Session Fixation – the Forgotten Vulnerability?

Michael Schrank[1] Bastian Braun[2] Martin Johns[3] Joachim Posegga[2]

[1] University of Passau, schrank@fim.uni-passau.de
[2] ISL Passau, {bb, jp}@sec.uni-passau.de
[3] SAP Research, martin.johns@sap.com

Abstract:
The term 'Session Fixation vulnerability' subsumes issues in Web applications that under certain circumstances enable the adversary to perform a session hijacking attack through controlling the victim's session identifier value. We explore this vulnerability pattern. First, we give an analysis of the root causes and document existing attack vectors. Then we take steps to assess the current attack surface of Session Fixation. Finally, we present a transparent server-side method for mitigating vulnerabilities.

1 Introduction and paper outline

Session Fixation has been known for several years. However, compared to vulnerability classes such as Cross-site Scripting (XSS), SQL Injection, or Cross-site Request Forgery (CSRF), this vulnerability class has received rather little attention. In this paper, we delve into Session Fixation in three ways: First we explore the vulnerability pattern and document existing attack methods (see Sec. 2). Secondly, we take steps to assess the current attack surface of Session Fixation (Sec. 3). Finally, we present a transparent server-side method for mitigating vulnerabilities (Sec. 4). We finish the paper with a look on related work (Sec. 5) and a conclusion (Sec. 6).

2 Exploring Session Fixation

2.1 Technical background: Session management

HTTP is a stateless protocol. Thus, HTTP has no protocol-level session concept. However, the introduction of dynamic Web applications resulted in workflows which consist in a series of consecutive HTTP requests. Hence, the need for chaining HTTP requests from the same user into usage sessions arose. For this purpose, session identifiers (SID) were introduced. A SID is an alphanumerical value which is unique for the corresponding Web session. It is the Web application's duty to implement measures that include the SID in every HTTP request that belongs to the same Web session. A SID mechanism can be implemented by transporting the value either in form of an HTTP cookie or via HTTP

parameters [FGM+99]. All modern Web application frameworks or application servers, such as PHP or J2EE, implement application transparent SID management out of the box.

Any request that contains the SID value is automatically regarded to belong to the same session and, thus, to the same user. After a user has authenticated himself against the application (e.g., by providing a valid password), his authorization state is also directly tied to his SID. Hence, the SID de facto becomes the user's authorization credential.

2.2 Session Fixation

Several classes of attacks that target SID-based authorization tracking have been documented in the past. Especially Session Hijacking via XSS [End02] has received considerable attention. This attack is based on the adversary's ability to obtain a user's valid and authenticated SID through the injection of JavaScript code into the application. Using this SID, the adversary is able to impersonate the victim in the context of the attacked application. Besides XSS other techniques to obtain valid SIDs exist, such as Sidejacking [Gra07] or brute-force session guessing [Kam09].

Session Fixation [Kol02] is a variant of Session Hijacking. The main difference to the variants discussed above is that Session Fixation does not rely on SID theft. Instead the adversary tricks the victim to send a SID that is controlled by the adversary to the server. See Figure 1 for a brief example:

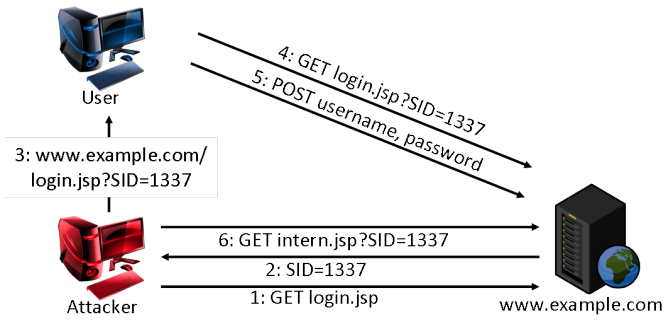

Figure 1: Exemplified Session Fixation attack [Kol02]

- The attacker obtains a SID value from the server (1,2).
- He tricks the victim to issue an HTTP request using this SID during the authentication process (3,4).
- The server receives a request that already contains a SID. Consequently, it uses this SID value for all further interaction with the user and along with the user's authorization state (5).

- Now, the attacker can use the SID to access otherwise restricted resources utilizing the victim's authorization context (6).

2.3 Why does Session Fixation exist?

At first glance, the Session Fixation attack pattern seems both contrived and unlikely. Why would an application accept a user's SID that was not assigned by the application to this user in the first place? The root problem lies within a mismatch of responsibilities: SID management is executed by the utilized programming framework or the application server. All related tasks, such as SID generation, verification, communication, or mapping between SID values to session storage, are all handled transparently to the application. The programmer does not need to code any of these tasks manually. However, the act of authenticating a user and subsequently adding authorization information to his session data is an integral component of the application's logic. Consequently, it is the programmers duty to implement the corresponding processes. The framework has no knowledge about the utilized authentication scheme or the employed authorization roles.

Finally, in general, modern Web applications assign SID values automatically with the very first HTTP response that is sent to a user. This is done to track application usage even before any authentication processes have been executed, e.g., while the user accesses the public part of the application. Often the programmer is not aware of this underlying SID functionality as he is not responsible for managing the SIDs. He simply relies on the framework provided automatism. The combination of the circumstances listed above lead to situations in which applications neglect to reissue SID values after a user's authorization state has changed. This in turn causes Session Fixation vulnerabilities.

2.4 Exploiting Session Fixation

Given the sources of Session Fixation vulnerabilities as described in Sec. 2.3, a broad field of attack vectors comes into play. All attacks strive to 'implant' a known SID to the victim's browser. As described in Sec. 2.1, SIDs are communicated either via HTTP parameters or cookies:

URL parameter: The easiest way to push a SID to the user's browser is a URL parameter (see Figure 1) in situations in which the attacked application accepts such SIDs. The attacker generates a valid SID, e.g. `1337`, of the vulnerable application on address `http://www.example.com`. Then, he sends the URL `http://www.example.com/?SID=1337` to the victim. For example, he could promise that the pictures from his last summer holiday are published there. The victim's browser sends a request to the application making use of the given SID if he clicks on this link. From the application's point of view, he already has a valid session. So, it does not need to deliver a new SID.

As a next step, the victim provides his credentials and logs in to the application. The session has not changed but he is now logged in. The application still identifies the victim

by his SID that is sent with every request. However, the attacker also knows this SID and can thus act as the victim in his account. Therefore, he does not even have to interfere in the communication between the victim and the application. The only thing to do is sending a request to the application like `http://www.example.com/account_balance.php?SID=1337`.

Cookies: A slightly more difficult way for the attacker are cookies set by the application. In this scenario, the adversary requests a session cookie from the application. Then, he needs to make his victim accept the same cookie. There are several options for him to reach his goal as we will show. When the victim owns the adversary's session cookie, his browser will provide this cookie to the application. As the adversary can use this cookie too, he can own his victim's session.

- **Setting cookies via XSS:** The adversary can use cross-site scripting (XSS) to set a cookie at his victim's site if the web application is vulnerable to this attack. Therefore, he inserts JavaScript code into a web page in the same domain, `.example.com`. When the victim visits this page, the cookie will be set. The adversary can set the expiry date to a date in distant future so that the cookie will not be deleted when the victim restarts his browser. Unlike a cookie stealing attack, the victim does not have to be logged in, thus, the attack also works at the public part of the application. In the past, we saw browser vulnerabilities that even allowed the attacker to set the cookie from a foreign domain [Zal06].

- **Setting cookies via `meta` tags:** Under certain circumstances, a Web application might allow user-provided HTML markup but filters user-provided JavaScript. In such cases, the attacker might be able to inject special `meta` tags. The tag `<meta http-equiv="Set-Cookie" Content="SID=1337;expires=Monday, 07-Mar-2011 12:00:00 GMT">` sets a cookie which is valid until March 2011.

- **Setting cookies via cross-protocol attacks:** The same-origin policy [Rud01] explicitly included the port of the URL that was utilized to retrieve a JavaScript as a mandatory component of its origin. In consequence, if a service in a given domain allows the adversary to execute JavaScript, e.g., via XSS, services in that domain which are hosted on different ports are unaffected by this.

  However HTTP cookies are shared across ports [KM00]. Thus, cross-protocol attacks come into play: Cross-protocol attacks [Top01, Alc07] (see below for further explanations) allow the adversary to create XSS like situations via exploiting non-HTTP servers, such as SMTP or FTP, that are hosted on the same domain.

  First, the adversary prepares a website with a specially crafted HTML form [Top01]. The form contains JavaScript code to set the attacker's cookie at his victim's browser. The target of this form is the targeted non-HTTP server[1]. To avoid URL encoding of the content, the adversary chooses the `enctype="multipart/form-data"` parameter. The target server interprets the HTTP request as a valid request of its own protocol, e.g. FTP [Dab09]. However, most of the incoming commands cannot

[1] NB: Some browser, e.g., Firefox, block HTTP requests to certain well known ports. In such cases the attack requires a susceptible server on a non-blocked port to function.

be understood and will be reflected with a respective error message. This message generally contains the erroneous input which is in our case the JavaScript code with the cookie. For compatibility reasons, browsers take all textual input as HTTP even without any valid HTTP header. So, the browser accepts the cookie since it comes from a server in the same domain.

- **Subdomain Cookie Bakery:** JavaScript's same origin policy prohibits the setting of cookies for other domains. Nevertheless this only applies to top level domains. Therefore a possible attack vector is setting a cookie using a vulnerable subdomain, e.g. JavaScript code or a `meta` tag on `vulnerable.example.com` can set a cookie which is subsequently sent by the victims browser for `example.com`. Hence a vulnerable application on a subdomain may compromise the session security of the whole top level domain.

- **HTTP Response Splitting and HTTP Header Injection:** In case the web application is also vulnerable to HTTP response splitting, the attacker could use this to send his cookie to the victim's browser [Kle04]. Therefore, he needs a redirected page that includes unfiltered user provided content and a proxy with a web cache.

  First, the attacker sends a crafted HTTP request through the proxy to `http://www.example.com/redirect.php`. The application takes information from the attacker's request to generate the target of the redirection, e.g. `/app_by_lang.php?lang=`*attacker's parameter*. This way, he can influence the `Location` header field where the redirection URL is denoted. He can then shape a request that finishes the HTTP response and append a new response. The latter is now fully controlled by himself, i.e. he can use the `Set-Cookie` header and deliver his cookie, e.g. `/redirect.php?lang=en%0d%0a Content-Length:%200%0d%0a%0d%0aHTTP/1.1%20200%20OK%0d%0aContent-Type:%20text/html%0d%0aSet-Cookie:%20SID=1337%0d%0a Content-Length:%2019%0d%0a%0d%0a<html>Foobar</html>` [Kle04]. The proxy caches the second response if he manages to send a harmless request to `http://www.example.com/` at the right time. The actual answer to his second request will be discarded and considered superfluous. Finally, the proxy serves his specially crafted HTTP response to requests on `http://www.example.com/`.

Figure 2: A Header Injection Attack

The attack works similarly if the application takes user provided data to set a cookie. In this case, the `Set-Cookie` header serves as an entry point instead of the `Location` header. The attacker could run a HTTP header injection attack (see Figure 2) instead. Then, he prepares a special URL like `http://www.example.com/app_by_lang.php?lang=en%0d%0aSet-Cookie:%20SID=1337%0d%0a` and sends this to his victim. This attack is then analogical to the URL parameter attack. The victim has to click on the link and enter his login credentials. After that, the attacker can own his account. We tested the feasibility of header injection attacks and give results in Sec. 3.

2.5 Impact and discussion

Currently, Session Fixation is only a second stage attack that usually requires several preconditions.

The attacker needs another vulnerability to provide his cookie to the victim. Alternatively, he must mislead the victim into clicking on his link if the target web application allows URL parameters for session management. We presented different attack vectors in Sec. 2.4. In case the attacker is successful at the first step, he has to make the victim log into his account. The SID is useless as long as it does not belong to a logged in user. However, the attacker neither knows when the victim logs in nor when he logs out again. He has an unknown window of opportunity. Finally, the target web application has to be vulnerable as we pointed out in Sec. 2.2. Actually, it would be essential for a broad attack to provide a unique cookie for each victim. Otherwise, one victim would take the session of another. This is however not always easy to implement for the attacker depending on the attack vector.

When the conditions are met, though, Session Fixation is a severe attack that allows the attacker to fully impersonate the victim. It is generally not obvious to the victim to be under attack, especially if he is not familiar with Session Fixation. Most attack scenarios appear to be a software malfunction to the unexperienced user. The victim may even not notice the attack afterwards depending on the actions of the attacker on his behalf.

3 Practical experiments

In the context of this paper, we took first steps to assess to which degree Session Fixation currently poses a realistic threat. For this purpose, we conducted two series of practical experiments.

First we tested open source Content Management System (CMS) applications for Session Fixation issues. This way we aimed to get an estimate if developers are actually aware of the lingering threat of Session Fixation or if they unknowingly rely on the framework's protection mechanisms. The result of these tests suggested that a considerable fraction of existing applications indeed are susceptible to Session Fixation under circumstances that

allow the attacker to set cookies on the victim's browser (see Sec. 2.4). Consequently, we examined several popular web application frameworks with respect to their susceptibility to potential header injection vulnerabilities.

3.1 Examination of open source CMS applications

To assess an application's susceptibility to Session Fixation, we adhered to the following testing methodology (see Fig. 3): First we verified that the application indeed issues SIDs before any authentication processes have been undertaken. Then, we tested if the application leaves the SID unchanged in case a successful authentication process has happened. If these tests could be answered with 'yes', we concluded that under certain circumstances the application could expose susceptibility to Session Fixation. The final test probed which attack vectors (see Sec. 2.4) were applicable.

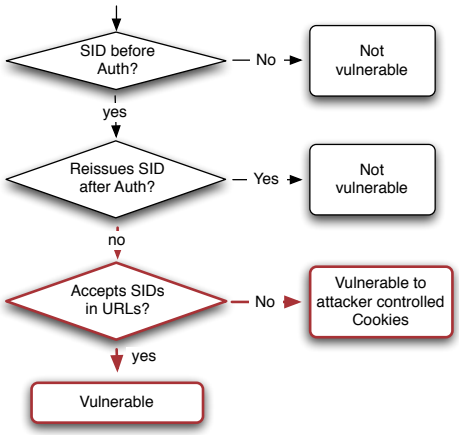

Figure 3: Testing methodolgy

We considered several open source Web applications in their configuration 'out-of-the-box'. The applications implement user management on their own. The results are given in Table 1. A plus sign (+) in a column denotes that the application is vulnerable to the respective attack vector. An application is vulnerable to *Cookie* if it accepts foisted cookies. The attacker can be successful if he manages to set a cookie at the victim's browser. In case the application allows session tracking via a URL parameter, it is vulnerable to *URL*. Finally, those applications which are vulnerable to *SID* allow the attacker to utilize arbitrary SID values that have not been generated by the application in the first place.

Application	Version	Cookie	URL	SID	Lang
Joomla	1.5	+	-	+	PHP
CMSmadesimple	1.6.6	+	-	+	PHP
PHPFusion	7.00.06	-	-	+	PHP
Redmine	0.9.2	+	-	-	PHP
XWiki	2.0.2.24648	+	-	-	Java
JAMWiki	0.9	+	+	+	Java
Wordpress	2.9.1	-	-	-	PHP
Novaboard	1.1.2	+	-	+	PHP
PHPBB	3.0.6	-	-	-	PHP
SimpleMachinesForum	1.1.11	-	-	-	PHP
Magento Shop	1.3.4.2	+	-	-	PHP
OSCommerce	2.2 RC 2a	+	-	-	PHP

Table 1: Test results

3.2 Header injection

We analyzed common web application frameworks to comprehend the feasibility of HTTP header injection attacks. Therefore, we implemented script files which either set a cookie or do forwarding respectively. The scripts take attacker controlled input to determine the forwarding target and the cookie value. The results are given in the next sections.

- **PHP:** Usually, HTTP forwarding in PHP is implemented by the use of the `header()` function. We implemented a file that takes one parameter and inserts this into the `Location` header field. We provided valid input but appended the payload to set our cookie. In the initial configuration (PHP 5.3.0, Apache 2.2.11), the cookie was not set but we got a warning message because `header()` may not contain more than a single header line since version 4.4.2 and 5.1.2 and our new line was detected. Then, we downgraded (PHP 5.1.1, Apache 2.0.63) and the attack was successful. As the forwarding worked as expected we were able to set the cookie 'drive-by' without any notice for the victim.

  Cookie setting is done with the `setcookie()` function. We appended a line break and a new `Set-Cookie` header. However, the function URL-encodes the cookie value and thus transforms our `:` and = characters to non-interpreted URL codes. We can avoid URL encoding of our input by using `setrawcookie()`. However, it prohibits control characters in the value field.

- **J2EE:** For the Java test scenario, we used Tomcat 6.0.20 as a servlet container. The redirection could not be successfully spoofed. The payload was treated as part of the URL. During our cookie setting approach, we got a `java.lang.IllegalArgumentException` due to the control characters in the payload.

- **CherryPy:** We made use of CherryPy (version 3.1.2) to test the Python functions. The injection to the `Location` header was successful whereas the cookie value turned out to be not exploitable.

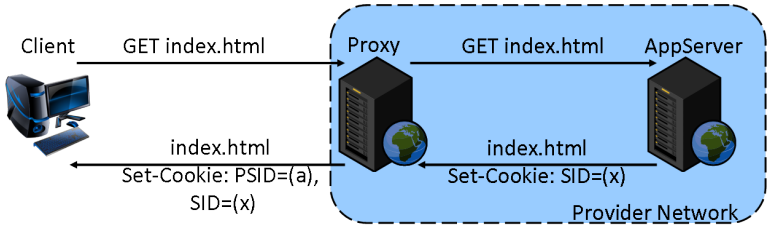

Figure 4: Introduction of the proxy session identifier.

- **Perl:** We also implemented a Perl (version 5.10.1 with `CGI.pm` version 3.48) script that does nothing but forwarding to a given site. Indeed, we managed to set a cookie at the victim's site, however, for some reason the equal sign between the cookie name and its value is finally coded as a colon and a blank. So, we got a cookie with a given value but without a name. Then, we inserted two equal signs and the second one was not recoded. Actually, we were able to set an arbitrary cookie which name ended with a colon. The cookie setting scenario was more difficult due to URL encoding of cookie names and values. We were not able to set our own cookie given that we could influence the value of an unimportant cookie.

- **Ruby on Rails:** For Rails (version 1.9.1), we omitted the tests for header injection in forwarding sites as this vulnerability was recently patched [Web08, Sec09]. During the cookie task, we faced the same situation as in the Perl scenario. Cookies must be declared and cannot be set as ordinary headers. The value that we inserted is thus first URL encoded and never interpreted. So, we did not find a way to escape the cookie value context.

4 Session Fixation Protection Proxy

In this section, we propose a method for transparent, frame-work level protection against Session Fixation attacks. Our approach allows to secure vulnerable applications without changing the application itself. This is achieved by introducing a proxy which monitors the communication between the user and the vulnerable application. The proxy implements a second level session identifier management (see Fig. 4). In addition to the SIDs that are set by the application, the proxy issues a second identifier (PSID). The set of SID and PSID is stored by the proxy. Only requests that contain a valid combination of these two values are forwarded to the application (see Fig. 5).

Requests that are received with an invalid combination are striped of all `Cookie` headers before sending them to the server. Furthermore, the proxy monitors the HTTP requests' data for incoming password parameters. If a request contains such a parameter, the proxy assumes that an authentication process has happened and renews the PSID value, adds an according `Set-Cookie` header to the corresponding HTTP response, and invalidates the

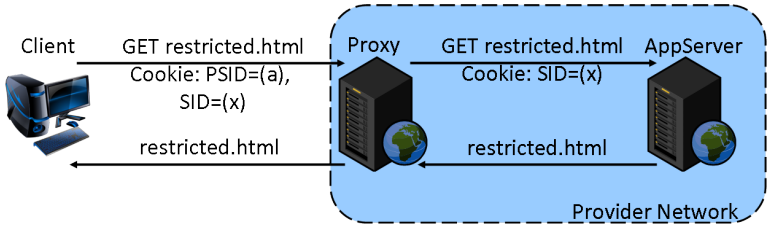

Figure 5: Verification of the proxy session identifier.

former PSID/SID combination.

Hence, the session's security does no longer lie in the SID of the vulnerable application but in an additional identifier issued by the proxy. Accordingly this proxy can be put in front of almost any vulnerable application, giving the application's developers time to fix it.

5 Related work

This paper serves two purposes: comprehensively documenting the Session Fixation vulnerability pattern and proposing a novel method for protecting against such attacks. We structure this section accordingly.

Vulnerability documentation: To the best of our knowledge, [Kol02] was the first public paper on Session Fixation. It describes the basic fundamentals of the attack and the most obvious attack vectors. In contrast to this paper, it provides no information about the spreading of the vulnerability or more advanced attack schemes. Furthermore, OWASP and WASC added articles about Session Fixation to their security knowledge bases. The OWASP article [(OW09] briefly names common Session Fixation issues and attack vectors. In contrast, the WASC article [(WA10] also provides small code examples of different attacks. Both sources name the HTTP response splitting attack as an attack vector, but they do not discuss its impact in today's world of web applications. Furthermore, they lack a general rating of the attack.

Related protection mechanisms: Web application firewalls are server-side proxies that aim to mitigate security problems. However, as the OWASP best practices guide on Web Application Firewalls (WAF) [OWA08] states, current WAFs can only prevent Session Fixation "if the WAF manages the sessions itself." In comparison, our approach does not touch the application level session management and only introduces additional security related information for each request. Furthermore, several protection techniques have been proposed that utilize proxies to mitigate related Web application vulnerabilities. [KKVJ06, Joh06] describe proxy solutions to counter Cross-site Scripting attacks, [JW06, JKK06] propose related techniques for Cross-site Request Forgery protection.

6 Conclusion and Future Work

In this paper, we thoroughly examined Session Fixation: We showed how Session Fixation attacks work (see Sec. 2.2), identified the root cause of this vulnerability class (Sec. 2.3), and documented existing attack vectors (Sec. 2.4).

As mentioned in the introduction, the public level of attention to Session Fixation is comparatively low. We tried to assess if this apparent ignorance is justified. For this purpose, we conducted two sets of tests (Sec. 3). First, we examined open source applications for evidence that their developers were aware of the vulnerability class, i.e., we tested if session identifier were changed after an authentication process. Out of 12 only 4 applications take this measure. Consequently, the remaining 8 applications would be vulnerable to Session Fixation if a supporting problem, such as a header injection flaw, exists (one application was vulnerable out of the box due to URL-support; the issue has been fixed in the meantime). Secondly, motivated by the outcome of the first set of tests, we examined various Web application programming frameworks with respect to protection against header injection flaws. These tests resulted in the observation that the majority of the regarded frameworks indeed take measures to protect against header injection vulnerabilities, thus, indirectly protecting otherwise vulnerable applications against Session Fixation attacks.

Finally, we outlined a novel method for framework-level, transparent protection against Session Fixation. Our solution introduced a dedicated server-side Web proxy that adds a second identifier value (PSID) to the application's outgoing requests. This value is treated as a companion to the application's SID. Only incoming SID values that appear together with a valid PSID are relayed to the application. Session Fixation attacks are successfully prevented as the PSID is renewed by the proxy after an authentication process has happened. As a next step we will implement and thoroughly test the proxy.

References

[Alc07] Wade Alcorn. Inter-Protocol Exploitation. Whitepaper, NGSSoftware Insight Security Research (NISR), `http://www.ngssoftware.com/research/papers/InterProtocolExploitation.pdf`, March 2007.

[Dab09] Arshan Dabirsiaghi. Cross-protocol XSS with non-standard service ports. TechNote, `http://i8jesus.com/?p=75`, August 2009.

[End02] David Endler. The Evolution of Cross-Site Scripting Attacks. Whitepaper, iDefense Inc., `http://www.cgisecurity.com/lib/XSS.pdf`, May 2002.

[FGM+99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, `http://www.w3.org/Protocols/rfc2616/rfc2616.html`, June 1999.

[Gra07] Robert Graham. SideJacking with Hamster. [online], `http://erratasec.blogspot.com/2007/08/sidejacking-with-hamster_05.html`, (02/02/10), August 2007.

[JKK06] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Preventing cross site request forgery attacks. In *Proceedings of the IEEE International Conference on Security and Privacy for Emerging Areas in Communication Networks (Securecomm 2006)*, 2006.

[Joh06] Martin Johns. SessionSafe: Implementing XSS Immune Session Handling. In Dieter Gollmann, Jan Meier, and Andrei Sabelfeld, editors, *European Symposium on Research in Computer Security (ESORICS 2006)*, volume 4189 of *LNCS*, pages 444–460. Springer, September 2006.

[JW06] Martin Johns and Justus Winter. RequestRodeo: Client Side Protection against Session Riding. In Frank Piessens, editor, *OWASP Europe 2006*, May 2006.

[Kam09] Samy Kamkar. phpwn: Attack on PHP sessions and random numbers. Security Advisory, `http://samy.pl/phpwn/`, August 2009.

[KKVJ06] 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.

[Kle04] Amid Klein. "Divide and Conquer" - HTTP Response Splitting, Web Cache Poisoning Attacks, and Related Topics. Whitepaper, Sanctum Inc., `http://packetstormsecurity.org/papers/general/whitepaper_httpresponse.pdf`, March 2004.

[KM00] D. Kristol and L. Montulli. HTTP State Management Mechanism. RFC 2965, `http://www.ietf.org/rfc/rfc2965.txt`, October 2000.

[Kol02] Mitja Kolsek. Session Fixation Vulnerability in Web-based Applications. Whitepaper, Acros Security, `http://www.acrossecurity.com/papers/session_fixation.pdf`, December 2002.

[(OW09] The Open Web Application Security Project (OWASP). Session Fixation. TechNote, `http://www.owasp.org/index.php/Session_Fixation`, February 2009.

[OWA08] OWASP German Chapter. OWASP Best Practices: Use of Web Application Firewalls. [whitepaper], `http://www.owasp.org/index.php/Category:OWASP_Best_Practices:_Use_of_Web_Application_Firewalls`, July 2008.

[Rud01] Jesse Ruderman. The Same Origin Policy. [online], `http://www.mozilla.org/projects/security/components/same-origin.html` (01/10/06), August 2001.

[Sec09] SecurityFocus. Ruby on Rails 'redirect_to' HTTP Header Injection Vulnerability. TechNote, `http://www.securityfocus.com/bid/32359`, December 2009.

[Top01] Jochen Topf. The HTML Form Protocol Attack. TechNote, `http://www.remote.org/jochen/sec/hfpa/hfpa.pdf`, August 2001.

[(WA10] The Web Application Security Consortium (WASC). Session Fixation. TechNote, `http://projects.webappsec.org/Session-Fixation`, January 2010.

[Web08] Heiko Webers. Header Injection And Response Splitting. TechNote, `http://www.rorsecurity.info/journal/2008/10/20/header-injection-and-response-splitting.html`, October 2008.

[Zal06] Michal Zalewski. Cross Site Cooking. Whitepaper, `http://www.securiteam.com/securityreviews/5EP0L2KHFG.html`, January 2006.