on Security and Privacy (W2SP)*, 5 2013.

[14] M. Pilgrim. *Dive into HTML5*. O'Reilly Media, 2010.

[15] A. van Kesteren. HTML 5 differences from HTML 4. *Working Draft, W3C*, 2013.

[16] T.F. Yen, Y. Xie, F. Yu, R.P. Yu, and M. Abadi. Host fingerprinting and tracking on the web: Privacy and security implications. In *Proceedings of the 19th Annual Network & Distributed System Security Symposium*. NDSS, 2012.