Prevent Session Hijacking by Binding the Session to the Cryptographic Network Credentials

Willem Burgers[1], Roel Verdult[1], and Marko van Eekelen[1,2]
willemburgers@student.ru.nl, {rverdult,marko}@cs.ru.nl

[1] Institute for Computing and Information Sciences,
Radboud University Nijmegen, The Netherlands.

[2] School of Computer Science,
Open University of The Netherlands.

Abstract. Many cyber-physical applications are responsible for safety critical or business critical infrastructure. Such applications are often controlled through a web interface. They manage sensitive databases, drive important SCADA systems or represent imperative business processes. A vast majority of such web applications are well-known to be vulnerable to a number of exploits. The focus of this paper is on the vulnerability of session stealing, also called *session hijacking*. We developed a novel method to prevent session stealing in general. The key idea of the method is binding the securely negotiated communication channel to the application user authentication. For this we introduce a server side reverse proxy which runs independently from the client and server software. The proposed method wraps around the deployed infrastructure and requires no alterations to existing software. This paper discusses the technical encryption issues involved with employing this method. We describe a prototype implementation and motivate the technical choices made. Furthermore, the prototype is validated by applying it to secure the particularly vulnerable Blackboard Learn system, which is a important and critical infrastructural application for our university. We concretely demonstrate how to protect this system against session stealing. Finally, we discuss the application areas of this new method.

Keywords and phrases: software security, web applications, cross site scripting, session stealing, session hijacking.

1 Introduction

Web applications are hard to secure. Many web applications suffer from security vulnerabilities that can be exploited by an attacker. A widely used method to secure web applications involves the creation of an application session for which the user has to authenticate using a registered login name and corresponding password. Before such a session is established, a secure encrypted communication channel is negotiated at a network level to ensure confidentiality. However, the creation of a session and the use of encrypted communication is not sufficient to make an application secure against all attacks.

The focus of this paper is on one of the serious attacks: *session stealing* or *session hijacking*. This is aimed at the session mechanism itself. An adversary takes over a valid user session with a recovered authentication token that is distributed to an genuine user. From this point on we call such a valid authentication token a session identifier (*session ID*). Most modern websites use encrypted communication between the client and the server to prevent an adversary from eavesdropping this session ID. However, it does not prevent stealing the session ID by means of malicious scripts or rogue browser plug-ins.

Sessions in web applications are very common on many infrastructural application areas. Many business critical applications and safety critical applications use a session mechanism. Also cyber-physical applications often use a web server and a session mechanism for communication purposes. Supervisory control and data acquisition (SCADA) systems are well-known to be vulnerable to session hijacking at the transport layer [9]. Improving the safety of sessions contributes to increasing the security level of such applications.

A user whose session is stolen may not notice anything strange while the attack is performed, since the execution of the script may run in the background without changing anything on the screen of the user. This means that the user can be offered little advice in order to prevent such attacks. The main advice is to avoid surfing to pages hosted on the same domain that could be infected by malicious scripts during an active session. This means always closing an open session before surfing to a website that does not require the same session credentials. Such advice does not help much if the application for which the session is opened, is itself vulnerable to cross site scripting. This is the case for many web applications where data can be entered by users and is to be read by other users. Vulnerabilities can occur if the output, generated from the entered data, is not properly encoded. Output encoding prevents executable scripts by replacing meaningful characters with harmless annotated symbols. For example, when an adversary is able to post a malicious script, it could compromise the complete website and steal all active sessions. In that case an attack can happen directly after a genuine user visits the website only once.

Vulnerabilities like these may greatly reduce the trust of the user in the system. The user feels very insecure since there seems to be no way for the user to prevent such an attack.

Motivational example

The method we propose can be applied in general. As an application example a virtual learning environment is chosen. Such a learning environment is part of the infrastructure: it is a virtual extension of a school/university building which aims to create a safe place for students and teachers. In such a safe place students have confidential discussions with teachers, grades are administered and exams can be held. The activities that take place in the virtual learning environment are for a large part the same as the ones that take place in the physical environment. When within such a virtual learning environment sessions can be stolen without anyone noticing, which could cause that the hard to gain trust is easily lost.

More specifically, our contribution is motivated by the fact that we discovered several vulnerabilities in the Blackboard Learn system used at our university [4]. We demonstrated multiple ways to steal the session identifier and successfully perform a privilege escalation attack. In line with the principles of

responsible disclosure, we have notified Blackboard inc. and informed them of our findings back in July 2011. They reacted quickly with some ad-hoc fixes and formed a special security task-force team to locate the nature of these problems [5,6]. The fixes were mostly improvident, one example is that they try to bind the session to the IP address that is used by a genuine user. Such limitation does not work very effectively for large university networks which operate behind one big routing firewall, they all seem to have the same IP address. Furthermore, such network information is publicly accessible, can be determined remotely (using a malicious XSS script) and is easy to spoof. Although some imminent threats were resolved, a more conceptual solution for such vulnerabilities is preferred. The goal is an independent and general applicable design that works without having access to the source code of the application. Hence, we propose a secure protocol that wraps around any closed source and proprietary system and extends its security with significant protection against session hijacking.

The contributions of this paper:

- a new method of binding the application session to the cryptographic network credentials that effectively prevents hijacking of web sessions;
- a fully functional prototype implementation of the method (for cookies), built and released under the royalty free BSD license.

The structure of this paper:
Sect. 2 discusses session stealing in more detail. Our method to prevent session stealing is presented in Sect. 3. Next, in Sect. 4 we demonstrate a prototype of our proposed method and evaluate its effectiveness in a specific application instance. Finally, we discuss related work in Sect. 5 and conclude in Sect. 6.

2 Session stealing and prevention

Sessions are necessary to keep track of users, to see which pages they visit and if they are allowed to visit them. When a user logs on to a website, a new session is started for that user. The rights of the user to follow links and view webpages are stored in the session data. Upon each page view, the rights should be checked. HTTP is a stateless protocol, so it does not provide this user tracking and access verification. Sessions are therefore implemented in the application which runs on top of HTTP. The session ID is kept by the client to be sent with each HTTP request to let the server know the state of the session and verify the user. A session ID can however be stolen and used by another person. The literature refers to this issue with the terms *session stealing* or *session hijacking*.

2.1 Stealing the session

An adversary with limited access can post a script on a webpage (e.g. via cross site scripting *XSS*) and wait for the genuine user to access the infected website. When the user opens the page, the malicious script executes automatically and gains access to the decrypted credentials. Such a script often tries to recover the session ID and discretely communicates it back to the adversary. A variation of this attack is performed by sending a genuine user a link that triggers a malicious

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

Fig. 1. A XSS attack within a URL. Published by Nikiforakis et al. [10]

script from within the browser. An example of an XSS attack via the URL is given in Fig. 1.

The malicious script sends the cookie of the user to the website of the attacker. With the freshly recovered session ID the adversary gains all the session capabilities of the genuine user without having to authenticate. The session ID is often stored in a cookie, but can also be part of the URL[3]. This latter form is mostly used in older web applications. The form of the session ID is not really relevant, as long as there is a value kept by the client to be sent with every request. This session ID represents the state of the session. In this paper, we will focus on the method that involves cookies, but our solution proposed in Sect. 3 is generic and will also work for other forms.

2.2 Strengths and weaknesses of http-only cookies

There are special cookies that can not be accessed by any script that gets executed in the browser. Such cookies are referred to as http-only cookies, since they are stripped away and added again when the http headers are processed in the browser. This seems to be a powerful countermeasure against scripts that try to steal the credentials from cookies. Nowadays, most globally used services (like Facebook, Google and Microsoft) are accessed through users credentials based on a persistent session ID stored in a http-only cookie. The endless count of these active services increases the threat of users being tricked to install a malicious browser plug-in that eave-drops a session ID and seize their user accounts. An example that clearly demonstrates how to steal sessions with a browser plug-in is Firesheep[4]. It is not exactly a malicious plug-in, but can be used to demonstrate the severity of an adversary on the network. The main problem is that a browser plug-in has access to all decrypted incoming website traffic, including *all* cookies.

Interestingly, the encrypted data and cryptographic credentials are inaccessible for a browser plug-in. The decryption is often handled in the browser core or, preferably, at the network level of the operating system. We propose to use this specific property to prevent session hijacking by binding the application session to the already negotiated cryptographic credentials at the network level.

2.3 Session stealing prevention

There are several papers that address the prevention of session hijacking. Our solution is based on a method proposed in 2006 by Oppliger et al. [11]. Oppliger et al. propose their method as a defense against a man-in-the-middle (*MITM*) attack where the credentials are stolen. Even though the attack is different from the attack we face, the basic idea can still be used.

[3] for example: http://domain.com/index.php?session_id=rj3ids98dhpa0mcf3jc89mq1t0

[4] http://codebutler.com/firesheep/

The idea is to combine the application session with the HTTPS session. Where HTTP is stateless, HTTPS needs to keep the state of a connection. With the combination of HTTPS and the application session, you can make use of the security of the HTTPS session to secure your application session. The coupling of the SSL/TLS session and the application session provides a failsafe.

It is straightforward to detect if a session ID is used by another HTTPS connection. In such case, the server should immediately ask for renegotiation. With such a countermeasure it gets a lot harder to take over an application session if it is cryptographically coupled with the network session. Oppliger et al. combined the sessions by binding the application session to a client certificate. With a client certificate, the user proves to the server that he is indeed who he claims to be. In this paper we propose to use a different form of binding the sessions. Our method does not require client certificates. Client certificates require management and seem to be a hassle to install for inexperienced users. Client certificates are also an optional part of SSL/TLS. Our method uses the cryptographical keys already available in SSL/TLS.

3 Session securing by proxying

This section discusses a new prevention method for session hijacking. First, the general idea is explained in a little more detail. After that, some design details that were made during the implementation are discussed. Also more details are given about the protocol and the inner workings of the method. Finally, there is an attacker model that describes what an attacker can and cannot do.

3.1 Session Binding Proxy

In this paper, we propose Session Binding Proxy (SBP), a method that combines SSL/TLS session-aware authentication with a reverse proxy. This proxy relays the requests to the back end application server only if the client that originally got the application session ID is sending the request. To authenticate a client over HTTPS, you register the SSL session and application session information. When a request with the same application session ID is used with a different SSL session, you know that the session is stolen. By removing the session cookie from the request, the application session is invalidated. The proxy makes sure the HTTPS session and application session combination does not need to be kept inside the application (server). The idea is to use a server side reverse proxy that handles the HTTP(S) requests as they come in and sends them to the back-end application server. The application server should only be accessible by the reverse proxy as shown in Fig. 2.

To administer sessions, the reverse proxy needs to be extended with functionality to read the requests and responses and manage both the SSL/TLS and the application session. First, we present the identifiers bound together. Then, two solutions are proposed to manage them to ensure the validity of the session.

SSL identifiers
There are multiple identifiers for the SSL/TLS connection and session provided by the SSL cipher suite. The most important identifier for the network layer

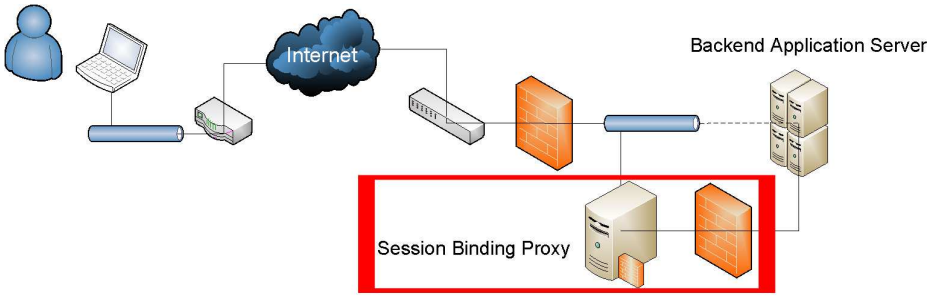

Fig. 2. An application server protected with SBP

is the SSL session ID, which uniquely identifies the current SSL network connection. However, it does not strongly identify a client but rather a connection. For instance, when the SSL session ID is renegotiated, either by a timeout or disconnection, the SSL session ID changes.

An alternative identifier for the SSL/TLS connection is the SSL master key. The master key is part of the SSL handshake, just like the SSL session ID, but is persistent during a session renegotiation. Therefore, the master key can be used to identify multiple SSL connections which represent one client session. The master key is a shared secret between the client's SSL implementation and the server's SSL cipher suite.

Application session identifiers

The way the application session identifier works, depends on the application itself. Most applications use a session ID stored in a cookie [10]. The administrator of the SBP is able to choose which cookies represent the application session identifier and therefore, should be protected. The application might use JavaScript to read other (non session related) cookies in the browser and change parts of the webpage based on this value. Such cookies can obviously not be protected, since it breaks the functionality of the web application if SBP changes the value. Next, we propose two solutions to bind the SSL and application session identifiers together.

Solution 1

The first option is to store the SSL/TLS session and application session combination in memory. When the 'Set-Cookie' header is sent by the application server, the SSL master key and the cookie value pair is stored by the proxy. When the next request from the client comes in, the session cookie value is checked against the pairs that are stored by the proxy. If the incoming pair does not match one of the pairs in memory, the session is invalid. To invalidate the session on the application server, the request is just sent to the server without the Cookie header. The server does not recognize any active session and redirects the user to the login page.

Solution 2

Another method is to authenticate the cookie value with a combination of the SSL master key and a secret key, known to the proxy only. One way of authenticating the cookie is by combining it with an HMAC value. Another way is to encrypt the cookie value. We chose to encrypt the cookie, because when this option is deployed in larger environments, and the proxy is tested next to the

application server (see the dotted line in Fig. 2), the cookies will not be recognizable when encrypted. So, the two systems can not interfere with each other. For new systems that include SBP in the application, an HMAC will suffice.

When encryption is used, the secret key owned by the proxy is hashed together with the SSL master key using a secure hash function. The output of this hash is the key to a AES-256-cbc encryption/decryption function. This means that the encryption/decryption key is different for every client connection. 'Set-Cookie' headers, for specific cookies, are intercepted by the proxy and their values are encrypted before sending them to the client. The client cannot decrypt the resulting cookie, because it does not know the secret key. This method saves memory and synchronization of parallel processes with shared resources is not necessary. Fig. 2 shows some schematics of the layout, with and without SBP (With SBP, the application server is not directly reachable).

In our proposed prototype in Sect. 3.3, the SBP, *Solution 2* is used.

3.2 Session management

This section describes how the SBP handles a request. The first thing that the SBP server does when it gets a request, is redirect the user to the HTTPS port if the user did not connect on that port already. This will start the SSL/TLS handshake to establish the necessary identifiers. Fig. 3 shows the SSL negotiation in the first block (lines 0 to 10). All further traffic passes this SSL connection.

When the SSL connection is made, the application session can be established. The first request sent to the server does not contain a cookie, because the server has not set any cookies yet. The SBP can simply replay the request to the application server. Any request on a page without a session cookie results in a redirect to the login page. When the user logs in, the application server will send a 'Set-Cookie' header. This header is intercepted by SBP and the value of the cookie is encrypted with the key k_c, which is a hash of a secret system key K_p and the SSL master key k concatenated, performed by $k_c \leftarrow hash(K_p||k)$. In our prototype, we use SHA256 as the hashing algorithm. Every encryption with AES-256-cbc (denoted by $\{-\}$) requires a fresh random Initialization Vector (IV) such that an attacker cannot generate multiple session ID values encrypted with the same key and IV. We generate a new random IV for every new 'Set-Cookie' header. The IV is not required to be secret. The cookie is encrypted as follows $\{cookie\}_{k_c} \leftarrow encrypt(cookie, k_c, IV)$. In order to later retrieve the IV, we concatenate it with the encrypted cookie. The encrypted version of the header $\{cookie\}_{k_c}$ and the IV is sent to the client. This process is shown in the second block (lines 11 to 19) in Fig. 3. From this point on with each request by the user, the client sends the encrypted cookie along with every request. When a request is received with encrypted session data, SBP decrypts the value of the cookie and send the plaintext cookie to the back end server. This can be seen in the final block (lines 20 to 25) of Fig. 3.

When the request is sent over the same SSL connection, the same master key will be used and the cookie value decrypts normally. When the request is sent from a different client, the SSL master key differs and the decrypted result will be some random data. When the back end server receives such a request with random cookie data, it tries to load a session that does not exist. The application responds on a non-existing session request with a redirection to the login page.

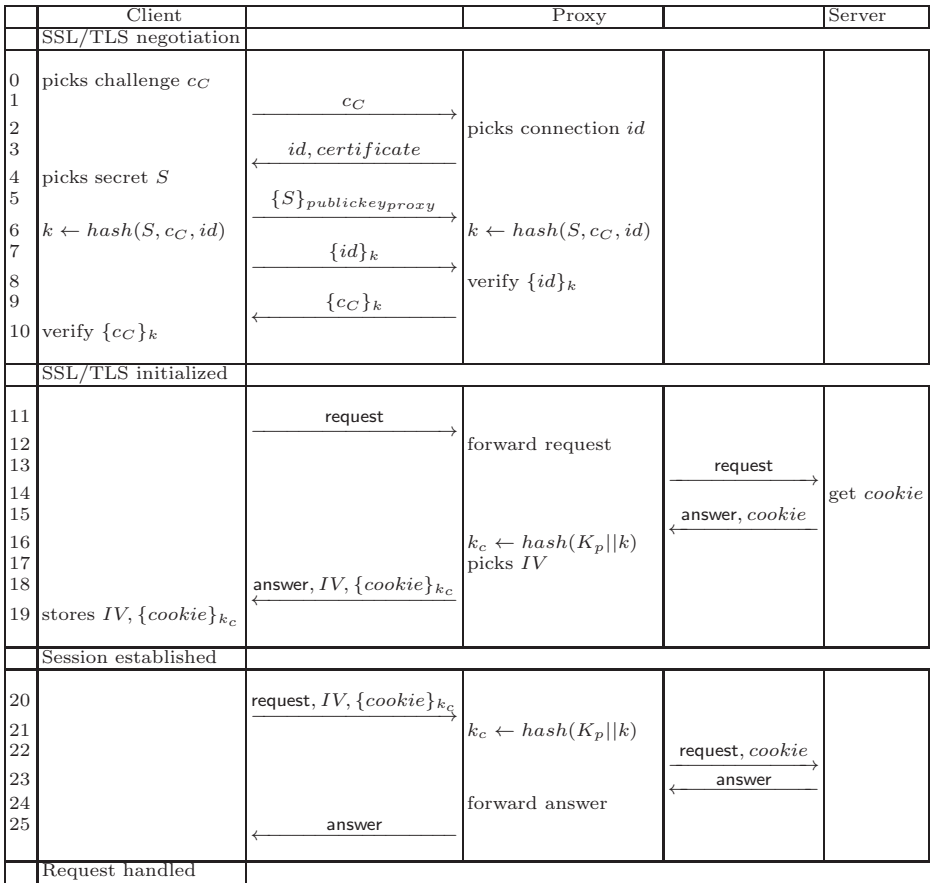

Fig. 3. Session Binding Proxy protocol

The same goes for an expired SSL session. As said in Sect. 3.1, the SSL master key is used for renegotiation, but whenever an SSL session is completely terminated, the corresponding master key expires. Values that are encrypted with an expired master key become invalid and are just ignored. This will result in a session invalidation and the user is logged out and redirected to the login page. SSL session expiration also has an effect on so called long-living cookies. These cookies are needed for a 'Stay signed in' option that allows the user to keep visiting a website with the same session for multiple days or even weeks. We want to improve SBP in the future to handle expired SSL sessions such that long-living cookies can also be used.

3.3 Prototype

To show that the idea works in practice, a prototype for SBP is implemented as a module for the popular reverse proxy server Nginx. Nginx is a very lightweight application and can be used as reverse proxy, webserver and load balancer. It is written in C and highly optimized for performance. Because SBP relies heavily on the efficiency of the reverse proxy, we chose to implement it as a module for a proven to be robust and reliable reverse-proxy server like Nginx. The framework can be used to handle the requests and only the application logic of the cookie

names and SSL master key data should be configured. This was slightly harder than we initially thought however, because the Nginx framework is not very well documented[5]. Nginx is built to work in phases. An HTTP request is processed by all the phases in order, starting from phase 1 all the way up until the response is sent out at phase 10. Each phase can have zero or more handlers. There are ten phases in total as depicted in Fig. 4.

	Nginx Phase	Description
1	NGX_HTTP_SERVER_REWRITE_PHASE	Request URI transformation on virtual server level
2	NGX_HTTP_SERVER_CONFIG_PHASE	Configuration location lookup
3	NGX_HTTP_REWRITE_PHASE	Request URI transformation on location level
4	NGX_HTTP_POST_REWRITE_PHASE	Request URI transformation post-processing phase
5	NGX_HTTP_PREACCESS_PHASE	Access restrictions check preprocessing phase
6	NGX_HTTP_ACCESS_PHASE	Access restrictions check phase
7	NGX_HTTP_POST_ACCESS_PHASE	Access restrictions check post-processing phase
8	NGX_HTTP_TRY_FILES_PHASE	Try files directive processing phase
9	NGX_HTTP_CONTENT_PHASE	Content generation phase
10	NGX_HTTP_LOG_PHASE	Logging phase

Fig. 4. Phases of Nginx

The module hooks into the rewrite phase (phase 3 in Fig. 4) to decrypt and modify the cookie values in the request headers. Then, the request is handled by the reverse proxy module of Nginx. The request is forwarded to the back end and its response is returned to the Nginx proxy and at some point handed to the filters of the module. Filters hook in to phase 10 of Nginx, where they perform some last modification to the response before sending it to the client. In the presence of 'Set-Cookie' headers, the cookie value in the header is encrypted. Finally, the resulting headers and page body are returned to the client.

```
Set-Cookie: s_session_id=609A38D1ECB3A70590BC51D41EA44048
    ; Path=/; Secure; HttpOnly
```

Fig. 5. Cookie sent from backend server to proxy

```
Set-Cookie: s_session_id=CD444464249E9227-Sz/
    I2JEoX4uWvTfvzXAc4r2OAXsMF/MmvZBYcF7CQCFGWBIcq+
    CJbNKwglZbU7G6CGSCI59QDagYhrZQu2RPCXLKRzX/
    Te58QVFMB5Uk5J8SigaTOJY8dr5fLJnUyYGP; Path=/; Secure;
     HttpOnly
```

Fig. 6. Cookie sent from proxy to client

Fig. 5 shows a simple example of a HTTP header from the back end server. An example of the the encrypted cookie is shown in Fig. 6.

The encryption and decryption is done using an AES-256-cbc function, provided by the used OpenSSL cipher suite. It has three main parameters, namely the key, the initialization vector and the data. For the key, the system private key K_p is concatenated with the SSL master key k using SHA256. The output of this hash is the encryption key. In our prototype, the system key K_p is a 256 bit hexadecimal string, randomly generated at the startup of Nginx. The

[5] SBP started out as a bachelor thesis subject for Willem Burgers [1]. A proof of concept of SBP for the thesis was implemented in PHP

initialization vector is a 64 bit random, generated upon each new 'Set-Cookie' header intercepted. In order to decrypt with the same IV, it is placed in front of the cookie value, separated by a '-'. To ensure that the cookie is still handled correctly by the browser, only the value of the cookie is encrypted.

3.4 Attacking the SBP

This section describes the implications of an adversary with access to different levels of the server and client. In Fig. 7 a schematic overview is given of the protection level of SBP. Known attacks against SSL and cookies are represented by arrows on the level they can attack.

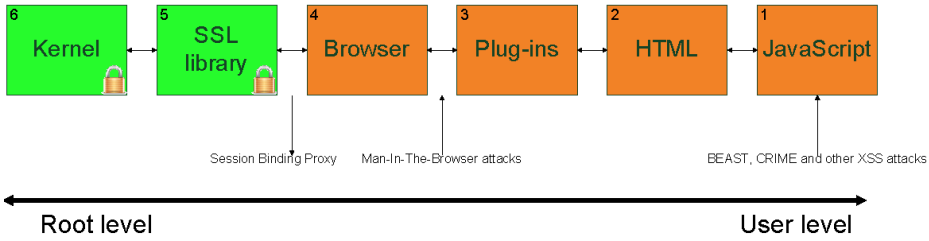

Fig. 7. Attacker model.

Suppose an adversary can execute JavaScript code (access to level 1 or 2 in Fig. 7). He can use this to craft an XSS attack and steal a cookie from a user. With SBP in place, the attacker can still steal the cookies, but he will not be able to take over the user's session, because he cannot decrypt or re-encrypt the cookie. An alternative prevention method against an adversary with access to level 1 or 2 would be to make use of http-only cookies as described in Sect. 2.2.

When the attacker has access to level 3, more advanced attacks can be crafted. A browser plugin has more rights than JavaScript and can read all cookies, even the http-only cookies. A malicious plug-in is able to send sensitive cookies to the adversary. To put it in perspective, a browser plug-in is also able to perform other attacks. A browser plug-in can forge requests while the user is logged in such that it looks like the user did the request. This way, an attacker does not need to steal the session. It is a different kind of attack called man-in-the-browser [13] and therefore this kind of attack is out of the scope for this paper. A browser plug-in can also view the user credentials while logging in. Just capturing the username and password can be sufficient to take over the entire account. However, modern systems migrate to two-factor authentication to make sure that even with the username and password, an adversary is still not able to log on. After a successful login, the session identifier is a crucial credential that should be protected. SBP aims to provide such a protection.

When level 4 is compromised, the adversary has full user level access to a machine. The adversary can access the cookies of the browser directly through the file system. The adversary can capture user input directly which enables phishing attacks. As explained for level 3, with modern authentication techniques like two-factor authentication, the adversary is still not able to do anything with the recovered cookies.

An attacker has to use operating system exploits to gain access to level 5 and 6. Kernel exploits are often fixed within days of discovery making it hard

to gain access to SSL/TLS credentials. Only an adversary with access to the SSL/TLS master key k and the system private key K_p can bypass the SBP system protection.

On the server side, the attacker needs to gain root access to recover the keys needed to hijack the session. Nginx runs under a isolated user account on the server and once again, operating system exploits need to be used in order to get a hold of the necessary information.

4 Validating SBP

To validate our SBP method, we have designed and implemented a prototype. The code of our prototype is open source and available on a public github repository[6]. This prototype is fully functional and released under the same license as Nginx, the royalty free BSD License, which defines very minimalistic distribution restrictions. The prototype was deployed on a widely used and complex application in order to check whether it can actually prevent session hijacking in a real-life context. We chose Blackboard Learn since this application fits our requirements very well.

Blackboard Learn (previously Blackboard Academic Suite) is one of the most popular e-learning systems or Learning Management System (LMS) in higher education worldwide. It is used by thousands of educational institutions spread over numerous countries. One of its main features is to publish the contents of a course on the Internet for students. This is not directly very security sensitive. However, Blackboard Learn also has the ability to keep track of grades for assignments with the intention to derive the final grades from the Blackboard Learn system. Clearly, influencing the grades might be a goal of an attacker. Furthermore, Blackboard Learn has the ability to take online exams completely within the system. Being able to access these exams in advance is obviously another feasible attack goal.

Several serious vulnerabilities have been found in Blackboard Learn. Online24 [12] did a full black box investigation of Blackboard Learn version 8. They found all kinds of feasible attacks. One vulnerability with major impact allows an adversary to insert executable code or send emails with viruses from Blackboard Learn. Because of these flaws in Blackboard Learn, one of the possibilities is that students can elevate their permissions to the permissions of a teacher. This can be done e.g. by session hijacking. A student can insert cross site scripting (XSS) code in an assignment and when the teacher opens this assignment to grade the work, the code is executed in the browser of the teacher. Blackboard Inc., the company behind Blackboard Learn responded with a new version in which all the problems were claimed to be fixed. LaQuSo[7] verified this claim by again testing the Blackboard Learn system version 9.1 SP5 as a black box. LaQuSo concluded [4] that Blackboard Learn blacklisted the previous attacks, but that workarounds for the blacklisting filter were very easily found. Vulnerabilities can be expected to keep popping up until structural security measures have been fully incorporated by Blackboard Learn.

[6] https://github.com/wburgers/Session-Binding-Proxy

[7] The Laboratory for Quality Software (LaQuSo) is a joint activity of Technische Universiteit Eindhoven and Radboud Universiteit Nijmegen.

The structural nature of the revealed vulnerabilities lead the Board of Directors of the Radboud University in Nijmegen to decide in September 2011 not to use Blackboard Learn for any privacy or security sensitive activities until further notice. Hence, Blackboard Learn can not be used for online exams and teachers e.g. have to keep a separate version of the grades they gave to students since the students can edit the Blackboard Learn grades.

Sect. 4.1 first explains the details of a cross site script that steals a session in Blackboard Learn. Then, Sect. 4.2 reports on the results of testing our prototype on the Radboud University Nijmegen Blackboard Learn test server. Finally, Sect. 4.3 evaluates the test result impact for Blackboard Learn.

4.1 Session Stealing in Blackboard Learn

It was still very easy to steal sessions in Blackboard Learn version 9.1 SP5 by executing an XSS attack. The LaQuSo research showed that especially the Discussion board and Blog-Assignments were vulnerable and scripts could be injected in the title/subject field as well as the message field of both modules. When a student hands in a Blog-Assignment or submits something to the Dis-

```
document.write ("<img src=\"http://website.com/bb/?cookie
    =" + document.cookie+ "\">");
```

Fig. 8. bb.js

cussion board and the teacher requests the page, the script in Fig. 8 can get the cookies of the teacher and send them to the attackers website (at website.com).

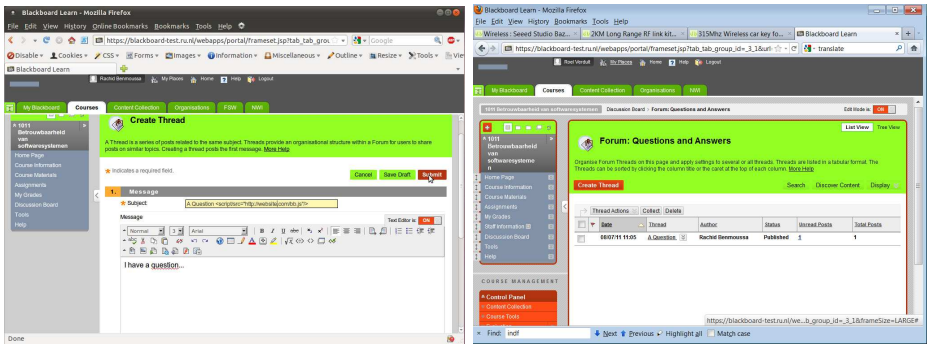

Fig. 9. Student injects JavaScript code — Instructor views malicious post

The screens that invoke the XSS attack are shown. The upper image of Fig. 9 shows the student injecting the attack in the Discussion board and the bottom image of Fig. 9 shows the teacher viewing the discussion thread. The teacher does not even have to open the thread, because the attack is injected in the title. With such an attack, the adversary is able to steal all the necessary cookies from a teacher and hijack the session.

4.2 Applying the SBP prototype

The prototype was tested on the, fully functional, local Blackboard Learn test server of the Radboud University Nijmegen. In order to detect and resolve

incompatibilities with earlier versions this server is used for testing new Blackboard Learn releases and patches before putting them in actual operation. It generally takes several weeks or even a few months before a new release is fully operational. The Blackboard Learn test environment runs on several application servers. A load balancer divides the work between the application servers. This load balancer is not the SSL/TLS endpoint, so the SSL connection is still intact after being routed through the load balancer. For every application server, a SBP is included on the same system such that the routing will look like: incoming request—load balancer—SBP —Blackboard Learn server.
We first tested whether it is still possible to steal a session in Blackboard Learn on the standard Blackboard Learn test server. We used two browsers A and B, both having the login page of the Blackboard Learn test server open. We logged in on browser A, copied the cookies from browser A to browser B and refreshed the page on browser B. By copying the cookies, the session was transferred to browser B and we were logged in on both browsers.

Then, we performed the same attack again but first made sure that the installed SBP prototype was used. This was achieved by one simple redirect at the highest level. Again we used two browsers A and B. Both had the login page of the Blackboard Learn test server open, but this time we were connected to the SBP machine. This means that all traffic flowed through the proxy. Again, we logged in on browser A and copied the cookies to browser B. When we refreshed the page on browser B now, we were not logged in. The reason for that is that both browsers had a different SSL session. The cookie that is sent to the back end server will not be valid and the session is not taken over. So, SBP was effectively used to prevent session hijacking in Blackboard Learn.

4.3 Validation evaluation

The Blackboard Learn test server we used did not have the most recent patches by Blackboard Inc., where some cookies are protected from JavaScript access by making them http-only. The browser will then keep the cookies for itself and when requesting access to the cookies by the JavaScript command 'document.cookie', it only returns the cookies that do not have the http-only attribute. This means that XSS attacks based on JavaScript requesting the cookies, will not work. The most important cookie in Blackboard Learn is the s_session_id cookie. The s_session_id cookie is set to http-only in the recent patches. The session cannot be hijacked by XSS attacks because of this property. This does not mean that session stealing becomes impossible. Recent papers have shown that cookies sent over an HTTPS connection can be read using a chosen plaintext attack on SSL/TLS [3]. This recent Blackboard Learn patch is therefore less effective than our SBP approach. In Sect. 5 we will discuss this attack on SSL/TLS. It is important to note that SBP does not prevent XSS attacks or other attacks from happening (proper input validation and output encoding should be in place in order to achieve that), but it does prevent one of their uses. SBP is a general approach in which sessions can no longer be stolen by only obtaining the session ID from the cookie of a client's browser.

The validation has shown that our SBP prototype is fully operational, easily deployed and effective in practice against session stealing via XSS. Even in a production environment with multiple application servers.

4.4 General applicability of SBP

The use of SBP is not limited to e-learn environments only. Many legacy web applications suffer from various XSS vulnerabilities, mainly due the lack of proper maintenance. Without having access to the source-code it is hard to protect them against widely distributed and general applicable XSS exploit scripts. Our contribution provides a setup that does not require any knowledge of the web application that it protects. As long as it is accessible through a secure channel (SSL/TLS) and uses a cookie to store the session credentials, then the SBP is able to fortify the security of its online sessions. Our case study shows the robustness and demonstrates that hijacking a session is only mountable when the clients computer or application server is completely compromised.

5 Related Work

5.1 Session hijacking prevention

There are several other proposals to prevent session hijacking. Johns (2006) [7] proposes a solution where the cookies in which the session ID is kept are sent from a different subdomain. This way the JavaScript code cannot get the cookie, because it does not fall under the same-origin policy, so the cookie is safe. This does not prevent every type of attack though. With browser hijacking or XSS propagation, session cookies can still be obtained by an attacker. Johns uses URL randomization and one-time URLs to prevent these attacks from being executed. He also writes that these methods are not meant as a complete replacement for input and output validation in the application, but it is an extra layer of protection. This sure is a good way of preventing session hijacking, though it is a lot of hassle to implement. Most of the application needs to be rewritten.

Another method is to run a piece of software on the client computer which intercepts the 'Set-Cookie' header before it is sent to the browser. This way the cookies will never be in the browser at all. This method is proposed by Nikiforakis et al. (2011) [10]. Without much overhead this system will prevent JavaScript code from accessing the cookie information. This still relies on the client side. A secure implementation without memory leaks makes this a good solution. As mentioned in Sect. 2.3, this paper is based on the work of Oppliger et al. [11]. They propose to bind the application session to the SSL/TLS session to prevent MITM attacks. To bind the two sessions, they use either a software token (like a client cerficate or a private key) or a hardware token (like a smartcard or dedicated device). This is a safe solution, but it requires the distribution of a pre shared key to the client/user. The same binding idea can be used for session hijacking, but we propose a different binding method.

The only other paper that uses the binding of SSL/TLS Session-Aware User Authentication as a basis is a proposal by Chen et al. [2]. They make use of a two factor authentication method by means of a separate device (3g phone). With this device they bind the SSL/TLS session to the application session. It requires both client and server side changes.

In Fig. 10, an overview is given of the modifications that are required to secure sessions with the various proposed methods.

Protection method	Side	Software patches		System changes		
		browser	application	software token	hardware token	server
SessionSafe [7]	Server	no	yes	no	no	no
SessionShield [10]	Client	yes	no	no	no	no
Session-Aware [11]	Server	no	yes	yes[1]	yes[1]	no
TLS-SA + GAA [2]	Both	no	yes[2]	no	yes	yes[2]
SBP	Server	**no**	**no**	**no**	**no**	**no**

[1] The implementation can work with either a software token or a hardware token

[2] Either the server application needs to be modified or install additional software

Fig. 10. Comparison of patch requirements to prevent session stealing

5.2 Related attack setups

There are also papers that describe attacks on cookies and sessions. As mentioned in Sect. 4.3 there exist attacks like Browser Exploit Against SSL/TLS (BEAST) [3] and CRIME to steal cookies. Both attacks are implemented in JavaScript for speed, but can be run on any user level. BEAST and CRIME use known plaintext attacks to guess the unencrypted cookie that is sent over an encrypted SSL connection. The cookie is guessed character by character. This brute force method allows to guess an entire cookie. Where BEAST works only on certain versions of SSL/TLS, CRIME works for any version. CRIME makes use of the compression in SSL/TLS to guess the cookie. The flaws of compression in combination with encryption are already described in a paper by John Kelsey in 2002 [8]. There is a proof of concept for the CRIME attack. With some modifications it can be used to actually capture cookies even though they have the http-only property. This is just another method to get the cookie from the client. Our proposed method also defends against both attacks as depicted in the attacker model. Even though they can steal the cookie. It is still hard to copy the SSL session. Also the last two years people could download a Firefox add-on that would sniff network traffic and intercept cookies from other users. This extension is called Firesheep. The main focus of the creator of Firesheep was to encourage sites like Facebook and Twitter to always use HTTPS, not just when logging in. Nowadays those sites do use HTTPS for all their traffic and Firesheep is useless. BEAST and CRIME can be used however. Firesheep will also not work on sites that use SBP, because SBP needs HTTPS to work.

6 Conclusions

We have presented a new, general technique to prevent session stealing, SBP. Using a server side reverse proxy the secure communication channel is bound to the user authentication of the session. We validated the approach by implementing SBP and testing it on a test server of a widely used infrastructural application which was vulnerable to session stealing. Using SBP, this application was shown to be effectively protected against the earlier session stealing attacks. We made our prototype implementation available as open source to be used by a broad community in a wide context. This prototype is fully functional and released under the same license as Nginx, the royalty free BSD License, which defines very minimalistic distribution restrictions. We want to improve on our prototype to handle full SSL/TLS connection termination and renegotiation, such that long-living cookies can also be used with SBP. This will make it deployable in all contexts.

Acknowledgements

We thank the anonymous reviewers for useful feedback and for believing this solution has the potential to grow into the universally acceptable security standard solution. This motivates us to continue along this path and make our solution work for every context.

References

1. Willem Burgers. Session proxy, a prevention method for session hijacking in blackboard. bachelor thesis, Institute for Computing and Information Sciences, Radboud University Nijmegen, The Netherlands. *Bachelors Thesis*, July 2012.
2. Chunhua Chen, Chris J. Mitchell, and Shaohua Tang. SSL/TLS session-aware user authentication using a gaa bootstrapped key. In *5th IFIP WG 11.2 international conference on Information security theory and practice: security and privacy of mobile devices in wireless communication (WISTP 2011)*, volume 6633 of *Lecture Notes in Computer Science*, pages 54–68. Springer-Verlag, 2011.
3. Thai Duong and Juliano Rizzo. Here come the XOR Ninjas. White paper, Netifera, May 2011.
4. Marko van Eekelen, Rachid Ben Moussa, Engelbert Hubbers, and Roel Verdult. Blackboard Security Assessment. Technical Report ICIS–R13004, Radboud University Nijmegen, April 2013.
5. Blackboard Inc. Release notes for blackboard learn 9.0 service pack 7 (9.0.692.0). *Behind the Blackboard for System Administrators & Developers*, 2011.
6. Blackboard Inc. Release notes for blackboard learn 9.1 service pack 8 (9.1.82223.0). *Behind the Blackboard for System Administrators & Developers*, 2012.
7. Martin Johns. SessionSafe: Implementing XSS immune session handling. In *11th European Conference on Research in Computer Security (ESORICS 2006)*, volume 4189 of *Lecture Notes in Computer Science*, pages 444–460. Springer-Verlag, 2006.
8. John Kelsey. Compression and information leakage of plaintext. In Joan Daemen and Vincent Rijmen, editors, *9th Fast Software Encryption (FSE 2002)*, volume 2365 of *Lecture Notes in Computer Science*, pages 95–102. Springer-Verlag, 2002.
9. HyungJun Kim. Security and Vulnerability of SCADA Systems over IP-Based Wireless Sensor Networks. *International Journal of Distributed Sensor Networks*, 2012, 2012. Article ID 268478.
10. Nick Nikiforakis, Wannes Meert, Yves Younan, Martin Johns, and Wouter Joosen. SessionShield: Lightweight protection against session hijacking. In *3rd International Symposium Engineering Secure Software and Systems (ESSoS 2011)*, volume 6542 of *Lecture Notes in Computer Science*, pages 87–100. Springer-Verlag, 2011.
11. Rolf Oppliger, Ralf Hauser, and David Basin. SSL/TLS session-aware user authentication – or how to effectively thwart the man-in-the-middle. *Computer Communications*, 29(12):2238–2246, August 2006.
12. Michiel Prins and Jobert Abma. Security research blackboard academic suite. https://www.online24.nl/blackboard-security-research, Online24, 2010.
13. Nattakant Utakrit. Review of browser extensions, a man-in-the-browser phishing techniques targeting bank customers. 2009.