Reliable Protection Against Session Fixation Attacks

Martin Johns
SAP Research
martin.johns@sap.com

Bastian Braun
ISL, University of Passau
bb@sec.uni-passau.de

Michael Schrank
ISL, University of Passau
mschrank@gmx.net

Joachim Posegga
ISL, University of Passau
jp@sec.uni-passau.de

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. A successful attack allows the attacker to fully impersonate the victim towards the vulnerable Web application. We analyse the vulnerability pattern and identify its root cause in the separation of concerns between the application logic, which is responsible for the authentication processes, and the framework support, which handles the task of session tracking. Based on this result, we present and discuss three distinct server-side measures for mitigating Session Fixation vulnerabilities. Each of our countermeasures is tailored to suit a specific real-life scenario that might be encountered by the operator of a vulnerable Web application.

1. INTRODUCTION

Session Fixation has been known for several years [12]. 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 even though the impact ranges on the same level. Session Fixation is a widespread problem due to the low attention it receives and its 'occurence by default' in new Web applications (see Sec. 2 for details). Prevention is easy during development but fixing vulnerable applications is generally non-trivial.

Our contribution is twofold: For one, we provide an approach for transparent, light-weight protection on the framework level. It allows 'patching' Web applications without access to the code but just to the underlying framework. Furthermore, we developed a proxy based solution that implements Session Fixation protection with neither access to the application code nor to the framework. In addition, we explain Session Fixation prevention at development phase and, thus, provide comprehensive protection against Session Fixation vulnerabilities.

SAC'11 March 21-25, 2011, TaiChung, Taiwan.

Paper outline: The remainder of this paper is organised as follows: We present the context of the vulnerability in Section 2. We explain the technical background and the circumstances that lead to Session Fixation vulnerabilities as well as attack vectors. Then, we present our server-side approach to counter the vulnerability (Sec. 3). We show how the vulnerability can be prevented on code-level during development phase. We developed two solutions that fix the vulnerability on framework-level and application-level respectively. After exploring related (Sec. 4) and future work (Sec. 5), we finish the paper with a conclusion (Sec. 6).

2. SESSION FIXATION

In this section, we give a thorough insight into Session Fixation. We first describe how session management is done in current Web applications. Then, we sketch a simple Session Fixation attack to shed light on the general attack process. The underlying deficiencies that lead to Session Fixation vulnerabilities are discussed before more sophisticated attack vectors are presented. Finally, the Session Fixation vulnerability is discussed in the context of its more general, enclosing vulnerability type, the class of Session Hijacking vulnerabilities.

2.1 Technical background

HTTP is a stateless protocol. Past transactions are not protocol inherently linked to incoming requests. 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. A SID mechanism can be implemented by transporting the value either in form of an HTTP cookie or via HTTP parameters [6]. All modern Web application frameworks or application servers, such as PHP or J2EE, implement application transparent SID management out of the box.

It is common practice in modern Web applications to link a user's authentication and authorization state to his current session. In consequence, the SID value becomes the user's de facto authentication credential.

2.2 Session Fixation - The attack

Session Fixation is an attack in which the victim is tricked into using a SID value that is controlled, and thus known,

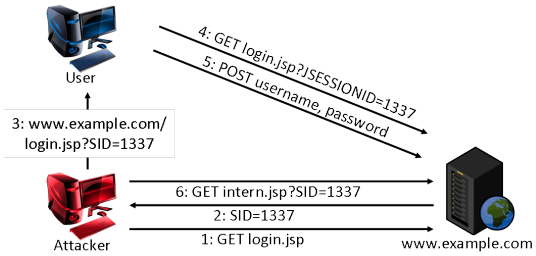

Figure 1: Exemplified Session Fixation attack [12]

by the attacker. This can be achieved either by supplying a crafted URL including this SID as a parameter to the victim (in case that the vulnerable Web application accepts parameter-based SIDs) or by finding a way to set a copy of this SID cookie to the victim's browser (more on this attack vector in Sec. 2.4).

See Fig. 1 for a brief example of the attack via a crafted URL. The individual steps of the attack are the following:

1. The attacker obtains a SID value from the server (1,2).
2. He tricks the victim to issue an HTTP request using this SID during the authentication process (3,4).
3. 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).
4. Now, the attacker can use the SID to access otherwise restricted resources utilizing the victim's authorization context (6).

In Section 2.5, we discuss the relationship between Session Fixation and similar attacks, such as Session Hijacking through Cross-site Scripting.

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 programmer's 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 application frameworks 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 framework level 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: As the SID remains unchanged, any initial SID value which was fixed by the attacker stays "valid" and, thus, can be abused.

2.4 Exploiting Session Fixation

For our considerations of attack vectors, we assume an adversary who has the same capabilities as any other external Web user. The adversary can access Web sites, send E-mails and instant messages. He is able to control all data that resides in his domain. However, he is supposed to be unable to access Web resources or change application states that cannot be accessed or changed by any other unpriviledged user.

As mentioned above, the attacker could try to provide the SID value to the victim through a crafted URL, in case the application accepts session identifiers in GET parameters. However, as current Web applications only very seldom still allow SIDs in URLs [18], a realistic attack scenario would very likely require the adversary to illegitimately set a cookie. We could identify several ways to set a cookie at the victim's site [18]:

For one, 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. In this case, he inserts appropriate JavaScript code into a web page in the same domain, using the `cookie.write()` function. 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 http-equiv="Set-Cookie">` tags. Unlike a cookie stealing attack, e.g. for Session Hijacking, the victim does not have to be logged in, thus, the attack also works at the public part of the application. The attacker can also target a Web page in a vulnerable subdomain to insert his payload. Then, the cookie is valid for the whole domain.

Furthermore, cross-protocol attacks [22, 4] could be utilized. Such attacks 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. Such attacks cannot be used for most XSS-based attack vectors, as JavaScript's same-origin policy [17] explicitly includes the port value of a given URL as a mandatory component when determining a component's origin and the vulnerable services run on non-HTTP ports. However HTTP cookies are shared across ports [13]. Thus, cross-protocol attacks can be utilized to set long-lived session cookies as outlined above.

Alternatively, the attacker could attempt a *HTTP Header Injection attack* [11]. If the targeted Web application is vulnerable to such attacks, this enables the attacker to control parts of the HTTP response header that is retrieved by the user. This in turn allows the attacker to craft a `Set-Cookie`-header which contains the fixed SID value.

Finally, in the past, we saw browser vulnerabilities that allowed the attacker to set the cookie from a foreign domain [23].

Besides finding an enabling vulnerability, the attacker faces additional obstacles. He has to lure the victim into logging in his account and has a window of opportunity of unknown

and limited duration. The attacker can bypass the latter by automating his access attempts. Nonetheless, a successful attack allows the attacker to fully impersonate the victim while it is generally not obvious to the victim to be under attack, especially if he is not familiar with Session Fixation.

2.5 Session Fixation - Context

An HTTP session can be considered as the abstraction of a dedicated communication channel between client and server. Only the knowledge of the communication channel's name (SID) is needed to access this channel. Hence, knowing the SID enables the adversary to conduct a Session Hijacking attack.

To address this problem, additional measures, such as browser recognition (which can be trivially circumvented) or IP binding, have been proposed. Though these measures raise the bar they can not finally solve the problem. Instead, they bring new problems under certain circumstances. For instance, IP binding makes a service unusable if accessed from anonymity networks that tend to send packages from changing IP addresses. A mobile device switching from 3G network to wireless LAN at home gets a new address and looses the current session with unsaved data. Network Address Translation (NAT) is used in company as well as university networks. All requests from the same network appear to come from the same address and thus eliminate IP binding protection. That is why we disregard such additional security measures for the remainder of this paper unless any measure plays a decisive role.

As conventional man-in-the-middle attacks have stopped being the easiest hijacking attack due to switched networks and SSL secured channels, attackers now target the communication's end points. Especially Session Hijacking via Cross-site Scripting (XSS) [5] has become a considerable threat.

The Session Fixation attack is similar to the Session Hijacking attack via XSS in that it needs a vulnerability to prepare the attack. Like the XSS vulnerability allows the attacker to steal the SID, i.e. to gain knowledge of the communication channel's name, the Session Fixation attack needs a preceding attack to determine the victim's SID before connection establishment to the Web application. So, the attacker sets up a new session and receives a SID. Next, he makes the victim use the same SID. Thus, he transfers the communication channel's end point to the victim. Finally, the attacker can take over the victim's session after the latter authenticates himself against the application. Thus, the attacker uses the authenticated communication channel addressed by the well-known SID without ever authenticating himself.

The main reason for the Session Fixation vulnerability lies in the missing renaming of the communication channel after authentication. The SID is not security-critical data before authentication as the user is still unknown and neither the user nor the channel are trustworthy. However, after the authentication process has been passed successfully, the same communication channel with the same name has become trusted and security-critical. The same data must turn from untrustworthy to trusted as the Web application must rely on the opponent's identity. So, only the authorized client is supposed to know the 'ticket' to the established communication channel. This can be easily guaranteed by renewing the SID after every authorization raise, e.g. from unauthenticated user to an authenticated user but also from unpriviledged user to administrator.

3. SERVER-SIDE MEASURES AGAINST SESSION FIXATION

In this section, we list three alternative approaches to counter Session Fixation. The proposed techniques were designed to fit different situations in respect to the degree of control of the vulnerable application's source code or the application server respectively.

3.1 Code-level countermeasures

As described above, the root cause of Session Fixation problems is in general a mismatch in the implementations of the session handling, which usually is done on the framework level, and the authentication management, which is realized on the application layer. Consequently, to be secure against Session Fixation, the application's developer has to renew a user's session identifier manually every time this user's authentication state changes (see Listing 1). Note that only the SID is renewed but the stored session data (e.g. a shopping cart) is then tied to the new SID.

```
1 if ($authentication_successful){
2     $_session["authenticated"] = true;
3     session_regenerate_id();
4 }
```

Listing 1: Example – code based Session Fixation protection in PHP [16]

While for newly written applications this requirement can be fulfilled rather straight forward, the same task might prove hard for non-trivial legacy applications, depending on the complexity of the application's authentication management and its degree of encapsulation within the code base.

In addition, assessing if a given application is susceptible to Session Fixation based on the application's source code alone is also non-trivial. In most cases, a manual test through monitoring and manipulating HTTP communication with the application is easier (see Fig. 2):

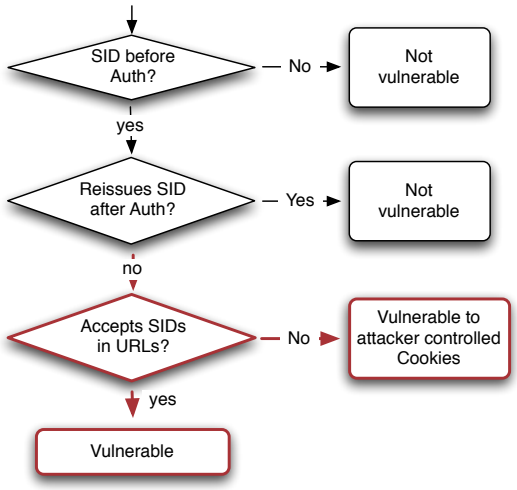

Figure 2: Testing methodolgy

To experimentally test an application's susceptibility to Session Fixation attacks, one can adhere to the following testing methodology: First, it has to be verified that the application indeed issues SIDs before any authentication processes have been undertaken. Then, it should be tested if the application leaves the SID unchanged in case a successful authentication process has happened. If these tests could be answered with 'yes', it can be concluded that under certain circumstances the application exposes susceptibility to Session Fixation attacks.

3.2 Protection on the framework level

As described above, the divide between the framework's session tracking and the application's authentication management is responsible for Session Fixation vulnerabilities. To overcome this divide, we designed a transparent, lightweight solution that takes protective measures on the framework level.

3.2.1 Protection methodology

Our approach functions by mirroring the advised behaviour of Section 3.1 within the application framework: Whenever an authentication process has been executed, the session identifier gets regenerated. However, on the framework level no knowledge about the internal processes of the application exists. Therefore, the protection mechanism has to deduct that an authentication process has taken place through observations of data that is available on the framework level.

While many characteristics in respect to observable application behaviour is depended on the utilized combination of application framework and server, we expect one data source to be available universally: The ongoing HTTP communication between user and application. Therefore, our solution aims to derive the information regarding authentication processes from this data.

We propose the following methodology: The countermeasure is integrated in the framework's component that is responsible for parsing incoming HTTP requests. These requests are examined whether they contain HTTP parameters that might carry password data. Such parameters can simply be identified by their name, which in turn was preconfigured by the application's operator. Whenever such a parameter is detected in an incoming request, the framework internal functions for session identifier regeneration are triggered to create a new SID value for the user. This approach renews the session identifier even if the authentication attempt fails. However, this does not pose a problem as the user and the application then share a new valid SID.

3.2.2 Implementation and evaluation

For our practical experiments, we chose the J2EE application framework [19] as the implementation target. We realised the actual protection mechanism in the form of a J2EE filter. J2EE filters are a properly defined way to add framework components to applications that intercept all incoming and/or outgoing HTTP communication (see Fig. 3). Our filter implements the functionality as described above: All incoming HTTP requests are examined for HTTP parameters which carry the name of a pre-configured password field. If such a parameter was found, the J2EE session container is triggered to reissue a fresh JSESSIONID value to the user's session - so, the session data remains with a new identifier.

Realising the mechanism in the form of a J2EE filter has several advantages: Foremost, no changes to the application-server have to be applied - all necessary components are part of a deployable application. Furthermore, to integrate our mechanism into an existing application only minor changes to the application's `web.xml` meta-file have to be applied. The only configuration that has to be done is providing the name(s) of the application's password parameter(s). Thus, outfitting an existing J2EE application with our solution is easily and quickly done.

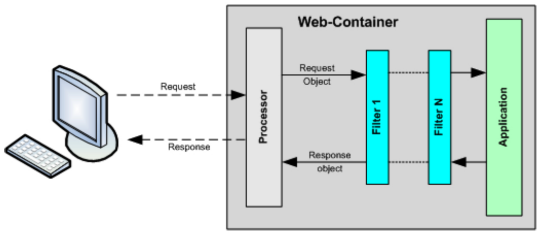

Figure 3: J2EE Filter

We tested our implementation manually using a vulnerable J2EE application. For this purpose, we chose the open source J2EE based Wiki JamWiki, Version 0.8.0 [1] which is susceptible to Session Fixation[1]. After installing the software on our testsystem, we verified that the installed version is in fact vulnerable, using the testing method outlined in Section 3.1.

In the next step, we added our J2EE filter to the installation and entered the name of JAMWiki's password parameter ("`j_password`") to the filter's configuration file. Finally, after restarting the application server, we verified that after every login attempt, the JSESSIONID value indeed changes and, thus, the vulnerability was properly mitigated.

3.3 Protection via a reverse proxy

In certain situations, it is neither feasible to fix a vulnerable application's source code nor to apply a framework level countermeasure, as described in Section 3.2. Such scenario include, for instance, the hosting of closed source applications, mission critical applications which cannot be patched timely because of otherwise expected downtime, or legacy applications that require frameworks which do not support session re-generation, such as PHP prior to version 4.3.2 [16]. Furthermore, sometimes a short-term solution is needed even if an application inherent fix can be applied later, e.g., when an identified vulnerability is under active attack and the fix is still under development.

In all these scenarios an application external solution is required – a generic self standing protection mechanism that does not necessitate alteration of the actual application or its application server. In this section, we propose a method for transparent, proxy-based protection against Session Fixation attacks for such scenarios.

3.3.1 Challenges

To implement such a solution several hurdles have to be overcome. In this section, we list the identified problems and briefly outline our corresponding solutions.

[1]We discovered the software's vulnerability during our first experiments with Session Fixation. The JAMWiki's authors have been notified and a fix is on its way.

- **Application external solution:** The protection mechanism necessarily has to take its measures outside of the actual application as in the given situation (see above) resolving the situation directly at the application is not possible. Consequently, we designed our solution in the form of a server-side reverse proxy.

- **Complementing the application's session management:** Our proxy has no direct control over the application's internal session management, unlike our solution that we presented in Section 3.2. For this reason, our solution has to be able to invalidate fixed sessions while maintaining legitimate application usage. We solve this problem through the introduction of a secondary session identifier that is issued by the proxy (PSID). The proxy's identifier management component is tightly secured against Session Fixation and only requests which carry a valid PSID are forwarded by the proxy to the actual Web application.

- **Login detection:** Similar to the framework-level solution described in Section 3.2, detecting that a login process has happened is crucial for the solution to function properly. We tackle this problem analogously.

In the following section we give details how we solved the above mentioned problems.

3.3.2 *Protection methodology*

As outlined above, we introduce a proxy which monitors the communication between the user and the vulnerable application. The proxy implements a second level session identifier management. In addition to the SIDs that are set by the application, the proxy issues a second identifier (the *proxy SID* – PSID).

Whenever an HTTP request without a PSID value is received by the proxy, this request is regarded to be the user's very first request to the application. If the request carries any stale SID values, such data is discarded. For the corresponding HTTP response a fresh PSID value is generated and attached to the response via `set-cookie` (see Fig. 4). In the course of the following HTTP communication, the application's responses are monitored for outgoing SID values that are to be assigned from the application to the user. If such a value is detected, the combination of the PSID and SID value is stored by the proxy. From now on, 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 of SID/PSID are treated as if they would carry no session information. Consequently, they are stripped off all `Cookie` headers before sending them to the application and are outfitted with a fresh PSID value upon response.

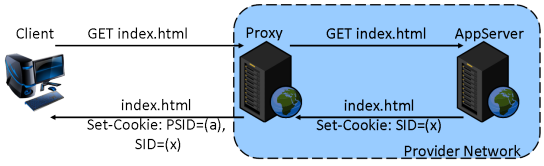

Figure 4: Introduction of the proxy session identifier.

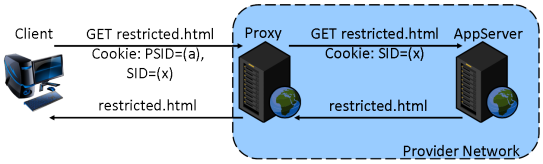

Figure 5: Verification of the proxy session identifier.

To provide protection against Session Fixation, 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 former PSID/SID combination. This way, only the session identifier is renewed whereas the session data remains unchanged. The SID is even renewed if the authentication attempt fails. This, however, is no threat as the new SID does not carry any security assumptions.

3.3.3 *Implementation and evaluation*

To test our approach, we implemented a prototype in Python. The reason why we chose Python instead of another programming language is because Python's features make it ideal for fast prototyping.

We utilized *CherryPy* [2] as the basis for the proxy server. CherryPy is a lightweight Web framework which offers a smart interface for developers of Web applications. CherryPy only provides the framework for handling incoming requests and rendering responses to clients, hence, providing the proxy's frontend. For backend communication with the vulnerable application the python module *Urllib2* [3] was used. It provides methods for retrieving data from a URL using either HTTP GET or POST requests. Furthermore it offers the needed possibility to access cookies which were involved within the communication with the URL.

The proxy implements the issuing and verification of PSIDs as described above. The PSID value is stripped off incoming requests before they are forwarded to the application to avoid potential problems that some applications might expose when they receive unexpected parameters or headers.

To verify the provided protection, we again tested our implementation against JAMWiki. We both passively observed the proxy's behaviour in respect to ongoing login processes as well as actively attempted Session Fixation attacks.

3.4 Discussion

In this section we presented three different protective measures that can be utilized by a Web application's operator to avoid Session Fixation problems. Each of the measures is targeted at a distinct scenario in respect to the level of control that the operator has when it comes to altering the Web application's internals.

Fixing the problem within the application logic through reworking the authentication handling code, as shown in Section 3.1, should always be the first choice as long as no restrictions exist when it comes to altering the source code and timely applying the resulting security patch. The main problem that can be encountered in this scenario is that the authentication and session handling code in a given application might turn out to be non-trivial and spread thorough the code. For the fix to function properly, it is essential

that *all* code segments, in which a user's authorization level changes because of an authentication process, are addressed correctly. If one of such processes is missed in the creation of the security patch, the protection is incomplete. For this reason, handling of Session Fixation should be integral part of the software development process and be addressed from the begin on.

The second proposed measure (see Sec. 3.2) is applicable if a direct alteration of the application's source code is not feasible but the utilized application server and framework are under full control. Such situations mainly arise, if the operator of the application is a different entity than the application's developer. This applies, e.g., for third party components, closed source applications, or legacy applications for which the original author has left the company long time ago. The described approach provides reliable protection, as every authentication process causes the framework to renew the SID value. Furthermore, the approach is very light-weight. This stems from two characteristics: For one, the mechanism is completely stateless. It does not require temporary storage of any data as it only reacts based on incoming password parameters. Furthermore, it is closely integrated in the existing framework infrastructure. Consequently, there is no need to execute any complex operations on its own – all the hard work, such as parsing the HTTP headers and parameters, is already handled by the application framework. These characteristics resulted in a runtime behaviour that, at least if implemented in the form of a J2EE filter, does not cause noticeable performance overhead. Finally, as the mechanism operates completely transparent to the application an addition to an existing application is easy and straight forward. The main drawback is that the countermeasure is application framework specific. If for a given application changes in the runtime infrastructure are taken, such as exchanging the underlying application server, the protection might be lost and has to be reintroduced – a characteristic that does not apply to handling Session Fixation directly on the source code level.

The third discussed countermeasure (see Sec. 3.3) is to be used whenever no changes at all to the vulnerable system are possible (see above for a list of reasons). It is designed to be completely self-standing and can be set up to protect arbitrary Web applications simply by positioning it between the application and the user. It is very reasonable to expect that the proposed mechanism can easily be integrated in an existing Web application firewall (WAF) [15]. WAFs are in essence Web proxies which were introduced for the exact same scenario as the discussed countermeasure – to protect against Web application attacks without altering the application itself. Consequently, WAFs already handle all operations, such as parsing incoming requests, that are required by our countermeasure. In turn, our countermeasure itself is comparatively light-weight and does not add significant complexity to a WAF's functionality.

As mentioned above, which of the three described measures is to be taken, depends on the given situation. In general, if possible, the solution that is most closely integrated in the application's core functionality should be chosen to reduce setup complexity and avoid potential security regression due to future changes in the application's infrastructure.

4. RELATED WORK

Session Fixation has received little attention in the past, mostly due to the vulnerability's obscurity and the fact that more severe XSS vulnerabilities are still very common in current Web applications.

Vulnerability documentation: To the best of our knowledge, [12] 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. Nevertheless it is the main source for information about Session Fixation up to now. Furthermore, OWASP and WASC added articles about Session Fixation to their security knowledge bases [20, 21]. Finally, we recently published a survey paper on Session Fixation in which we practically assessed how wide spread the vulnerability pattern is in today's applications [18].

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) [15] 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. For one, [10, 7] describe proxy solutions to counter Cross-site Scripting attacks. Furthermore, [8, 9] propose related techniques for Cross-site Request Forgery protection. Finally, [14] presents a proxy for client-side detection of SSL stripping attack.

5. FUTURE WORK

The next step will be to transfer the protection to the client-side. As discussed before, Session Fixation is an attack that targets the end-user. In consequence, a user's measures to protect himself are rather limited, if a Web application's operator is unaware or chooses to ignore potential Session Fixation problems. In theory, our proxy based solution (see Sec. 3.3) is suited to be installed on the client-side as a companion to the Web browser. However, unlike on the server-side, the issue of configuring the proxy is non-trivial. To enable the proxy's protection mechanism, the names of the SID and password parameters have to be made known to the proxy. Expecting a manual per-site configuration done by the user is unrealistic. Instead suitable measure to do this configuration step automatically have to be found, i.e., robust heuristics that reliably identify password and SID parameters. Achieving this is still subject to future work.

Another step of our future work is the analysis of Session Fixation attack surface in identity management systems. The authentication protocol as well as the actual authentication token are of major importance in this scenario. New protection measures need to be taken in presence of multi-party authentication protocols to prevent access to a set of services.

6. CONCLUSION

In this paper, we thoroughly examined Session Fixation: We showed how Session Fixation vulnerabilities arise, how

they can be exploited, what the impact of a successful attack is, and what relations to more common issues exist (Sec. 2). In addition, we described the fundamental problem of HTTP sessions and the mismatch of responsibilities that leads to Session Fixation scenarios. We outlined several attack vectors that may be used to prepare Session Fixation attacks. We pointed out the common context of these attacks as well as their differences.

Based on our observations, we proposed three distinct server-side measures against Session Fixation: For one, we showed how to avoid the problem in the applications' development phase (Sec. 3.1). Furthermore, we presented two approaches to fix running Web applications with reasonable interference (Secs 3.2 and 3.3). These countermeasures require minimal configuration effort, which solely consists in providing the parameter names of the session identifier and password fields, thus, allowing fast and easy mitigating freshly detected Session Fixation issues. Our countermeasures are robust in respect to failed login attempts as the actual link between the server-side session storage and the application's user is preserved in all cases.

In sum, we provided defensive solutions for all potential scenarios in respect to control over an application's source code which can be encountered when operating a Web application and, thus, achieve complete protection coverage against Session Fixation attacks.

7. ACKNOWLEDGEMENTS

This work was in parts supported by the EU Project WebSand (FP7-256964), http://www.websand.eu. The support is gratefully acknowledged.

8. REFERENCES

[1] JAMWiki. [software], http://jamwiki.org/, Version 0.8.0, December 2009.

[2] CherryPy - Lightweight, pythonic web framework. [software], http://www.cherrypy.org/, April 2010.

[3] URLlib2 - Python HTTP URL opener library. [software], http://docs.python.org/library/urllib2.html, April 2010.

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

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

[6] 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.

[7] M. Johns. SessionSafe: Implementing XSS Immune Session Handling. In *European Symposium on Research in Computer Security (ESORICS 2006)*, volume 4189 of *LNCS*. Springer, September 2006.

[8] M. Johns and J. Winter. RequestRodeo: Client Side Protection against Session Riding. In F. Piessens, editor, *OWASP Europe 2006*, May 2006.

[9] N. Jovanovic, C. Kruegel, and E. 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.

[10] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic. Noxes: A Client-Side Solution for Mitigating Cross Site Scripting Attacks. In *21st ACM Symposium on Applied Computing (SAC 2006)*, April 2006.

[11] A. 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.

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

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

[14] N. Nikiforakis, Y. Younan, and W. Joosen. Hproxy: Client-side detection of ssl stripping attacks. In *Seventh Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA'10)*, 2010.

[15] 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.

[16] PHP Group. session_regenerate_id(). PHP documentation, [online], http://www.php.net/manual/de/function.session-regenerate-id.php (4/4/10), June 2010.

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

[18] M. Schrank, B. Braun, M. Johns, and J. Posegga. Session Fixation - the Forgotten Vulnerability? In *Proceedings of GI Sicherheit 2010, Lecture Notes in Informatics (LNI)*, 2010.

[19] Sun Microsystems Inc. J2EE - Java Platform Enterprise Edition 5. [online], http://java.sun.com/javaee/technologies/javaee5.jsp, (05/05/07), 2007.

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

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

[22] J. Topf. The html form protocol attack. TechNote, http://www.remote.org/jochen/sec/hfpa/hfpa.pdf, August 2001.

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