Cookiejar Kintsugi: Reviewing the state of web application session security

Julian Lobbes
j.lobbes@tu-braunschweig.de
Technische Universität Braunschweig
Braunschweig, Niedersachsen, Germany

Abstract

A large number of web applications store sensitive information about their users. Access to this information is often managed in the form of web sessions, which are attractive targets for malicious actors. This paper is a review of existing literature, outlining the methods used to handle the shared secrets pertaining to web sessions, common attacks used to discover these secrets, and defenses used to protect them. We also review existing empirical studies which have attempted to uncover the prevalence of web session vulnerabilities in the wild, showing that existing defensive mechanisms are often misused or underutilized.

CCS Concepts: • **Security and privacy → Web application security**.

Keywords: Session security, Cookie hijacking, Empirical analysis, Review

1 Introduction

Web applications have existed as a large part of the internet ecosystem ever since it has become accessible to the wider public in 1993 [11]. Transfer of information between websites and their users occurs using the Hyper Text Transfer Protocol (HTTP).

The protocol was designed for *stateless* communication, but most web applications need to store state information in between requests, for instance to track whether a user is already authenticated [4]. Thus, web applications require a mechanism for sharing and storing state information with clients in order to track the user's identity across HTTP requests.

Different methods exist to implement such a tracking mechanism on top of HTTP. Most commonly, session identifiers (SIDs) are used [17] in the form of a cookie [5]. Since SIDs are often used to authenticate a client accessing a web application and its protected resources, the SID must remain a shared secret between client and server, inaccessible to third parties.

Ensuring the confidentiality of session identifiers has proven to be difficult since the inception of web applications [10]. Many attack vectors have been identified in the past [13], and security features and methods were added to existing protocols retroactively [9] in order to patch these cracks. We outline the background of the underlying technology used to facilitate session management in web applications, and explain the most common attacks against sessions and the defenses against these attacks.

Finally, we review some previous studies which examine the prevalence of session management vulnerabilities in the wild, with a special focus on a fully automated session cookie vulnerability scanning framework by Drakonakis et al. [6]. Their work introduces a novel method which makes large scale analyses of session security in web applications possible, confirming previous findings [18] [19] [20] [17] [22] which indicate the prevalence of these vulnerabilities in the wild.

2 Background

For context, we provide a brief overview of the underlying technologies and mechanisms pertaining to web applications, session management and the internet in general.

2.1 Hyper Text Transfer Protocol (HTTP)

Content on the web is transferred using the text-based Hyper Text Transfer Protocol (HTTP), which adheres to the client-server paradigm. The user-side client initiates communications by sending an *HTTP request* to a web server, most commonly requesting a specific resource (often an HTML document) from it. The server responds with an *HTTP response*, indicating the completion status of the request and, if applicable, the requested resource. Both requests and responses send metadata along with the transmitted message. The attached metadata is sent inside the HTTP message *header*, and the contents of the message are sent inside the so-called message *body*.

Requests can take different forms [15], whereby the most common types of HTTP requests are `GET` requests and `POST` requests. It is important to note that `GET` requests do not contain a message body, as they are intended to retrieve resources from the server. Clients usually send data to a server by issuing `POST` requests.

Historically, HTTP was intended as a simple way to exchange documents formatted in the *Hyper Text Markup Language* (HTML) [8]. As such, it is stateless protocol, which treats each request as independent from preceding requests [15].

HTTP messages are sent in clear text. In order to ensure the integrity and confidentiality of requests and responses, as well as authenticating the server, HTTP was formally

extended to include a protocol specification for exchanging messages which are encrypted using the Transport Layer Security protocol (TLS) in 2000 [7]. This extension to HTTP is called Hypertext Transfer Protocol Secure (HTTPS). To ensure that all requests and responses exchanged between a client and server always use encrypted HTTPS connections, the HTTP Strict Transport Security (HSTS) standard was formalized in 2012 [9]. Modern web browsers ship with built-in lists of HSTS-enabled domains. These browsers will never connect to an HSTS-enabled domain over plain HTTP, even on the first visit to that domain or if the user manually specifies `http://` in the browser's URL bar.

2.2 Web sessions

As the adoption of HTTP rapidly grew shortly after its formal inception in 1996 [15], many revisions of the protocol standard, largely centered around performance increases, were published and adopted in quick succession. Application developers soon desired ways to build applications which preserve information about their users between requests, despite the stateless nature of HTTP. Early web browsers did not support storing state information [14], but the need to keep track of such data quickly became apparent, especially in the context of web applications [5].

Uniquely identifying a user across individual requests is represented by the idea of a *session*. A session can be established by the server generating a *session identifier* (SID) and sending it to the client. The client now re-transmits the SID with each request, for the duration of the session, thus authenticating themselves as a session owner whom the application can identify.

Several mechanisms exist and have been widely used to persist SIDs across requests [10]. Two methods which were widely employed in the past to exchange and store SIDs are *hidden form fields* and *URL rewriting*. Due to poor reliability, limitations in user experience and security issues [23] [10], they have been widely replaced by *cookies*.

2.2.1 Hidden form fields. An SID can be embedded in a hidden HTML form by the server, and the document containing the form is then sent to the client. The client must now submit the form containing the SID to the server with each subsequent request in order to keep the session alive. The SID is embedded in the HTML source code of each page the user receives for the duration of the session, albeit hidden from the user's direct view by the browser.

A downside to this approach is that the SID, or other state information, may get lost if the user clicks on their browser's back-button. Additionally, the performance overhead of parsing a form for every request on the server side is significant, and hidden form fields do not lend themselves well to client-side caching of web pages [14].

Since the SID is plainly visible to anyone with access to the HTML source code on the client's machine, sessions relying on hidden form fields are particularly vulnerable against cross-site-scripting attacks (XSS), as shown in section 3.2.

2.2.2 URL rewriting. In URL-rewriting, the server generates an SID and redirects the client to a URL containing the SID as a URL parameter. The SID gets appended to the URL for each subsequent request the client sends to the server. A URL containing a session ID may look like listing 1:

```
https://example.com/profile.html?sid=a92nl52
```

Listing 1. Session ID key-value pair embedded in a URL

The SID is visible in the client's address bar and browser history.

Just like with hidden form fields, state information is lost if the user presses their browser's back-button. Web applications utilising URL rewriting suffer from poor server-side caching performance [14]. The biggest downside with this approach, however, is that the SID can quite easily leak to third parties. Users may copy a link containing an SID from their address bar, and inadvertently share their SID with others. The browser may also leak the SID to other web servers if a logged-in user follows a link leading to another domain, because the link origin's URL is displayed in the `Referrer` HTTP header field in the request leading to the third-party domain. Moreover, applications which manage sessions using URL rewriting are particularly vulnerable to *session fixation* vulnerabilities, as shown in section 3.5.

2.2.3 Cookie-based session management. The HTTP specification was extended to include *cookies* in 1997 [14]. Cookies are name/value pairs, transmitted using HTTP message headers, and stored in the client's browser and on the web server. As shown in Figure 1, a cookie is set by the web server, using the HTTP header's `Set-Cookie` directive, and then sent to the client. Clients send all cookies they are storing in the context of a particular domain along with each request they send to this domain's web server, thus preserving state information across multiple requests [2]. If the cookie contains an SID, the server can identify the session and authenticate the user for each request.

Besides being name/value-pairs, cookies can be individually configured to have some additional attributes telling both clients and servers to handle them in a certain way. Two important attributes for cookies are the `Secure`-attribute and the `HttpOnly`-attribute. `Secure` prevents cookies from being sent over unencrypted channels such as plain text HTTP. `HttpOnly` disallows client-side JavaScript code from reading the cookie. Correct usage of each of these attributes for authentication cookies is crucial to securing web sessions, as we show in sections 3.1 and 3.2 respectively.

3 Session security threats

A vast number of web applications on the internet allow access to sensitive information. With the number of Internet

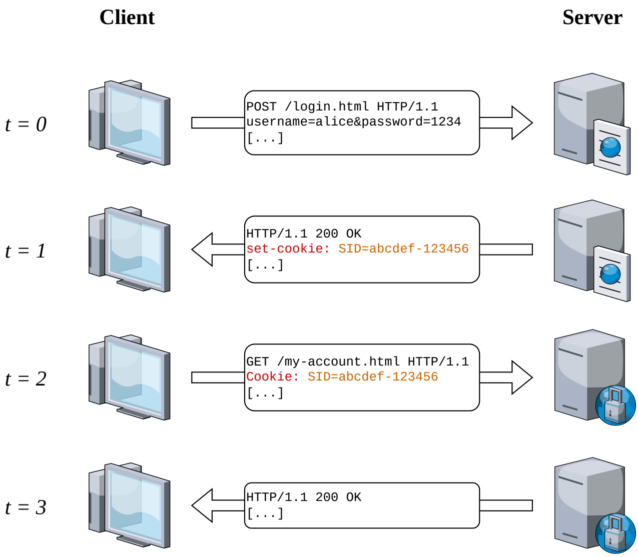

Figure 1. HTTP cookie exchange

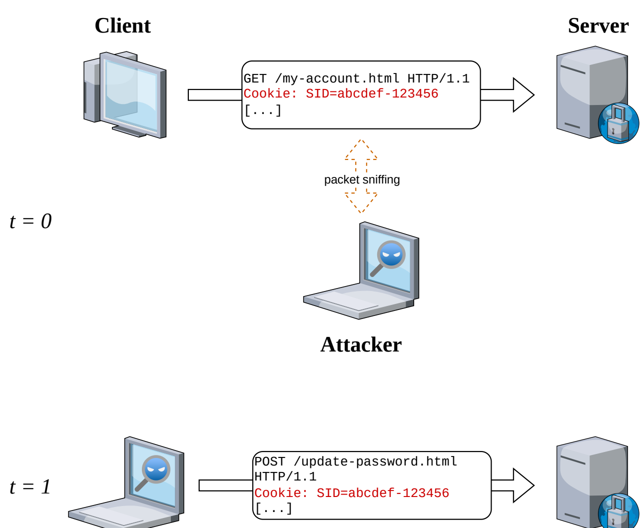

Figure 2. Man in the middle packet capture

of Things (IoT) devices sharply on the rise, the number of endpoints providing access to not just sensitive information, but to control systems as well, is growing rapidly. Many of these devices are controlled via web interfaces [21].

Many modern web applications utilise an authentication system, restricting access to such resources to authorised users, whereby session management and authentication using cookie-based SIDs is the de-facto standard [5]. SIDs are a highly desirable target for malicious actors, particularly in light of how severe the consequences of a breach in confidentiality of the SID can be. *Session hijacking* is a class of attack in which an attacker obtains an innocent user's session identifier. The attacker can then use the session identifier to authenticate as the victim, granting them unauthorised access to protected resources. Session hijacking can be carried out at the network layer, as well as at the application layer [10]. In this section, an overview of the common ways in which session tokens can be stolen is examined.

3.1 Plain text packet capture

In this network level attack, the attacker monitors TCP traffic on their local network, or on a route between the victim and the destination web server. If the vulnerable application transmits SIDs in plain text HTTP, an attacker acting as a man in the middle can read the SID. This is shown in Figure 2.

When web traffic is unencrypted, web applications utilizing URL rewriting or hidden forms are susceptible to plain text packet capture, as attackers can read the full HTTP request, including message header and body. If cookies are used to transmit the SID, the session is only protected against this kind of attack if the `secure` option is set on the cookie, as this flag prevents its transmission over plain text protocols.

3.2 Cross-site scripting (XSS)

Scripts running within a website's first origin can access all HTML elements within the document in which they are embedded. This makes all documents containing hidden form fields susceptible to access by malicious scripts. First origin scripts may also read and set cookies unless the cookie's `HttpOnly`-flag is set. A web application vulnerable to XSS which does not set `HttpOnly` for its session cookies allows an attacker to inject a script which extracts the SID and sends it to the attacker.

3.3 Unisolated scripts

The majority of websites import third party JavaScript scripts from remote sources [10]. These scripts run in the website's first origin if they are not isolated within an HTML `<iframe>` block, which effectively allows the provider of the remote script access to all session cookies, if they are not set to `HttpOnly` [16]. Malicious or compromised script providers can abuse this to access unwitting users' sessions.

3.4 Cross-site request forgery (CSRF)

According to Zeller and Felten [24], "CSRF attacks occur when a malicious web site causes a user's web browser to perform an unwanted action on a trusted site". Rather than stealing the session ID, the attacker's goal is to force the victim to unknowingly execute an action chosen by the attacker on the target website.

An authenticated user's browser will send the session cookie along with every request to the target website, re-authenticating them with every request. An attacker can trick the user into submitting a specially crafted request to the target website, for example by embedding the request as a hidden HTML element in an email or on a malicious

website. As an example, an attacker may craft an invisible image such as the one shown in listing 2:

```
<img src="https://bank.com/transfer.php?acct=
    ATTACKER&amount=100000" width="0" height="0"
    border="0">
```

Listing 2. HTML image containing a CSRF exploit

Simply loading a page containing the above HTML image tag would cause the user's browser to send a request for the URL specified as the image source to `bank.com`. If the user is currently logged in, their session cookie for the site will be appended, authorizing the transaction without their knowledge.

In contrast to web applications which utilize cookies for session management, those employing URL rewriting or hidden forms are typically not vulnerable to CSRF.

3.5 Session fixation

In a session fixation attack, the attacker first establishes a session on the server, retrieving an SID in the process. They then introduce the retrieved session token into the victim's browser, for example by employing social engineering. Both attacker and victim will now be logged in to the same session. If the session ID is stored in the URL, session fixation attacks are much easier for an attacker, as they only need to get their victim to click on a link containing their SID in order to trick them into using the attacker's crafted session. This process is depicted in Figure 3.

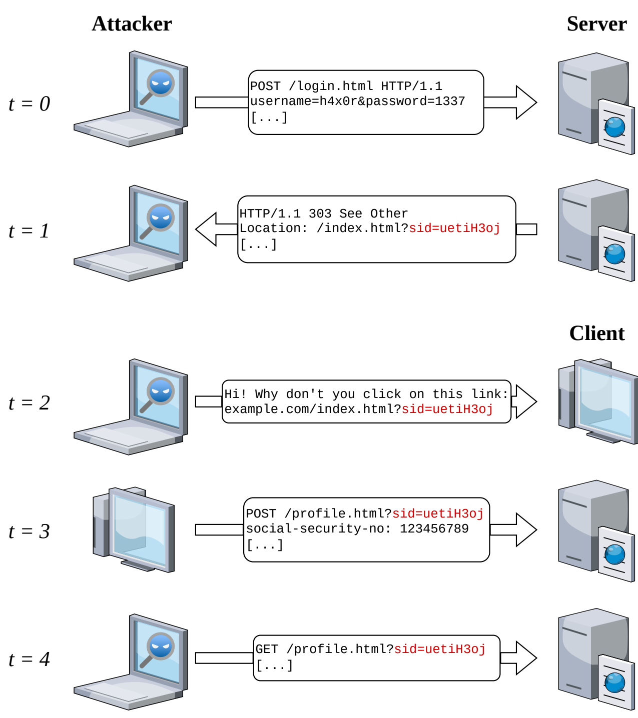

Figure 3. Session fixation attack

3.6 Cache/Log sniffing

The browser cache contains session cookies and cached HTML documents containing SIDs embedded in hidden forms. The browser's history keeps track of all URLs a user has visited, including those containing SIDs for websites which use URL rewriting. An attacker with access to a victim's browser cache can compromise their sessions on websites which use hidden forms or cookies. Access to the browsing history will reveal SIDs from websites which use URL rewriting.

A log sniffing attack does not necessarily require direct access to a victim's browser by the attacker. If a user authenticates with a web application which employs URL rewriting to manage the session, any machine relaying packets or HTTP requests for the victim may keep logs which contain the victim's SID as a URL parameter if the traffic is unencrypted. A malicious actor with access to the logs can thereby intercept the victim's SID and use it to hijack the session.

4 Threat mitigation

As presented previously, there are various attack vectors which malicious actors can exploit, almost all of which can, and should, be mitigated on the application developer's and system administrator's side. This section serves as an overview for the most fundamental and important security mechanisms which web site providers should employ in order to prevent session hijacking vulnerabilities by today's standards.

4.1 Use cookies

The use of cookies for session management has become the industry standard [5], and with good reason. The previously widespread methods of using URL rewriting or hidden forms to handle SIDs have proven to be unreliable and fundamentally insecure in the past [23] [10] [19] [20]. Using cookies in their place greatly reduces the attack surface for a range of attacks which enable session takeover. The remainder of this paper focuses on session security of web applications that use cookies for session management.

4.2 Use HTTPS

Utilizing encrypted communication to transfer session cookies is a prerequisite to preventing cookie hijacking. Restricting the transmission of session cookies to HTTPS is arguably the easiest way to prevent a wide range of man-in-the-middle vulnerabilities. Yet, a surprising number of websites, including some major ones, fail to do so [17] [22] [6]. The most straightforward way to ensure that session cookies are never transmitted in the clear is by setting their `Secure` flag.

Other mechanisms which ensure HTTPS, such as upgrading a visitor's connection from HTTP to HTTPS, provide some protection. However, these fall short if the session cookie is sent in plain text during the initial request, for

example when the user manually visits a URL prefixed with `http://` [3]. Web server administrators and domain owners should enable HSTS for their servers and domain to ensure that only HTTPS is used across all of the domain and all subdomains [9]. So far, however, HSTS has not seen wide adoption, and misconfigurations are common [6], as is exhibited in section 5.2.3.

4.3 Disable script access to session cookies

Developers should ensure that session cookies cannot be accessed from client-side scripts. This can be ensured by adding the `HttpOnly` flag to session cookies, mitigating the risks of XSS vulnerabilities, at least in the context of session hijacking.

4.4 Isolate 3rd party scripts

Scripts imported from remote content providers run in the importing website's first origin by default. If session cookies are accessible to scripts, this gives third parties access to user sessions, and opens the web application up to XSS attacks from compromised or untrustworthy content providers. Application developers should isolate external scripts [6] to prevent this.

5 Prevalence

Gathering data for large-scale empirical assessments about the prevalence of web session vulnerabilities in the wild is not straightforward. Analyzing whether a large number of web applications handle their respective session tokens securely requires creating a user account on each application to be audited, followed by signing in using the created account and subsequently analysing the website's behaviour with regards to the session cookies. As automated account creation and sign-in on web applications is generally considered undesirable, in some cases even a vulnerability, this process presents the largest hurdle to conducting large scale studies [6].

This is reflected by the lack of large scale empirical data available today. Several studies and data sources have attempted to determine the prevalence of session management vulnerabilities on the web in general, but all previously published studies require significant levels of manual effort by researchers for account creation and sign-in, which reduces the pool of auditable web applications significantly, with the exception of Drakonakis et al. [6].

5.1 Manual analysis

Most data sources determining the incidence of session vulnerabilities rely on manual interaction for account creation and sign-in [6]. The Open Web Application Security Project (OWASP) sources from a large pool of contributors manually reporting occurrences of web vulnerabilities, and is able to periodically provide a comprehensive report on the state of web application security in general [1], outlining the most significant vulnerabilities at the current point in time. Over the past decade, XSS injection was ranked among the top three most significant risks in OWASP's top 10. The significance rating of web site misconfiguration, which includes exposed authentication cookies, has been gradually increasing [18] [19] [20], and remains high today. The presence of each of these vulnerability types can leave web applications open to session hijacking attacks.

This trend seems to be reflected in studies such as Nikiforakis et al. [17], who found that less than 23% of websites which use session cookies set the *HttpOnly*-flag on their cookies. Research by Sivakorn et al. [22], which utilized large scale network packet capturing, showed that 15 major websites, including Google, exposed session cookies via unencrypted connections, with most of the web sites inspected being vulnerable to XSS cookie stealing as well.

Relying on manual interaction to audit a small selection of the vast number of web applications present on the web today, however, only permits us to catch a small glimpse of the overall state of web session security.

5.2 Automated analysis

Drakonakis et al. [6] developed a "fully automated black-box auditing framework that analyzes web apps by exploring their susceptibility to various cookie-hijacking attacks". The framework's goal was to audit web applications "without knowledge of their structure, access to the source code, or input from developers" on a large scale.

The study's methodology solved the hurdle presented by automated account creation and sign-in, at least for those websites which do not employ captcha challenges, and offers a good reference for the steps necessary to conduct an automated black-box security audit of web applications on the public web.

5.2.1 Methodology.

Drakonakis et al. [6]'s framework is able to crawl websites from a large dataset of URLs, and initially attempts to locate sign-up and login forms. The semantic purpose of crawled pages, such as whether or not a page is a user registration form, is inferred from the number, types and labels of any discovered HTML `<input>` tags.

If any login and sign-up pages were located, the discovered forms and input fields are labelled internally by the framework, to enable filling them with data. Reliable identification of a specific input field's purpose is necessary to pass sign-up form validation and keep track of the login credentials used to register. This was achieved by searching HTML element attributes for specific keywords which infer the element's purpose.

Once the sign-up form is filled with data and submitted, a registration status oracle determines whether the sign-up

process was successful. This includes scraping received sign-up validation emails and pages to which the framework was redirected following the sign-up form submission.

If the registration process was deemed to be successful by the oracle, a login module attempts to log in automatically. Success of the operation is determined by a login oracle, again using the presence of certain HTML elements, such as a log out button, to determine whether the operation was successful.

If the registration was unsuccessful but the website supports Google or Facebook single sign-on (SSO), the framework attempts this method for sign in, using a previously created default account for the respective service. Websites whose registration process involves solving a captcha were not audited automatically.

5.2.2 Audit. The vulnerability assessment of audited sites centers around the framework's cookie auditor. The auditor's goal is to find authentication cookies which are not sufficiently protected against session hijacking. Session cookies without an `HttpOnly` flag are assumed to be vulnerable to XSS sniffing if the website also imports unisolated scripts from third parties. Cookies missing the `Secure` flag are assumed to be vulnerable to network-based hijacking, if the site employs HSTS either incorrectly or not at all. Whether or not HSTS usage is correct for an audited website is determined by a separate module also implemented by the framework.

Determining which cookies are session cookies is crucial in this context. In a series of requests to the website, differing sets of cookies are omitted from the request each time, and the login oracle is consulted to determine whether the request led to the user being logged in. Once a set of authentication cookies has been identified, conclusions about their hijacking susceptibility can be drawn based on which flags they have.

The framework also utilizes a privacy auditor, with the goal of determining what kind of user data can be retrieved from the web site by an attacker after successfully capturing a session cookie. The privacy auditing module is capable of determining the nature of any sensitive information revealed once an attacker has successfully stolen a session on the vulnerable website, such as email or postal addresses, phone numbers and similar sensitive user information.

5.2.3 Findings. Drakonakis et al. [6]'s study inspected 1.5 million unique domains, 200,000 of which were detected to be web applications supporting account creation. 25,000 web applications were fully audited, whereby automatic account creation presented itself as the most significant obstacle.

12,014 domains (48.43%) were found to be vulnerable to eavesdropping because authentication cookies are transmitted over unencrypted connections. It is worth noting that out of these, 10,495 (87.36% of those vulnerable to eavesdropping) did not deploy HSTS. Websites which do not set session cookies as `Secure` are protected from eavesdropping if HSTS is deployed correctly on the web server. Drakonakis et al. [6] shows that even on sites where HSTS was deployed *incorrectly*, covering only certain subdomains for example, cookies without the `Secure` attribute were safe from eavesdropping in many cases, yet the vast majority of sites do not use HSTS. Four of the audited websites were themselves SSO identity providers providing authentication for many other sites, while transmitting session cookies in clear text.

5,680 domains did not set the `HttpOnly` flag on their cookies. The vast majority of these sites (5,099) also import remote JavaScript remotely without isolation, allowing the imported script to read authentication cookies. While these sites are not necessarily vulnerable to XSS, they do allow third parties to read their session cookies.

The findings indicate that the threat to a victim's privacy on those sites which are vulnerable to session hijacking is significant, allowing attackers to retrieve full names, email addresses, phone numbers and a plethora of other highly sensitive information.

6 Conclusion

Session hijacking attacks have existed since the dawn of the internet as we know it today. Web sessions make for an attractive target for actors with malicious intent, as they allow access to sensitive information about users. Hijacked sessions may even provide access to control systems or administrative interfaces, depending on the breached application and level of access obtained. The number of web applications in our daily lives is ever increasing. As their significance grows, so does the impact of how securely they are managed.

Drakonakis et al. succeeded in conducting the first large-scale, fully automated session security audit. They explored and implemented a novel method which overcomes many of the technical limitations of previous empirical studies, while confirming the findings of their predecessors. Their method allowed for a much larger sample size and hence much greater certainty in previously proposed insights: a large portion of publicly accessible web applications on the internet vulnerable are to session hijacking, even today. The majority of discovered vulnerabilities are preventable, yet the mechanisms required to starkly reduce these risks are not being implemented widely enough.

While occurrences of web session vulnerabilities remain frequent and affect even large and well-funded web sites, the security auditing framework presented by Drakonakis et al. sets a new precedent in the available methodologies at the disposal of researchers looking to conduct large scale empirical analyses related to web application sessions.

References

[1] [n.d.]. OWASP Top Ten Web Application Security Risks | OWASP. https://owasp.org/www-project-top-ten/

[2] Adam Barth. 2011. *HTTP State Management Mechanism.* Request for Comments RFC 6265. Internet Engineering Task Force. https:

//doi.org/10.17487/RFC6265 Num Pages: 37.

[3] Michele Bugliesi, Stefano Calzavara, Riccardo Focardi, and Wilayat Khan. 2015. CookiExt: Patching the browser against session hijacking attacks. *Journal of Computer Security* 23, 4 (Sept. 2015), 509–537. https://doi.org/10.3233/JCS-150529

[4] Stefano Calzavara, Riccardo Focardi, Marco Squarcina, and Mauro Tempesta. 2018. Surviving the Web: A Journey into Web Session Security. *Comput. Surveys* 50, 1 (Jan. 2018), 1–34. https://doi.org/10.1145/3038923

[5] DacostaItalo, ChakradeoSaurabh, AhamadMustaque, and Traynor-Patrick. 2012. One-time cookies. *ACM Transactions on Internet Technology (TOIT)* 12, 1 (July 2012), 24. https://doi.org/10.1145/2220352.2220353 Publisher: ACM PUB27 New York, NY, USA.

[6] Kostas Drakonakis, Sotiris Ioannidis, and Jason Polakis. 2020. The Cookie Hunter: Automated Black-box Auditing for Web Authentication and Authorization Flaws. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. ACM, Virtual Event USA, 1953–1970. https://doi.org/10.1145/3372297.3417869

[7] Roy T. Fielding and Julian Reschke. 2014. *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content.* Request for Comments RFC 7231. Internet Engineering Task Force. https://doi.org/10.17487/RFC7231 Num Pages: 101.

[8] Ilya Grigorik. 2013. *High-performance browser networking.* O'Reilly, Beijing ; Sebastopol, CA. OCLC: ocn827951729.

[9] Jeff Hodges, Collin Jackson, and Adam Barth. 2012. *HTTP Strict Transport Security (HSTS).* Request for Comments RFC 6797. Internet Engineering Task Force. https://doi.org/10.17487/RFC6797 Num Pages: 46.

[10] Vineeta Jain, Divya Rishi Sahu, and Deepak Singh Tomar. 2015. Session Hijacking: Threat Analysis and Countermeasures. In *International Conference on Futuristic Trends in Computational analysis and Knowledge management*, Vol. 1. amity University, Greater Noida, 7.

[11] Mehdi Jazayeri. 2007. Some Trends in Web Application Development. In *Future of Software Engineering (FOSE '07)*, Vol. 1. IEEE Computer Society, 199–213. https://doi.org/10.1109/FOSE.2007.26

[12] Guy Keulemans. 2016. The Geo-cultural Conditions of Kintsugi. *The Journal of Modern Craft* 9, 1 (Jan. 2016), 15–34. https://doi.org/10.1080/17496772.2016.1183946 Publisher: Routledge _eprint: https://doi.org/10.1080/17496772.2016.1183946.

[13] Mitja Kolsek. 2002. Session fixation vulnerability in web-based applications. *ACROS Security* (Dec. 2002).

[14] David M. Kristol. 2001. HTTP Cookies: Standards, Privacy, and Politics. (2001). https://doi.org/10.48550/ARXIV.CS/0105018 Publisher: arXiv Version Number: 1.

[15] Henrik Nielsen, Roy T. Fielding, and Tim Berners-Lee. 1996. *Hypertext Transfer Protocol – HTTP/1.0.* Request for Comments RFC 1945. Internet Engineering Task Force. https://doi.org/10.17487/RFC1945 Num Pages: 60.

[16] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. 2012. You are what you include: large-scale evaluation of remote javascript inclusions. In *Proceedings of the 2012 ACM conference on Computer and communications security (CCS '12)*. Association for Computing Machinery, New York, NY, USA, 736–747. https://doi.org/10.1145/2382196.2382274

[17] Nick Nikiforakis, Wannes Meert, Yves Younan, Martin Johns, and Wouter Joosen. 2011. SessionShield: Lightweight Protection against Session Hijacking. In *Engineering Secure Software and Systems (Lecture Notes in Computer Science)*, Úlfar Erlingsson, Roel Wieringa, and Nicola Zannone (Eds.). Springer, Berlin, Heidelberg, 87–100. https://doi.org/10.1007/978-3-642-19125-1_7

[18] The Open Web Application Security Project. 2010. *OWASP Top 10 - The Ten Most Critical Web Application Security Risks.* Technical Report.

[19] The Open Web Application Security Project. 2017. *OWASP Top 10 Application Security Risks - 2017.* Technical Report.

[20] The Open Web Application Security Project. 2021. *OWASP Top 10 - 2021.* Technical Report.

[21] Neha Sharma, Madhavi Shamkuwar, and Inderjit Singh. 2019. The History, Present and Future with IoT. In *Internet of Things and Big Data Analytics for Smart Generation*, Valentina E. Balas, Vijender Kumar Solanki, Raghvendra Kumar, and Manju Khari (Eds.). Springer International Publishing, Cham, 27–51. https://doi.org/10.1007/978-3-030-04203-5_3

[22] Suphannee Sivakorn, Iasonas Polakis, and Angelos D. Keromytis. 2016. The Cracked Cookie Jar: HTTP Cookie Hijacking and the Exposure of Private Information. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Jose, CA, 724–742. https://doi.org/10.1109/SP.2016.49

[23] Shellie Wedman, Annette Tetmeyer, and Hossein Saiedian. 2013. An Analytical Study of Web Application Session Management Mechanisms and HTTP Session Hijacking Attacks. *Information Security Journal: A Global Perspective* 22, 2 (March 2013), 55–67. https://doi.org/10.1080/19393555.2013.783952 Publisher: Taylor & Francis _eprint: https://doi.org/10.1080/19393555.2013.783952.

[24] William Zeller and Edward W Felten. [n.d.]. Cross-Site Request Forgeries: Exploitation and Prevention. ([n. d.]), 13.

A Glossary

A.1 Kintsugi

Kintsugi is a traditional Japanese craft in which ceramics, either accidentally or purposefully broken, are repaired with urushi lacquer and gold [12].

A.2 SID - session identifier

A session identifier 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 [17].