SessionShield: Lightweight Protection against Session Hijacking

Nick Nikiforakis[1], Wannes Meert[1], Yves Younan[1], Martin Johns[2], and Wouter Joosen[1]

[1] IBBT-DistriNet
Katholieke Universiteit Leuven
Celestijnenlaan 200A B3001
Leuven, Belgium
`{nick.nikiforakis,wannes.meert,yves.younan,wouter.joosen}@cs.kuleuven.be`
[2] SAP Research - CEC Karlsruhe
`martin.johns@sap.com`

Abstract. The class of Cross-site Scripting (XSS) vulnerabilities is the most prevalent security problem in the field of Web applications. One of the main attack vectors used in connection with XSS is session hijacking via session identifier theft. While session hijacking is a client-side attack, the actual vulnerability resides on the server-side and, thus, has to be handled by the website's operator. In consequence, if the operator fails to address XSS, the application's users are defenseless against session hijacking attacks.

In this paper we present SessionShield, a lightweight client-side protection mechanism against session hijacking that allows users to protect themselves even if a vulnerable website's operator neglects to mitigate existing XSS problems. SessionShield is based on the observation that session identifier values are not used by legitimate client-side scripts and, thus, need not to be available to the scripting languages running in the browser. Our system requires no training period and imposes negligible overhead to the browser, therefore, making it ideal for desktop and mobile systems.

Keywords: session hijacking, client-side proxy, http-only

1 Introduction

Over the past decade, users have witnessed a functional expansion of the Web, where many applications that used to run on the desktop are now accessible through the browser. With this expansion, websites evolved from simple static HTML pages to dynamic Web applications, i.e. content-rich resources accessible through the Web. In this modern Web, JavaScript has proven its usefulness by providing server offloading, asynchronous requests and responses and in general improving the overall user experience of websites. Unfortunately, the de facto

support of browsers for JavaScript opened up the user to a new range of attacks, of which the most common is Cross-site scripting (XSS[3]).

In XSS attacks, an attacker convinces a user's browser to execute malicious JavaScript code on his behalf by injecting this code in the body of a vulnerable webpage. Due to the fact that the attacker can only execute JavaScript code, as opposed to machine code, the attack was initially considered of limited importance. Numerous incidents though, such as the Sammy worm that propagated through an XSS vulnerability on the social network MySpace [34] and the XSS vulnerabilities of many high-impact websites (e.g., Twitter, Facebook and Yahoo [40]) have raised the awareness of the security community. More recently, Apache released information about an incident on their servers where attackers took advantage of an XSS vulnerability and by constant privilege escalation managed to acquire administrator access to a number of servers [1].

Today, the Open Web Application Security Project (OWASP) ranks XSS attacks as the second most important Web application security risk [28]. The Web Hacking Incident Database from the Web Application Security Consortium states that 13.87% of all attacks against Web applications are XSS attacks [4]. These reports, coupled with more than 300,000 recorded vulnerable websites in the XSSed archive [40], show that this problem is far from solved.

In this paper, we present SessionShield, a lightweight countermeasure against session hijacking. Session hijacking occurs when an attacker steals the session information from a legitimate user for a specific website and uses it to circumvent authentication to that website. Session hijacking is by far the most popular type of XSS attack since every website that uses session identifiers is potentially vulnerable to it. Our system is based on the observation that session identifiers are strings of data that are intelligible to the Web application that issued them but not to the Web client who received them. SessionShield is a proxy outside of the browser that inspects all outgoing requests and incoming responses. Using a variety of methods, it detects session identifiers in the incoming HTTP headers, strips them out and stores their values in its own database. In every outgoing request, SessionShield checks the domain of the request and adds back the values that were previously stripped. In case of a session hijacking attack, the browser will still execute the session hijacking code, but the session information will not be available since the browser never received it. Our system is transparent to both the Web client and the Web server, it operates solely on the client-side and it doesn't rely on the Web server or trusted third parties. SessionShield imposes negligible overhead and doesn't require training or user interaction making it ideal for both desktop and mobile systems.

The rest of this paper is structured as follows: In Section 2, we describe what sessions are and how XSS attacks are conducted followed by a detailed survey concerning a well-known protection mechanism, namely HTTP-Only cookies. In Section 3, we present the architecture and details of SessionShield. We evaluate

---

[3] Cross-site scripting is commonly abbreviated as XSS to distinguish it from the acronym of Cascading Style Sheets (CSS)

our system in Section 4 and provide implementation details in Section 5. We discuss related work in Section 6 and finally conclude in Section 7.

2 Background

2.1 Session Identifiers

The workhorse protocol of the World Wide Web, the HyperText Transfer Protocol (HTTP) and its secure counterpart (HTTP**S**) are by design stateless. That means that a Web application cannot track a client between multiple requests unless it adds a separate tracking mechanism on top of the HTTP(S) protocol. The most commonly used tracking mechanism are sessions identifiers. A session identifier (SID) is a unique string of random data (typically consisting of numbers and characters) that is generated by a Web application and propagated to the client, usually through the means of a cookie. After the propagation of the session, every request initiated by the client will contain, among others, the session identifier that the application entrusted him with. Using session identifiers, the Web application is able to identify individual users, distinguish simultaneously submitted requests and track the users in time. Sessions are used in e-banking, web-mail and virtually every non-static website that needs to enforce access-control on its users. Sessions are an indispensable element of the modern World Wide Web and thus session management support exists in all modern Web languages (e.g., PHP, ASP and JSP).

Session identifiers are a prime attack target since a successful capture-and-replay of such an identifier by an attacker provides him with instant authentication to the vulnerable Web application. Depending on the access privileges of the user whose id was stolen, an attacker can login as a normal or as a privileged user on the website in question and access all sorts of private data ranging from emails and passwords to home addresses and even credit card numbers. The most common way of stealing session identifiers is through Cross-site Scripting attacks which are explained in the following section.

2.2 Cross-Site Scripting attacks

Cross-site scripting (XSS) attacks belong to a broader range of attacks, collectively known as code injection attacks. In code injection attacks, the attacker inputs data that is later on perceived as code and executed by the running application. In XSS attacks, the attacker convinces the victim's browser to execute JavaScript code on his behalf thus giving him access to sensitive information stored in the browser. Malicious JavaScript running in the victim's browser can access, among others, the contents of the cookie for the running domain. Since session identifiers are most commonly propagated through cookies, the injected JavaScript can read them and transfer them to an attacker-controlled server which will record them. The attacker can then replay these sessions to the vulnerable website effectively authenticating himself as the victim.

XSS vulnerabilities can be categorized as *reflected* or *stored*. A reflected XSS vulnerability results from directly including parts of the HTTP request into the corresponding HTTP response. Common examples for reflected XSS issues include search forms that blindly repeat the search term on the results-page or custom 404 error pages. On the other hand, a stored XSS vulnerability occurs whenever the application permanently stores untrusted data which was not sufficiently sanitized. If such data is utilized to generate an HTTP response, all potentially injected markup is included in the resulting HTML causing the XSS issue. Stored XSS was found in the past for instance in guestbooks, forums, or Web mail applications.

Code listing 1 shows part of the code of a search page, written in PHP, that is vulnerable to a reflected XSS attack. The purpose of this page is to read one or more keywords from the user, search the database for the keyword(s) and show the results to the user. Before the actual results, the page prints out the keyword(s) that the user searched for. The programmer however has not provisioned against XSS attacks, and thus whatever is presented as a query, will be "reflected" back in the main page, including HTML and JavaScript code. An attacker can hijack the victim's session simply by sending him the following link:

```
http://vulnerable.com/search.php?q=</u><script>
document.write('<img src="http://hacker.com/
session_hijack.php?ck=' + document.cookie +'">');
</script>
```

When the user clicks on the above link, his browser will initiate a `GET` request to `vulnerable.com`. The `GET` parameter `q` will be added to the resulting page that the server sends back to the user's browser. The victim's browser will start rendering the page and once it reaches the "Search results for:" part, it will create an image URI which contains the values stored in the user's cookie and ask for that image from the attacker-controlled server. The attacker's script will record the cookie values and enable the attacker to masquerade as the victim at the `vulnerable.com`.

Code Listing 1 Code snippet vulnerable to an XSS attack

```php
<?php
  session_start();
  ...
  $search_query = $_GET['q'];
  print "Search results for: <u> $search_query </u>";
  ...
?>
```

2.3 HTTP-Only and Sessions

Developers realized from early on that it is trivial to hijack Web sessions in the presence of an XSS vulnerability. In 2002, Microsoft developers introduced the notion of `HTTP-Only` cookies and added support for them in the release of Internet Explorer 6, SP1 [22]. `HTTP-Only` is a flag that is sent by the Web application to the client, along with a cookie that contains sensitive information, e.g., a session identifier. It instructs the user's browser to keep the values of that cookie away from any scripting languages running in the browser. Thus, if a cookie is denoted as `HTTP-Only` and JavaScript tries to access it, the result will be an empty string. We tested the latest versions of the five most common Web browsers (Internet Explorer, Firefox, Chrome, Safari and Opera) and we observed that if the Web application emmits the `HTTP-Only` flag the cookie is, correctly, no longer accessible through JavaScript.

In an attempt to discover whether the `HTTP-Only` mechanism is actually used, we crawled the Alexa-ranked top one million websites [35] and recorded whether cookies that contained the keyword "sess" where marked as `HTTP-Only`. We chose "sess" because it is a common substring present in the session names of most major Web languages/frameworks (see Section 3.2, Table 2) and because of the high probability that customely named sessions will still contain that specific substring. We also provisioned for session names generated by the ASP/ASP.NET framework that don't contain the "sess" string. The results of our crawling are summarized in Table 1. Out of 1 million websites, 418,729 websites use cookies in their main page and out of these, 272,335 cookies contain session information[4]. We were surprised to find out that only a 22.3% of all websites containing sessions protected their cookies from session stealing using the `HTTP-Only` method. Further investigation shows that while 1 in 2 ASP websites that use sessions utilize `HTTP-Only`, only 1 in 100 PHP/JSP websites does the same.

These results clearly show that `HTTP-Only` hasn't received widespread adoption. Zhou et al. [42] recently made a similar but more limited study (top 500 websites, instead of top 1 million) with similar findings. In their paper they acknowledge the usefullness of the `HTTP-Only` mechanism and they discuss possible reasons for its limited deployment.

3 SessionShield Design

SessionShield is based on the idea that session identifiers are data that no legitimate client-side script will use and thus should not be available to the scripting languages running in the browser. Our system shares this idea with the `HTTP-Only` mechanism but, unlike `HTTP-Only`, it can be applied selectively to a subset of cookie values and, more important, it doesn't need support from

---

[4] Cookies that contained the `HTTP-Only` flag but were not identified by our heurestic are added to the "Other/With HTTP-Only" column.

Session Framework	Total	With HTTP-Only	Without HTTP-Only
PHP	135,117 (53.2%)	1,736 (1.3%)	133,381 (98.7%)
ASP/ASP.NET	60,218 (23.5%)	25,739 (42.7%)	34,479 (57.3%)
JSP	12,911 (5.1%)	113 (0.9%)	12,798 (99.1%)
Other	64,089 (18.2%)	33,071 (51.6%)	31,018 (48.4%)
Total	272,335 (100%)	60,659 (22.3%)	211,676 (77.8%)

Table 1: Statistics on the usage of HTTP-Only on websites using session identifiers, sorted according to their generating Web framework

Web applications. This means, that SessionShield will protect the user from session hijacking regardless of the security provisioning of Web operators.

The idea itself is founded on the observation that session identifiers are strings composed by random data and are unique for each visiting client. Furthermore, a user receives a different session identifier every time that he logs out from a website and logs back in. These properties attest that there can be no legitimate calculations done by the client-side scripts using as input the constantly-changing random session identifiers. The reason that these values are currently accessible to client-side scripts is because Web languages and frameworks mainly use the cookie mechanism as a means of transport for the session identifiers. The cookie is by default added to every client request by the browser which aleviates the Web programmers from having to create their own transfer mechanism for session identifiers. JavaScript can, by default, access cookies (using the `document.cookie` method) since they may contain values that the client-side scripts legitimately need, e.g., language selection, values for boolean variables and timestamps.

3.1 Core Functionality

Our system acts as a personal proxy, located on the same host as the browser(s) that it protects. In order for a website or a Web application to set a cookie to a client, it sends a `Set-Cookie` header in its HTTP response headers, followed by the values that it wishes to set. SessionShield inspects incoming data in search for this header. When the header is present, our system analyses the values of it and attempts to discover whether session identifiers are present. If a session identifier is found, it is stripped out from the headers and stored in SessionShield's internal database. On a later client request, SessionShield queries its internal database using the domain of the request as the key and adds to the outgoing request the values that it had previously stripped.

A malicious session hijacking script, whether reflected or stored, will try to access the cookie and transmit its value to a Web server under the attacker's control. When SessionShield is used, cookies inside the browser no longer contain session identifiers and since the attacker's request domain is different from the

domain of the vulnerable Web application, the session identifier will not be added to the outgoing request, effectively stopping the session hijacking attack.

In order for SessionShield to protect users from session hijacking it must successfully identify session identifiers in the cookie headers. Our system uses two identification mechanisms based on: a) common naming conventions of Web frameworks and of custom session identifiers and b) statistical characteristics of session identifiers.

3.2 Naming Conventions of Session Identifiers

Common Web Frameworks Due to the popularity of Web sessions, all modern Web languages and frameworks have support for generating and handling session identifiers. Programmers are actually adviced not to use custom session identifiers since their implementation will most likely be less secure from the one provided by their Web framework of choice. When a programmer requests a session identifier, e.g., with `session_start()` in PHP, the underlying framework generates a random unique string and automatically emmits a `Set-Cookie` header containing the generated string in a `name=value` pair, where `name` is a standard name signifying the framework used and `value` is the random string itself. Table 2 shows the default names of session identifiers according to the framework used[5]. These naming conventions are used by SessionShield to identify session identifiers in incoming data and strip them out of the headers.

Common Custom Naming From the results of our experiment in Section 2.3, we observed that "sess" is a common keyword among custom session naming and thus it is included as an extra detection method of our system. In order to avoid false-positives we added the extra measure of checking the length and the contents of the value of such a pair. More specifically, SessionShield identifies as session identifiers pairs that contain the word "sess" in their name and their value is more than 10 characters long containing both letters and numbers. These characteristics are common among the generated sessions of all popular frameworks so as to increase the value space of the identifiers and make it practically impossible for an attacker to bruteforce a valid session identifier.

3.3 Statistical Characteristics of session identifiers

Despite the coverage offered by the previous mechanism, it is beyond doubt that there can be sessions that do not follow standard naming conventions and thus would not be detected by it. In this part we focus on the fact that session identifiers are long strings of symbols generated in some random way. These two key characteristics, length and randomness, can be used to predict if a string, that is present in a cookie, is a session identifier or not. This criterion, in fact, is similar to predicting the strength of a password.

---

[5] On some versions of the ASP/ASP.NET framework the actual name contains random characters, which are signified by the wildcard symbol in the table.

Session Framework	Name of Session variable
PHP	phpsessid
ASP/ASP.NET	asp.net_sessionid
	aspsessionid*
	.aspxauth*
	.aspxanonymous*
JSP	jspsessionid
	jsessionid

Table 2: Default session naming for the most common Web frameworks

Three methods are used to predict the probability that a string is a session identifier (or equivalently the strength of a password):

1. **Information entropy**: The strength of a password can be measured by the information entropy it represents [6]. If each symbol is produced independently, the entropy is $H = L \cdot \log_2 N$, with N the number of possible symbols and L the length of the string. The resulting value, H, gives the entropy and represents the number of bits necessary to represent the string. The higher the number of necessary bits, the better the strength of the password in the string. For example, a pin-code consisting out of four digits has an entropy of 3.32 bits per symbol and a total entropy of 13.28.

2. **Dictionary check**: The strength of a password reduces if it is a known word. Similarly, cookies that have known words as values are probably not session identifiers.

3. χ^2: A characteristic of a random sequence is that all symbols are produced by a generator that picks the next symbol out of a uniform distribution ignoring previous symbols. A standard test to check if a sequence correlates with a given distribution is the χ^2-test [16], and in this case this test is used to calculate the correlation with the uniform distribution. The less the string is correlated with the random distribution the less probable it is that it is a random sequence of symbols. The uniform distribution used is $1/N$, with N the size of the set of all symbols appearing in the string.

Every one of the three methods returns a probability that the string is a session identifier. These probabilities are combined by means of a weighted average to obtain one final probability. SessionShield uses this value and a threshold to differentiate between session and non-session values (see Section 5 for details).

4 Evaluation

4.1 False Positives and False Negatives

SessionShield can protect users from session hijacking as long as it can successfully detect session identifiers in the incoming HTTP(S) data. In

order to evaluate the security performance of SessionShield we conducted the
following experiment: we seperated the first 1,000 cookies from our experiment
in Section 2.3 and we used them as input to the detection mechanism of
SessionShield. SessionShield processed each cookie and classified a subset of the
values as sessions identifiers and the rest as benign data. We manually inspected
both sets of values and we recorded the false positives (values that were wrongly
detected as session identifiers) and the false negatives (values that were not
detected as session identifiers even though they were). SessionShield classified
2,167 values in total (average of 2.16 values/cookie) with 70 false negatives (3%)
and 19 false positives (0,8%).

False negatives were mainly session identifiers that did not comply to our
session identifier criteria, i.e a) they didn't contain both letters and numbers or b)
they weren't longer than 10 characters. Session identifiers that do not comply to
these requirements are easily brute-forced even if SessionShield protected them.
With regard to false positives, it is important to point out that in order for a
website to stop operating correctly under SessionShield, its legitimate client-side
scripts must try to use values that SessionShield classified as session identifiers.
Thus the actual percentage of websites that wouldn't operate correctly and would
need to be white-listed is less than or equal to 0,8%.

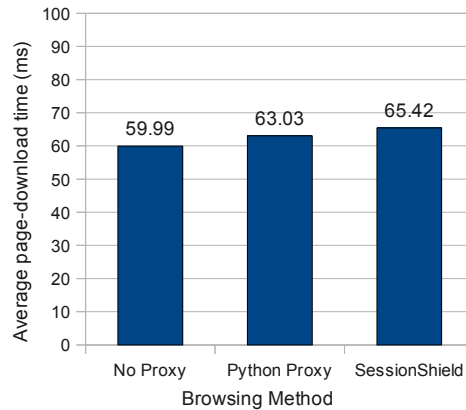## 4.2 Performance Overhead

Fig. 1: Average download time of the top 1,000 websites when accessed locally without
a proxy, with a simple forwarding Python-proxy and with SessionShield

In an effort to quantify how much would SessionShield change the Web
experience of users, we decided to measure the difference in page-download time
when a page is downloaded: a) directly from the Internet; b) through a simple
forwarding proxy [9] and c) through SessionShield. Using `wget`, we downloaded
the top 1,000 Internet websites [35] and measured the time for each.

In order to avoid network inconsistencies we downloaded the websites locally together with the HTTP headers sent by the actual Web servers. We used a fake DNS server that always resolved all domains to the "loopback" IP address and a fake Web server which read the previously-downloaded pages from disk and replayed each page along with its original headers. This allowed us to measure the time overhead of SessionShield without changing its detection technique, which relies on the cookie-related HTTP headers. It is important to point out that SessionShield doesn't add or remove objects in the HTML/JavaScript code of each page thus the page-rendering time isn't affected by its operation. Each experiment was repeated five times and the average page-download time for each method is presented in Fig. 1. SessionShield's average time overhead over a simple Python proxy is approximately 2.5 milliseconds and over a non-proxied environment is 5.4 milliseconds. Contrastingly, popular Web benchmarks show that even the fastest websites have an average page-download time of 0.5 seconds when downloaded directly from the Internet [38].

Since our overhead is two orders of magnitude less than the fastest page-download times we believe that SessionShield can be used by desktop and mobile systems without perceivable performance costs.

5 Implementation

We decided to prototype SessionShield using Python. We used an already implemented Python proxy, TinyHTTPProxy [9], and added the session detection mechanisms that were described in Section 3. The advantage of implementing SessionShield as a stand-alone personal proxy instead of a browser-plugin relies on the fact that cookies residing in the browser can still be attacked, e.g. by a malicious add-on [18]. When sensitive data are held outside of the browser, in the data structures of the proxy, a malicious add-on will not be able to access them. On the other hand, a browser plugin can transparently support HTTPS and provides a more user-friendly install and update procedure.

The threshold value of SessionShield, mentioned in Section 3.3, was obtained by using the known session identifiers from our `HTTP-Only` experiment in Section 2.3 as input to our statistical algorithm and observing the distribution of the reported probabilities.

6 Related Work

Client-side approaches for mitigating XSS attacks: Noxes [15] is a defensive approach closely related to ours – A client-side Web proxy specifically designed to prevent session identifier (SID) theft. Unlike our approach, Noxes does not prevent the injected JavaScript to access the SID information. Instead, Noxes aims to deprive the adversary from the capability to leak the SID value outside of the browser. The proposed technique relies on the general assumption that dynamically assembled requests to external domains are potentially untrustworthy as they could carry stolen SID values. In consequence,

such requests are blocked by the proxy. Besides the fact that this implemented policy is incompatible with several techniques from the Web 2.0 world, e.g., Web widgets, the protection provided by Noxes is incomplete: For example, the authors consider static links to external domains to be safe, thus, allowing the attacker to create a subsequent XSS attack which, instead of `script`-tags, injects an HTML-tag which statically references a URL to the attackers domain including the SID value.

Vogt et al. [36] approach the problem by using a combination of static analysis and dynamic data tainting within the browser to track all sensitive information, e.g., SID values, during JavaScript execution. All outgoing requests that are recognised to contain such data are blocked. However, due to the highly dynamic and heterogenous rendering process of webpages, numerous potential hidden channels exist which could lead to undetected information leaks. In consequence, [32] exemplified how to circumvent the proposed technique. In comparison, our approach is immune to threats through hidden channels as the SID never enters the browser in the first place.

Furthermore, browser-based protection measures have been designed that disarm reflected XSS attacks through comparing HTTP requests and responses. If a potential attack is detected, the suspicious code is neutralized on rendering-time. Examples for this approach include NoScript for Firefox [20], Internet Explorer's XSS Filter [31], and XSSAuditor for Chrome [3]. Such techniques are necessarily limited: They are only effective in the presence of a direct, character-level match between the HTTP request and its corresponding HTTP response. All non-trivial XSS vulnerabilities are out of scope. In addition, it is not without risk to alter the HTTP response in such ways: For instance, Nava & Lindsay [25] have demonstrated, that the IE XSS Filter could cause XSS conditions in otherwise secure websites. Finally, to confine potentially malicious scripts, it has been proposed to whitelist trusted scripts and/or to declare untrusted regions of the DOM which disallow script execution [10, 23, 5, 21]. All of these techniques require profound changes in the browser's internals as well as the existence of server-side policy information.

Security enhancing client-side proxies: Besides Noxes, further client-side proxies exist, that were specifically designed to address Web application vulnerabilities:

RequestRodeo [12] mitigates Cross-site Request Forgery attacks through selectively removing authentication credentials from outgoing HTTP requests. As discussed in this paper, the majority of all Web applications utilize the SID as the de facto authentication credential. In consequence, RequestRodeo (and its further refinements, such as [33]) could benefit from our SID detection algorithm (see Section 3) in respect to false positive reduction.

HProxy [27] is a client-side proxy which protects users from SSL stripping attacks and from malicious JavaScript code that a Man-In-The-Middle (MITM) attacker could have added in a page to steal Web credentials. In order to differentiate between original and "added" JavaScript, HProxy takes advantage of the user's browsing habits to generate a template of each script, recording the static and the dynamic parts of it. If, at a later time, a script is detected

which doesn't adhere to its original template, HProxy marks it as an attack and doesn't forward it to the browser.

Server-side approaches: The majority of existing XSS prevention and mitigation techniques take effect on the Web application's server-side. We only give a brief overview on such related work, as this paper's contributions specifically address client-side protection:

Several approaches, e.g., [29, 26, 8, 41], employ dynamic taint tracking of untrusted data on run-time to identify injection attacks, such as XSS. Furthermore, it has been shown that static analysis of the application's source code is a capable tool to identify XSS issues (see for instance [17, 39, 14, 37, 7]). Moreover, frameworks which discard the insecure practice of using the string type for syntax assembly are immune against injection attacks through providing suitable means for data/code separation [30, 11]. Jovanovic et al. [13] use a server-side proxy which rewrites HTTP(S) requests and responses in order to detect and prevent Cross-site Request Forgery. Finally, cooperative approaches spanning server and browser have been described in [2, 19, 24].

7 Conclusion

Session hijacking is the most common Cross-site Scripting attack. In session hijacking, an attacker steals session-containing cookies from users and utilizes the session values to impersonate the users on vulnerable Web applications. In this paper we presented SessionShield, a lightweight client-side protection mechanism against session hijacking. Our system, is based on the idea that session identifiers are not used by legitimate client-side scripts and thus shoudn't be available to the scripting engines running in the browser. SessionShield detects session identifiers in incoming HTTP traffic and isolates them from the browser and thus from all the scripting engines running in it. Our evaluation of SessionShield showed that it imposes negligible overhead to a user's system while detecting and protecting almost all the session identifiers in real HTTP traffic, allowing its widespread adoption in both desktop and mobile systems.

8 Acknowledgments

References

1. Apache.org. `https://blogs.apache.org/infra/entry/apache_org_04_09_2010`.
2. Elias Athanasopoulos, Vasilis Pappas, Antonis Krithinakis, Spyros Ligouras, Evangelos P. Markatos, and Thomas Karagiannis. xjs: Practical xss prevention for web application development. In *Proceedings of the 1st USENIX Conference on Web Application Development (WebApps'10)*, 2010.

3. Daniel Bates, Adam Barth, and Collin Jackson. Regular expressions considered harmful in client-side XSS filters. In *Proceedings of the 19th international conference on World wide web (WWW '10)*, New York, NY, USA, 2010. ACM.

4. Web Application Security Consortium. Web Hacking Incident Database.

5. Ulfar Erlingsson, Benjamin Livshits, and Yinglian Xie. End-to-end Web Application Security. In *Proceedings of the 11th Workshop on Hot Topics in Operating Systems (HotOS'07)*, May 2007.

6. Dinei Florencio and Cormac Herley. A large-scale study of web password habits. In *Proceedings of the 16th international conference on World Wide Web (WWW '07)*, New York, NY, USA, 2007. ACM.

7. Emmanuel Geay, Marco Pistoia, Takaaki Tateishi, Barbara Ryder, and Julian Dolby. Modular String-Sensitive Permission Analysis with Demand-Driven Precision. In *Proceedings of the 31st International Conference on Software Engineering (ICSE 2009)*, 2009.

8. William G.J. Halfond, Alessandro Orso, and Panagiotis Manolios. Using Positive Tainting and Syntax-Aware Evaluation to Counter SQL Injection Attacks. In *Proceedings of the 14th ACM Symposium on the Foundations of Software Engineering (FSE)*, 2006.

9. Suzuki Hisao. Tiny HTTP Proxy in Python.

10. Trevor Jim, Nikhil Swamy, and Michael Hicks. Defeating Script Injection Attacks with Browser-Enforced Embedded Policies. In *Proceedings of the 16th International World Wide Web Conference (WWW '07)*, May 2007.

11. Martin Johns, Christian Beyerlein, Rosemaria Giesecke, and Joachim Posegga. Secure Code Generation for Web Applications. In *Proceedings of the 2nd International Symposium on Engineering Secure Software and Systems (ESSoS '10)*. Springer, February 2010.

12. Martin Johns and Justus Winter. RequestRodeo: Client Side Protection against Session Riding. In *Proceedings of the OWASP Europe 2006 Conference*, 2006.

13. Nenad Jovanovic, Engin Kirda, and Christopher Kruegel. Preventing cross site request forgery attacks. In *Proceedings of IEEE International Conference on Security and Privacy for Emerging Areas in Communication Networks (Securecomm)*, 2006.

14. Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities. In *IEEE Symposium on Security and Privacy*, May 2006.

15. Engin Kirda, Christopher Kruegel, Giovanni Vigna, and Nenad Jovanovic. Noxes: A Client-Side Solution for Mitigating Cross Site Scripting Attacks. In *Security Track of the 21st ACM Symposium on Applied Computing (SAC 2006)*, April 2006.

16. Donald E. Knuth. *The Art of Computer Programming, Volume 2*. Addison-Wesley Publishing Company, 1971.

17. Benjamin Livshits and Monica S. Lam. Finding Security Vulnerabilities in Java Applications Using Static Analysis. In *Proceedings of the 14th USENIX Security Symposium*, August 2005.

18. Mike Ter Louw, Jin Soon Lim, and V. N. Venkatakrishnan. Extensible Web Browser Security. In *4th International Conference on Detection of Intrusions, Malware, and Vulnerability Assesment (DIMVA)*, 2007.

19. Mike Ter Louw and V.N. Venkatakrishnan. BluePrint: Robust Prevention of Cross-site Scripting Attacks for Existing Browsers. In *IEEE Symposium on Security and Privacy (Oakland'09)*, May 2009.

20. Giorgio Maone. NoScript Firefox Extension, 2006.

21. Leo A. Meyerovich and Benjamin Livshits. Conscript: Specifying and enforcing fine-grained security policies for javascript in the browser. In *Proceedings of 31st IEEE Symposium on Security and Privacy (SP '10)*, 2010.
22. Microsoft. Mitigating Cross-site Scripting With HTTP-only Cookies.
23. Mozilla Foundation. Content Security Policy Specification, 2009.
24. Yacin Nadji, Prateek Saxena, and Dawn Song. Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense. In *Network & Distributed System Security Symposium (NDSS '09)*, 2009.
25. Eduardo Vela Nava and David Lindsay. Our favorite XSS filters/IDS and how to attack them. Presentation at the BlackHat US conference, 2009.
26. Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. Automatically hardening web applications using precise tainting. In *the 20th IFIP International Information Security Conference*, May 2005.
27. Nick Nikiforakis, Yves Younan, and Wouter Joosen. HProxy: Client-side detection of SSL stripping attacks. In *Proceedings of the 7th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA'10)*, 2010.
28. OWASP Top 10 Web Application Security Risks.
29. Tadeusz Pietraszek and Chris Vanden Berghe. Defending against Injection Attacks through Context-Sensitive String Evaluation. In *Recent Advances in Intrusion Detection (RAID2005)*, 2005.
30. William Robertson and Giovanni Vigna. Static Enforcement of Web Application Integrity Through Strong Typing. In *Proceedings of the USENIX Security Symposium*, Montreal, Canada, August 2009.
31. David Ross. IE 8 XSS Filter Architecture/Implementation, August 2008.
32. Alejandro Russo, Andrei Sabelfeld, and Andrey Chudnov. Tracking Information Flow in Dynamic Tree Structures. In *14th European Symposium on Research in Computer Security (ESORICS'09)*, 2009.
33. Philippe De Ryck, Lieven Desmet, Thomas Heyman, Frank Piessens, and Wouter Joosen. CsFire: Transparent Client-Side Mitigation of Malicious Cross-Domain Requests. In *Proceedings of 2nd International Symposium on Engineering Secure Software and Systems (ESSoS '10)*, 2010.
34. WhiteHat Security. XSS Worms: The impending threat and the best defense.
35. Alexa: The Web information company.
36. Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Proceedings of the 14th Annual Network and Distributed System Security Symposium (NDSS '07)*, 2007.
37. Gary Wassermann and Zhendong Su. Static Detection of Cross-Site Scripting Vulnerabilities. In *Proceedings of the 30th International Conference on Software Engineering*, Leipzig, Germany, May 2008. ACM Press New York, NY, USA.
38. Performance Benchmark - Monitor Page Load Time — Webmetrics.
39. Yichen Xie and Alex Aiken. Static Detection of Security Vulnerabilities in Scripting Languages. In *15th USENIX Security Symposium*, 2006.
40. XSSed — Cross Site Scripting (XSS) attacks information and archive.
41. Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. In *15th USENIX Security Symposium*, August 2006.
42. Yuchen Zhou and David Evans. Why Aren't HTTP-only Cookies More Widely Deployed? In *Proceedings of 4th Web 2.0 Security and Privacy Workshop (W2SP '10)*, 2010.