Radboud University Nijmegen

Bachelor Thesis

Session Proxy

A prevention method for session hijacking in Blackboard

Author:
Willem Burgers
s0814830

Supervisor:
Prof.dr. M.C.J.D. van Eekelen

Second supervisor:
drs. ing. R. Verdult

July 4, 2012

Contents

1 Research Plan

1.1 Problem Statement

Blackboard Learn (previously Blackboard Academic Suite) is one of the most popular e-learning systems or Learning Management System (LMS) in higher education worldwide. Its main features are things like keeping track of grades and providing the contents of a course to the student through the Internet. But there are some flaws in the design of Blackboard[1]. It is possible to insert code to be executed or send emails with virussus for example. Because of these flaws, one of the possibilities is that students can elevate their permissions to the permissions of a teacher. This can be done by session hijacking. HTTP is a stateless transmission protocol. To keep track of the pages a person can visit, sessions can be implemented in the application. When a session id is obtained, you have control over the session. A student can insert javascript code in assignment forms that will be executed once the teacher looks at the handed in assignment. This code can steal the session id of the teacher. With the session of a teacher, a student can for example edit grades[2]. Universities can not make use of all the features of Blackboard because of these flaws. Therefore some universities decided to move to the open source e-learning platform Moodle[6][3][4].

1.1.1 Research Question

I want to know how fundamental these flaws in Blackboard are and how to prevent them. The most common flaw in blackboard is cross-site scripting[2]. This means that an attacker can execute code in the browser of a victim, without the victim knowing. Usually this is javascript code. With cross-site scripting someone can steal session id cookies called session hijacking. I want to give a more fundamental solution for session hijacking. Therefore the research question I will pose is:

How to prevent session hijacking in Blackboard?

In order to give an answer to this question, we have to find an answer to the following subquestions:

1. What are the main vulnerabilities in the Blackboard Learn system and how can we prevent them?
2. How to prevent the underlying problem, namely session hijacking, in a structural way?
3. Will the prevention solve security risks in Blackboard?

The answer to subquestion 1 will be a list of vulnerabilities taken from the online24 Blackboard security research paper[1] and the internal research paper by R. Ben Moussa[2]. The answer to subquestion 2 will be in the form of a session hijacking prevention method for Blackboard by comparing different solutions. I also want to implement this proposed solution. After implementing the solution, we also need to check if the problems with Blackboard are solved. This is the answer to subquestion 3.

1.2 Motivation

As stated before, Blackboard is one of the most used Learning Management Systems, but the problem is that there are some flaws in the design. Though Blackboard Inc. provides updates to stop attacks from working, still a lot of issues remain. Implementing a new system can be very expensive and can introduce several new problems. But it is also not very useful when the university can't use all the features the Blackboard Learn system actually provides because it has some serious security flaws. Therefore, we want to propose a fundamental solution that works for Blackboard. In this proposal we want to focus on the security aspects of the solution. If this solution works, it could be used in front of any application to prevent session hijacking, but mainly we want to use it for Blackboard. This security part is also my main personal motivation. I like writing code, and when I do, I always check my code to make sure I didn't make any mistakes that bring security risks with them. That is why my bachelor thesis is about security.

1.3 Theoretical Scope

1.3.1 Vulnerabilities and solutions

Blackboard is implemented as a web based application. It must therefore be defended against all kinds of web based attacks. The attacks discussed in the research paper by Online24 show a lot of web based vulnerabilities that are still possible to exploit in the latest version of Blackboard (version 9.1)[1]. These vulnerabilities can be categorized in several vulnerability types. For all of the vulnerability types mentioned in the paper, there are known solutions. For example cross-site scripting can be prevented by validating the input a user gives to the system[7]. The Open Web Application Security Project (OWASP) also has loads of information on how to prevent many of the other attacks.

1.3.2 Session hijacking

Sessions are necessary to keep track of users and to see which pages they visit and if they are allowed to visit them. HTTP is a stateless protocol, so it does not provide this user tracking. Session are therefore implemented in the application. 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. The client is most vulnerable to steal a session id from. The more fundamental problem that these vulnerabilities are the cause of is session hijacking. The vulnerabilities can be solved, but as long as the session can be stolen (in whatever way), grades and exams on the Blackboard Learn system are not safe. There are some papers on preventing session hijacking[8][9]. There is however another solution that I would like to work out. The idea, explained to me by R. Verdult, is to combine the application session and HTTPS session. With the combination of these two, you can make use of the security of the HTTPS session to secure your application session. It gets a lot harder to take over an application session if you also need to take over the network session. There are already solutions like this. In 2006 Rolf Oppliger et al. wrote a paper about authentication with SSL/TLS. They propose a system with client side certificates to make sure the same client is requesting the page. The client has to prove that he is indeed who he claims to be by sending the certificate to the server[10]. Our method is much like the method of Rolf Oppliger et al., but it has some differences that we will discuss later.

1.4 Strategy

The first step in finding an answer to the research question is explaining which security risks exist in Blackboard. The vulnerabilities of the system are documented by some online research reports. One of them is by Online24[1], but the vulnerabilities are not discussed in much detail because of a Non-Disclosure Agreement. The list of vulnerabilities is a part of the answer to subquestion 1.

In order to protect against these vulnerabilities, there are many known methods. OWASP and other literature documents can provide the information needed to implement a safer web application. These methods will solve the vulnerabilities that exist in the Blackboard system and are the second part of the answer to subquestion 1.

Thought these methods can solve the vulnerabilities in Blackboard, there is another more fundamental way for one of the attacks. By preventing session hijacking, some of these solutions to vulnerabilities will not be necessary, though it is still good practice to sanitize user input. This is because preventing session hijacking does not prevent the cross-site scripting attack itself. Cross-site scripting will still be possible and can only be prevented by proper input sanitizing. I will describe multiple ways of preventing session hijacking and will implement one of these methods. This is the answer to subquestion 2.

With this method implemented, I want to make sure that the Blackboard Learn system vulnerabilities are not exploitable anymore. This will form the answer to subquestion 3.

2 Blackboard vulnerabilities

2.1 vulnerabilities

In a research paper the company Online24 identifies several vulnerabilities in the Blackboard Learn system[1]. In the research they examine Blackboard version 8, but most of the vulnerabilities are not patched in version 9.0 and 9.1 or the vulnerabilities are blacklisted, but still easily exploitable. Online24 categorized all the vulnerabilities in eight types.

- Cross-site scripting
- Insufficient authorization
- Information leakage

- Mail command injection
- Abuse of functionality
- Local file inclusion
- Improper file system permissions
- Cross-site request forgery

Cross-site scripting Cross-site scripting (XSS) attacks form the largest group of vulnerabilities in Blackboard[1]. With XSS it is possible to steal usernames and passwords of all the Blackboard users. You can also change your permissions or user roles for Blackboard. This can be performed by combining XSS with Cross-site Request Forgery.

Insufficient authorization Blackboard does not check whether you have access to view some files. You can for example view a calendar of another person. It gets even more dangerous when you can also edit these files. Like removing appointments from another users calendar.

Information Leakage Blackboard can give information about the database, the network infrastructure and it's own program code. This can be valuable information for an attacker to get information about other vulnerabilities.

Mail command injection With mail command injection, an attacker can make use of the mail server of Blackboard to send spam. The server can get blacklisted for that and it will be a lot of hassle to get it off the blacklist again. Also malware and viruses can be sent via mail.

Abuse of functionality By abuse of functionality some features can be used for another purpose than originally intended. For example turning off the client side WYSIWYG filters in the text editor. This can be used in combination with other attacks.

Local file inclusion With local file inclusion, files of the host operating system can be included to display information inside these files. For example the /etc/passwd file on a unix system can be read (however this depends on the system settings). Or some password file that is necessary for the application to access the database. Getting access to a system gets easier with local file inclusion.

Improper file system permissions A teacher is able to delete files on the filesystem via the Blackboard application. This can take the whole application offline when critical files are removed. It is possible for students to elevate their permissions to teacher, so everyone can remove files.

Cross-site request forgery By editing URLs users can execute commands they should not be able to execute. In combination with social engineering, a user can let a teacher click a link. When the URL is requested, the teacher can grant the user access as a student assistant or co teacher of a course.

2.2 Prevention methods for Blackboard vulnerabilities

The problem in Blackboard is that the user input is not properly checked. Every textbox or form field should be scanned for unwanted input to make sure no code is executed that should not be executed. Even URLs can be used to access parts of Blackboard that should be off limits to most users. The most common and risky flaw in the Blackboard system is the lack of input validation on any form which will make XSS attacks possible. Blackboard made it possible to insert HTML tags in these forms so the layout of an exercise for example can be created with HTML. In the research paper by R. Ben Moussa,(2011), there is a proposal to separate the code of the layout editors. One for the students that cannot insert HTML and one for the teacher that will be able to use HTML. This prevents the students from using javascript in the forms they fill in (like assignments). Another proposal in this paper is to always check permissions. This should be done for every page. In the current situation, Blackboard hides the parts or links that should not be accessed from the user. But when they are requested anyway, Blackboard will show this information. There are no further checks to see if the user requesting this page is actually entitled to do so. Authentication verification should be done for every page to make sure valuable or confidential information will not leak. These suggestions will rule out most of the vulnerabilities in the Blackboard system. The rest of the vulnerabilities should be easy to fix and can be prevented with some minor changes to the code. However, these solutions require modification of the source code of Blackboard. So this means that all the universities depend on Blackboard Inc. to patch these vulnerabilities. Blackboard Inc. has done something about the problem, but has not yet provided a fundamental solution for the problem. They have blacklisted some possibilities for inserting some javascript code. But there are still many ways to be able to execute scripts.

3 Session Hijacking

Blackboard Inc. is not providing any solution for now. So it is necessary to look for other ways of securing the system. As said before, the most common vulnerability in Blackboard is XSS. This enables a user to execute scripts on other machines. One of the uses of XSS is stealing session information stored in cookies on the client machine. When the session id is copied, the whole session can be taken by setting the cookie in the browser of the attacker to the same value. The server then thinks you are the person it gave that session id to. In this way the session of a teacher can be stolen by a student. The student will get the permissions of the teacher. For Blackboard to be more secure, another level of authentication needs to be added.

3.1 Session Hijacking Prevention

3.1.1 SSL/TLS session-aware authentication

As stated in the Theoretical Scope, there are already some methods to prevent session hijacking available. One of those methods is made by Rolf Oppliger et al.(2006) It combines the use of SSL/TLS sessions with the application session by using client certificates. The coupling of the SSL/TLS session and the application session provides a failsafe. If the session id is used by another HTTPS connection, you know there is something wrong and the server can ask for reauthentication[10]. The SSL/TLS client certificate and application session combination is kept inside the application. This also requires the application to be rewritten. This proposal is not only built on the session id of the SSL connection. It uses client certificates to authenticate the user. SSL/TLS session renegotiation has no influence this way. We didn't want to use client certificates, because this also requires a lot more work to install and users usually don't want that, our method is pretty much pluggable.

3.1.2 SessionProxy

I want to propose a method that combines SSL/TLS session-aware authentication with a reverse proxy. It is much like the method Rolf Oppliger et al. proposed. Instead of implementing it inside the application, we want to implement this inside a simple reverse proxy. This proxy relays the requests to the backend 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 internally and not from the Internet. (See figure 1 and 2). To handle the sessions, the reverse proxy needs to be extended with functionality to read the requests and responses and manage the SSL/TLS session and application session. The proxy stores the SSL/TLS session and application session combination in it's own memory. The public key of the client should be enough to authenticate the client. You encrypt the application data sent to the client with this key and the incoming requests are encrypted and can only be decrypted with the public key of the client. If you intercept the "set-cookie" header sent by the application server, you can also read the application session status. When a request comes in, the cookie header must be read and checked against the key value pair that is stored in the proxy. If the public key, session id pair of the request does not match one in the local database of pairs, the session is invalid. To invalidate the session on the application server, the invalid request can be sent to the server without the cookie header. The server will then return the login page. In practice, the client's public key cannot be requested from the SSL suite that implements and handles the SSL connection. The suite does provide an SSL session id value. This value is a unique identifier of an SSL session, but it does not identify a client. When an SSL session is renegotiated, the SSL session id changes.

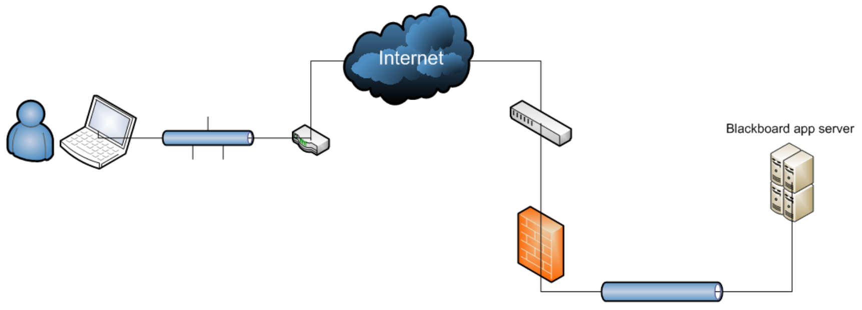

Figure 1: This is a simple model of the current setup for Blackboard.

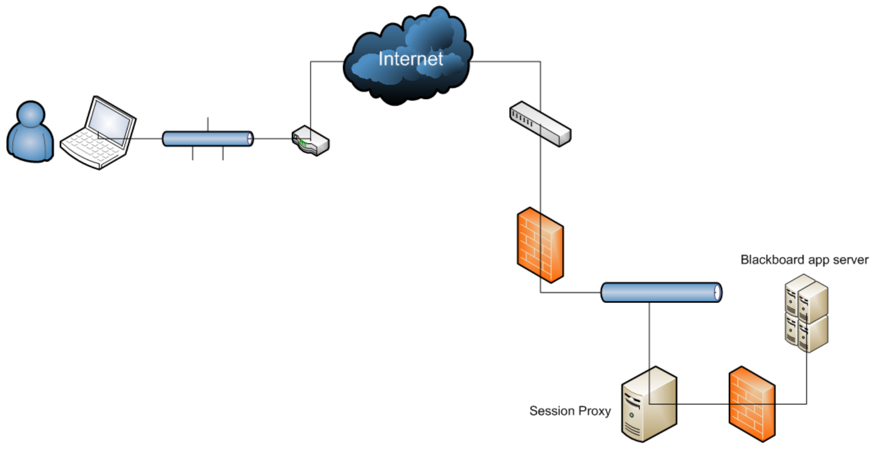

Figure 2: This is a simple model of the setup for Blackboard with Session Proxy.

Protocol message sequence The first thing that the Session Proxy server does when it gets a request, is redirect the user to the HTTPS port if the user did not request that already. When the user's browser does not provide a session cookie, the Session Proxy can simply replay the request to the application server. Any request without a session cookie will result in a redirect to the login page, so this can't go wrong. When the user logs in, the application server will send a "Set-Cookie" header. This header is intercepted (but still sent back to the user) by the Session Proxy and the Session Proxy will store the cookie data along with the SSL session id of the client's connection. Now with each request by the user, the browser of the user will send the cookie along. When a request is received with cookie data, the Session Proxy will check to see if the SSL session id of the request is the same as the one that received the cookie from the application server. So the cookie is the key and the SSL session id is the value in the key-value pair. If the SSL session id's match, the request can be replayed to the application server along with the session cookie. If they don't match, the cookie is removed from the headers of the request and then the request is sent to the application server. Because the application server cannot see the cookie, it will automatically redirect to the login page as said before. The only thing that can cause problems now is SSL renegotiation. Since Internet Explorer 5, the SSL session id is mostly considered as a non persitent value and cannot be relied on for client authentication/identification[5]. Most modern browsers however only renegotiate an SSL connection every few hours. So this will just result in an application session invalidation for now.

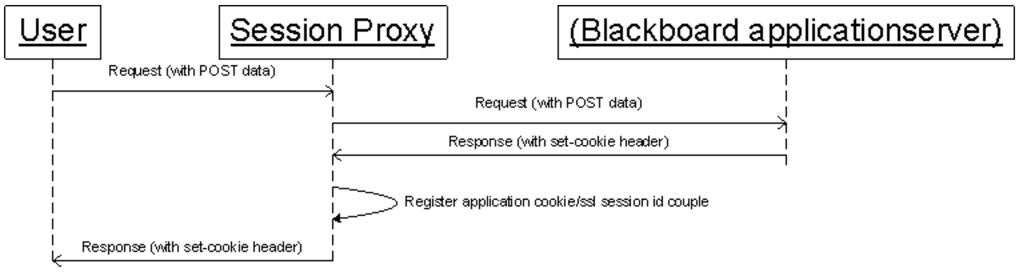
User
Session Proxy
(Blackboard applicationserver)
Request (with POST data)
Request (with POST data)
Response (with set-cookie header)
Register application cookie/ssl session id couple
Response (with set-cookie header)

Figure 3: The message sequence for requests before authentication (including the authentication request with cookie/SSL session id registration).

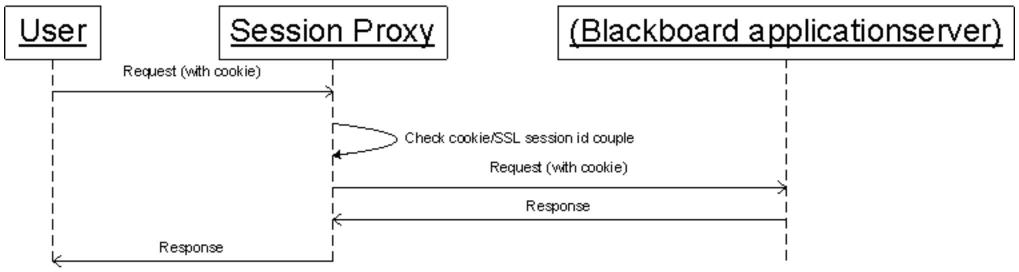

Figure 4: Requests while authenticated.

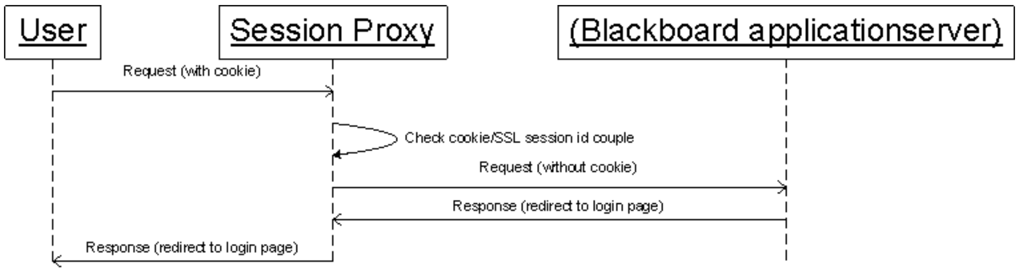
User
Session Proxy
(Blackboard applicationserver)
Request (with cookie)
Check cookie/SSL session id couple
Request (without cookie)
Response (redirect to login page)
Response (redirect to login page)

Figure 5: An invalidation of a session. The cookie/SSL session id couple will be removed from local memory

3.1.3 Related work

There are several other proposals to prevent session hijacking. In the paper by Martin Johns (2006), he 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 doesn't 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). Without much overhead this system will prevent javascript code from accessing the cookie information. This still relies on the client and a secure implementation of this piece of software, without memory leaks, will indeed do the trick.

	Server/Client side	Application modification
Session Proxy	Server side	Pluggable (only configure), with some firewall modification.
SessionSafe (Johns)	Server side	Requires network structure and application modification.
SessionShield (Nikiforakis)	Client side	Pluggable
SSL/TLS Session-Aware User Authentication	Server side	Requires client certificates.

3.2 Implementation

We wanted to implement a proof of concept for Session Proxy 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. Because Session Proxy relies heavily on the reverse proxy, I chose to implement it as a module for Nginx, so the framework can be used to handle the requests and only the application logic of the cookies and SSL session id data should be implemented. This was harder than I thought however. After learning about module development for Nginx I took an existing module with the same basic setup. It gets a request, then does something and then sends the request to the backend server. This is exactly what I want to do, so this module was a great example. However, the module does not work as it should. This has to do with the Nginx framework changing and not being too well documented. I have spent a lot of time to get the module to work by looking through the code of the module, as well as the framework, but I wasn't able to get it working correctly. I have made a different implementation based on PHP. Because PHP is not the most secure language, we chose not to test this version on the blackboard servers. I used an old application to test the Session Proxy system. The idea of using the SSL session id in combination with the application session id is not Blackboard specific and should work for any application.
To test whether the Session Proxy system will actually prevent session hijacking, we have made a setup where an application server is reachable the normal way, as well as through the Session Proxy reverse proxy. So it is like a combination of the current setup and the new setup. Both servers should be reachable. With this setup, we can show that an XSS attack to steal the session id will work on the current application server, but when you try to set the cookie while connecting to the Session Proxy, your session will be invalidated. Below is a model of the test setup.

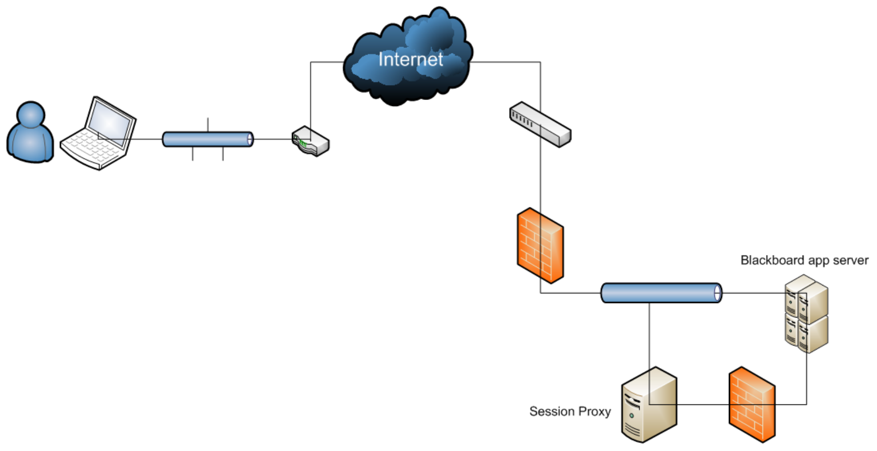

Figure 6: This is a simple model of the test setup for Blackboard with Session Proxy.

We plan to finish the Nginx module in the future, because the PHP version is not very well written. It might still need some bugfixes like urlencoding for spaces in filenames. Once the Nginx module is finished, this thesis will be rewritten in the form of a paper for a conference.

3.3 Attacker model

When defining this new system to prevent a certain attack, we need to keep the security of this application in mind as well. We need to consider what can go wrong if an attacker targets the Session Proxy server or application. The application runs on any Unix operating system. The only open ports should be port 80, 443 and 22 for HTTP, HTTPS and SSH access respectively. With a simple rewrite rule all HTTP connections will be rerouted to the HTTPS port and thus will activate an SSL/TLS secure channel. There is not much that can go wrong with those two ports other than denial of service attacks. When the server is secured with a proper password for the user(s) and the root user, it will also be hard to gain access to the SSH terminal. When you run the application under root and make sure the users have no root privileges, the configuration and data of the application cannot be accessed by other users. An attacker should use operating system exploits to gain root access. It is very hard to find such exploits. Furthermore, the Nginx framework has implemented substitutions for memory allocation functions in C, which implement the secure versions of the standard C functions. By using these framework functions, buffer overflow mistakes are very hard to make by the programmer. All these measures combined make it an almost unbreakable setup. We do not recommend the use of the PHP version of the Session Proxy system for production environments though. Let's say there is a way to take over the Session Proxy server or application. An attacker could then compromise the cookie information stored in memory by the application.

With access to the cookie information, the attacker can modify the data and take over the cookie. But this is the problem in the application now anyway. This proposal does not have any other disadvantages, so the only thing that can go wrong is a session hijack if this proposed Session Proxy fails to work. Therefore Session Proxy only adds an extra layer of security and only brings us back to the current state if the system is compromised.

4 Conclusion

When connecting to an application through the Session Proxy system, you are still able to use the application the normal way. As stated before, Session Proxy does not fix any cross-site scripting vulnerabilities. When trying to hijack a session through XSS however, you are redirected to the login page. This means that if you use Session Proxy to reach Blackboard, the session of a teacher cannot be taken over by a student. The student will still be able to execute other XSS attacks, but trying to edit your application session cookie with the information obtained via XSS from a teacher will not work anymore. Session hijacking is one of the most common uses for XSS, so this is a very useful extra layer of security without rewriting the application itself.

References to webpages and other papers

[1] Online24 Blackboard security research paper. `https://www.online24.nl/blog/blackboard-security-research-paper-has-been-released/`

[2] R. Ben Moussa(2011). Blackboard Security, *Internal research paper on the security of Blackboard at the Radboud University Nijmegen.*

[3] `http://west.wwu.edu/atus/blackboard/assets/Bb%20Satisfaction%20Report%202011.pdf` Found 23-02-2012 using Wikipedia `http://en.wikipedia.org/wiki/Blackboard_Learning_System`

[4] `http://www.montanakaimin.com/mobile/news/blackboard-no-match-for-moodle-1.1596737` Found 23-02-2012 using Wikipedia `http://en.wikipedia.org/wiki/Blackboard_Learning_System`

[5] `http://support.microsoft.com/kb/265369` Microsoft(2007). Found 10-06-2012 when searching for SSL session id persistence.

References to academic literature and articles

[6] Alan Lawler(2011). LMS transitioning to Moodle: A surprising case of successful, emergent change management, *Australasian Journal of Educational Technology*, 27(7), 1111-1123.

[7] T. Pietraszek1 and C. Vanden Berghe (2006). Defending Against Injection Attacks Through Context-Sensitive String Evaluation, *Lecture Notes in Computer Science*, 3858, 124-145.

[8] Martin Johns(2006). SessionSafe: Implementing XSS Immune Session Handling, *Lecture Notes in Computer Science* 4189, 444-460.

[9] Nick Nikiforakis et al.(2011). SessionShield: Lightweight Protection Against Session Hijacking, *International Symposium on Engineering Secure Software and Systems.*

[10] Rolf Oppliger et al.(2006). SSL/TLS Session-Aware User Authentication–Or How to Effectively Thwart the Man-in-the-Middle, *COMPUTER COMMUNICATIONS* 29(12), 2238-2246.

A Source code

A.1 Session Proxy index.php

```
<?php
  require('simpleSHM.php');
  include('config.php');
  $shm = new SimpleSHM(SHMID);
  $url = "https://".BACKEND_HOST;
  if(isset($argv[1])) {
    $url.=$argv[1];
  } else {
  isset($_SERVER['REDIRECT_URL'])?$url.=$_SERVER['REDIRECT_URL
      ']:$url.="";
  }
  if(!empty($_GET))
  {
    $url.="?";
    $number = 0;
    foreach ($_GET as $name => $value)
    {
      ($number>0)?$url.="&":$url.="";
      $url.=$name."=".urlencode($value);
      $number++;
    }
  }
  $ch = curl_init();
  if(!empty($_POST))
  {
    curl_setopt($ch, CURLOPT_POST, 1);
    $postFields=array();
    foreach ($_POST as $name => $value)
    {
      $postFields[$name]=$value;
    }
    curl_setopt($ch, CURLOPT_POSTFIELDS, $postFields);
  }
  if(!empty($_COOKIE)) {
    $cookie=array();
    foreach($_COOKIE as $key=>$value){
      if(in_array($key,$sessioncookie)){
        if(checkpair($value))
          $cookie[]="{$key}={$value}";
      }
      else{
        $cookie[]="{$key}={$value}";
      }
    }
    $cookie=implode('; ', $cookie);
    curl_setopt($ch, CURLOPT_COOKIE, $cookie);
  }
  if(isset($_SERVER['REMOTE_ADDR']))
    curl_setopt($ch, CURLOPT_HTTPHEADER, array('X-Forwarded-For:
        '.$_SERVER['REMOTE_ADDR']));
  curl_setopt($ch, CURLOPT_SSL_VERIFYPEER, 0);
  curl_setopt($ch, CURLOPT_URL, $url);
  curl_setopt($ch, CURLOPT_HEADERFUNCTION, 'read_header');
  curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
  if(($response = curl_exec($ch)) === FALSE) {
    echo curl_error($ch);
  }
  curl_close($ch);
  $response = preg_replace(BACKEND_HOST, 'willemburgers.nl',
      $response);
/*  $filename = "/var/www/logs/log.txt";
  $fh = fopen($filename, 'w') or die ("can't open file");
  fwrite($fh, $response);
  fclose($fh);*/
  echo $response;

function read_header($ch, $string)
{
  $length = strlen($string);
  $location="";
  $cookiearr = array();
  if(!strncmp($string, "Location:", 9))
  {
    $location = trim(substr($string, 9, -1));
  }
  if(!strncmp($string, "Set-Cookie:", 11))
  {
    $cookiestr = trim(substr($string, 11, -1));
    $cookie = explode(';', $cookiestr);
    $cookie = explode('=', $cookie[0]);
    $cookiename = trim(array_shift($cookie));
    $cookiearr[$cookiename] = trim(implode('=', $cookie));
  }
  $shm = new SimpleSHM(SHMID);
  $storedarray = json_decode($shm->read(),true);
  $SSL_SESSION_ID = $_SERVER['SSL_SESSION_ID'];
  foreach($cookiearr as $key=>$value)
  {
    header("Set-Cookie: ".$key."=".$value);
    $storedarray[$SSL_SESSION_ID]=$value;
  }
  $shm->write(json_encode($storedarray));
  if(!empty($location))
    header("location: ".$location);
  return $length;
}

function checkpair($cookievalue){
  $shm = new SimpleSHM(SHMID);
  $storedarray = json_decode($shm->read(),true);
  if(array_key_exists($_SERVER['SSL_SESSION_ID'],$storedarray))
    if($storedarray[$_SERVER['SSL_SESSION_ID']]==$cookievalue)
      return true;
  return false;
}
?>
```

A.2 Config.php

```
<?php
```

```
  define("BACKEND_HOST", "192.168.1.1");

  global $sessioncookie;
  $sessioncookie = array("PHPSESSID",);

  define("SHMID",900);
?>
```

A.3 Shared Memory Class

```
<?php

/**
* SimpleSHM
*
* A simple and small abstraction layer for Shared Memory
    manipulation using PHP
*
* @author Klaus Silveira <contact@klaussilveira.com>
* @package simpleshm
* @license http://www.opensource.org/licenses/bsd-license.php
    BSD License
* @version 0.1
*/
class SimpleSHM
{
    /**
* Holds the system id for the shared memory block
*
* @var int
* @access protected
*/
    protected $id;

    /**
* Holds the shared memory block id returned by shmop_open
*
* @var int
* @access protected
*/
    protected $shmid;

    /**
* Holds the default permission (octal) that will be used in
    created memory blocks
*
* @var int
* @access protected
*/
    protected $perms = 0644;


    /**
* Shared memory block instantiation
*
* In the constructor we'll check if the block we're going to
    manipulate
```

```
* already exists or needs to be created. If it exists, let's
    open it.
*
* @access public
* @param string $id (optional) ID of the shared memory block you
     want to manipulate
*/
    public function __construct($id = null)
    {
        if($id === null) {
            $this->id = $this->generateID();
        } else {
            $this->id = $id;

            if($this->exists($this->id)) {
                $this->shmid = shmop_open($this->id, "w", 0, 0);
            }
        }
    }

    /**
* Generates a random ID for a shared memory block
*
* @access protected
* @return int Randomly generated ID, between 1 and 65535
*/
    protected function generateID()
    {
        $id = mt_rand(1, 65535);
        return $id;
    }

    /**
* Checks if a shared memory block with the provided id exists or
     not
*
* In order to check for shared memory existance, we have to open
     it with
* reading access. If it doesn't exist, warnings will be cast,
    therefore we
* suppress those with the @ operator.
*
* @access public
* @param string $id ID of the shared memory block you want to
    check
* @return boolean True if the block exists, false if it doesn't
*/
    public function exists($id)
    {
        $status = @shmop_open($id, "a", 0, 0);

        return $status;
    }

    /**
* Writes on a shared memory block
*
```

```
* First we check for the block existance, and if it doesn't, we'
    ll create it. Now, if the
* block already exists, we need to delete it and create it again
     with a new byte allocation that
* matches the size of the data that we want to write there. We
    mark for deletion, close the semaphore
* and create it again.
*
* @access public
* @param string $data The data that you wan't to write into the
    shared memory block
*/
    public function write($data)
    {
        $size = mb_strlen($data, 'UTF-8');

        if($this->exists($this->id)) {
            shmop_delete($this->shmid);
            shmop_close($this->shmid);
            $this->shmid = shmop_open($this->id, "c", $this->
                perms, $size);
            shmop_write($this->shmid, $data, 0);
        } else {
            $this->shmid = shmop_open($this->id, "c", $this->
                perms, $size);
            shmop_write($this->shmid, $data, 0);
        }
    }

    /**
* Reads from a shared memory block
*
* @access public
* @return string The data read from the shared memory block
*/
    public function read()
    {
        $size = shmop_size($this->shmid);
        $data = shmop_read($this->shmid, 0, $size);

        return $data;
    }

    /**
* Mark a shared memory block for deletion
*
* @access public
*/
    public function delete()
    {
        shmop_delete($this->shmid);
    }

    /**
* Gets the current shared memory block id
*
* @access public
```

```
*/
    public function getId()
    {
        return $this->id;
    }

    /**
* Gets the current shared memory block permissions
*
* @access public
*/
    public function getPermissions()
    {
        return $this->perms;
    }

    /**
* Sets the default permission (octal) that will be used in
    created memory blocks
*
* @access public
* @param string $perms Permissions, in octal form
*/
    public function setPermissions($perms)
    {
        $this->perms = $perms;
    }

    /**
* Closes the shared memory block and stops manipulation
*
* @access public
*/
    public function __destruct()
    {
        shmop_close($this->shmid);
    }
}
```

A.4 .htaccess

```
RewriteEngine on
RewriteCond %{HTTPS} !=on
RewriteRule .* https://%{SERVER_NAME}%{REQUEST_URI} [R,L]
RewriteRule ^ index.php [L]
```