Web Application Security Assessment

Jack Wilson

White Paper

Abertay University

BSc Ethical Hacking

2016/2017

Note that Information contained in this document is for educational purposes.

ABSTRACT

Almost every modern web page runs as a web application, where dynamic content is delivered from a database or through JavaScript running in the background. Comparatively, older web pages when the internet was first created typically had static pages where no user input was necessary. Dynamic content has brought an entirely new set of challenges, including security.

This paper follows the website testing methodology from the Web Application Hackers Handbook, to test all the relevant areas of the website and find as many vulnerabilities as possible.

Administrative access was achieved almost instantly, through poor authentication mechanisms. However, more vulnerabilities such as SQL injection, cross-site scripting and bad input validation were also discovered.

TABLE OF CONTENTS

INTRODUCTION

This paper aims to conduct a web application security assessment on a test environment to find the potential vulnerabilities posed to the company from a malicious attacker. The only information provided is the URL for the server and login credentials (username and password) for a student user.

A web application is a program that receives some form of information from the internet (e.g. search engine results or media). The application may work on both mobile and desktop clients through a browser or from an application installed on the client's system. (PC Mag, [no date]).

The problems facing a web designer is moderating how much of the web application is publicly accessible, defining how much access users have and properly validating users input. The investigation will follow parts of the *Web Application Hackers Handbook* testing methodology. This involves:

- Mapping the application (finding public and hidden resources).
- Analysis of the application (identifying functionality and data entry points).
- Testing client-side control (investigating how data is sent and how input is validated).
- Testing access control.
- Test for logic flaws (how application handles incomplete input).
- Input-based vulnerabilities (testing for SQL injection, cross-site scripting and OS command injection).
- Session management (checking if cookies are similar between different users and for cross-site request forgery).
- Investigating authentication (password quality, username enumeration, account recovery functionality, predictability of auto-generated credentials).

There are a few main vulnerabilities that are commonly overlooked by web designers:

SQL Injection

SQL injection is an attack where the user can execute SQL commands that can read, write, modify and delete data through, for example, an input box in a HTML form. (OWASP, 2016).

A common method of injection is through username and password fields in login pages. If the user's input is improperly sanitised (e.g. excluding special characters) then they may be able to modify the command being executed to gain unauthenticated access to the database running behind the website and read customer information.

Cross-Site Scripting

Cross-Site Scripting (XSS) is an attack that allows for code injection into a browser session. The code is most commonly written in JavaScript but can also be written in HTML, Flash, or any other browser-based languages.

The vulnerability allows for execution of the code server-side which can result in the theft of cookies (for user’s sessions to be hijacked) or redirection to another website (e.g. a phishing website made to look identical to the intended website for credential theft).

There are two types of XSS attack: Non-persistent and persistent. A non-persistent attack would use, for example, a malicious link and requires user interaction to execute. A persistent attack would be executed once by the attacker and remain on the server, executing every time the page is loaded, until removed by the website administrator. (The Web Application Security Consortium, 2011).

1. PROCEDURE AND RESULTS

Following the Web Application Hackers Handbook methodology (MDSec, [no date]), the first area of testing was to find public-facing pages and pages that were publicly available, but not necessarily advertised as existing.

1.1 MAPPING

1.1.1 Robots.txt

This file can be found by automated searching tools (such as Google) and it tells the tools what web pages to filter out of search results. (Robots.txt, [no date]). However, the file is still human-readable. The full file read:

*User-agent: **

Disallow: /1501838/CLVUQZELLLYV/schema.sql

Robots.txt disallowed schema.sql which was schema for the database containing the database name, table names and field names. This information allowed for easier SQL injection later on. The entire database schema file can be found in *Appendix 1*.

1.1.2 DirBuster

The next area to assess was the public facing and hidden directories on the web server. This was achieved using a tool called DirBuster. This tool used a custom wordlist (Searchcode, [no date]). of common file names and compared that to the files on the server. (OWASP, [no date]). For example, if a HTTP status code for a web page was 200, then DirBuster would know the directory exists. However, if the status code was 404, the program would know the file or folder did not exist. (Rest API Tutorials, [no date]).

The results found the following pages:

- /index.php
    - Public-facing home page for the server.
- /viewresult.php
    - Public-facing student login page.
- /admin.php
    - Public-facing administrator/lecturer login page.
- /contact.php
    - Public-facing contact form.

- /robots.txt
    - Robots file described in *Section 1.1.1*.
- /photos.html
    - Unused page left from website template. No usable information, input boxes or file upload buttons.
- /blog.html
    - Unused page left from website template. No usable information, input boxes or file upload buttons.
- /images
    - A directory listing of images used throughout the website.
- /images2
    - A directory listing of images used throughout the website.
- /index.htm
    - An unused JavaScript datepicker. No data is sent anywhere so it is not exploitable.
- /new.php
    - An unused JavaScript close button. No data is sent anywhere so it is not exploitable.
- /course.php
    - A page to view courses the lecturer was enrolled in, hidden behind a lecturer login.
- /result.php
    - A page to view student details and change password, hidden behind a student login.
- /courseinsert.php
    - A page that allows for courses to be added to the database without authentication.

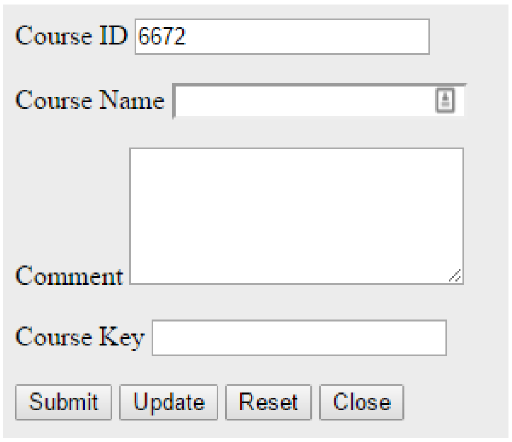


Figure 1: Course Insert Page

The course insert page would load with the "Course ID" field pre-populated with a primary key in a read-only text box. Filling in the text boxes with the relevant information and pressing the submit button would add the record to the database.

- /dashboard.php
    - This page allowed unauthenticated access to the administrator portal and displayed links to the various administrative areas shown in *Figure 2.* All of the links redirected back to the admin login page so that no changes could be made unauthenticated apart from "Inbox" which allowed an unauthenticated user to access, view and delete all contact requests.

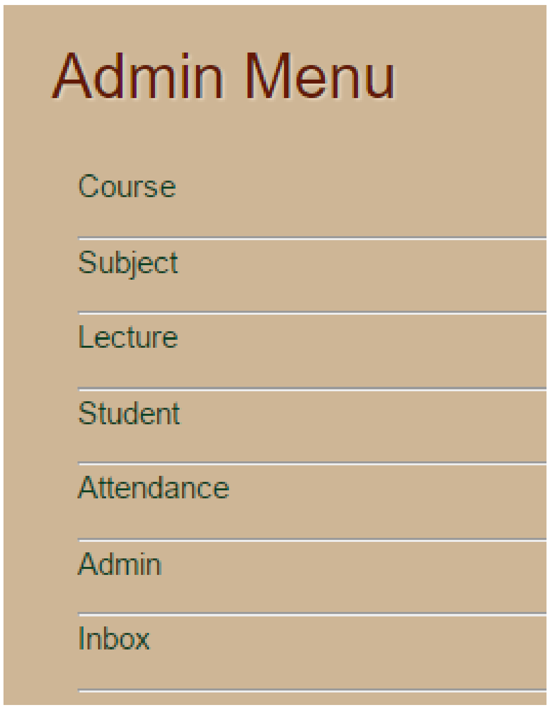
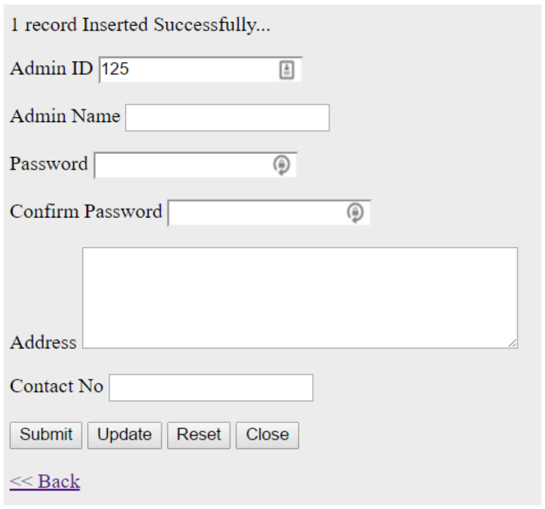


Figure 2: Admin Dashboard

- /addadmin.php
    - A page that allowed unauthenticated access to add administrative users. Having administrative access is essentially game over, however, there are still other vulnerabilities to be discovered. The add administrator page can be viewed in *Figure 3.*


1 record Inserted Successfully...

Admin ID	125
Admin Name	
Password	
Confirm Password	
Address	
Contact No	

Figure 3: Add Administrator Page

1.1.3 View Source

The next stage was to investigate the page source. The landing page for the web server had a comment in the HTML source, on the very first line, that contained the root user's password for the MySQL database, as shown in *Figure 4:*

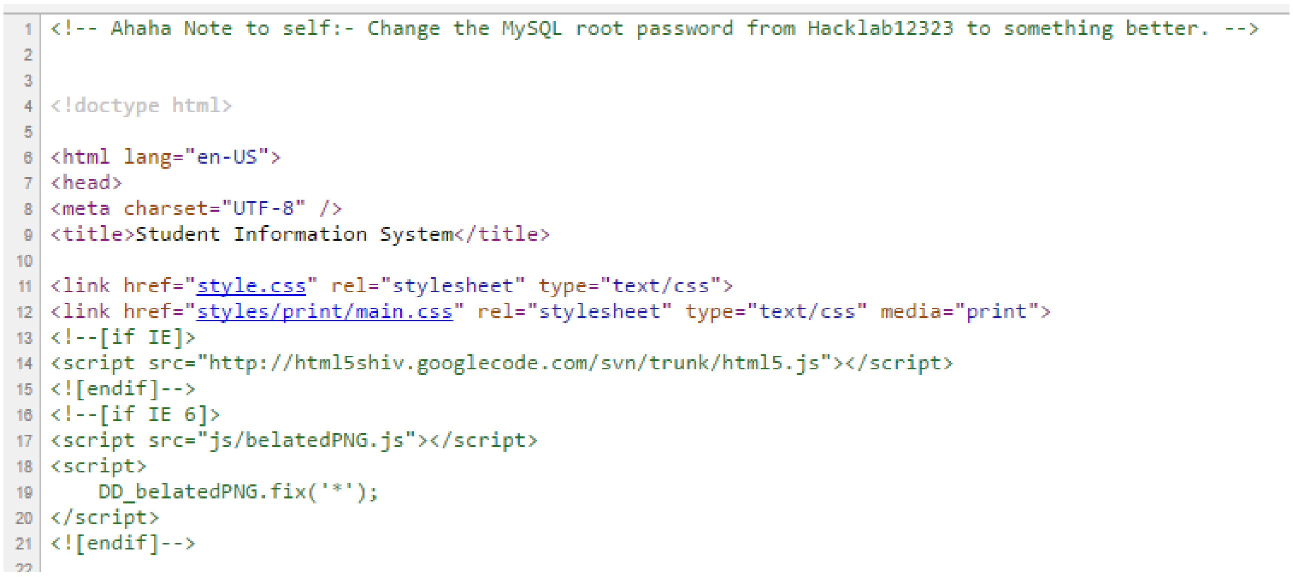
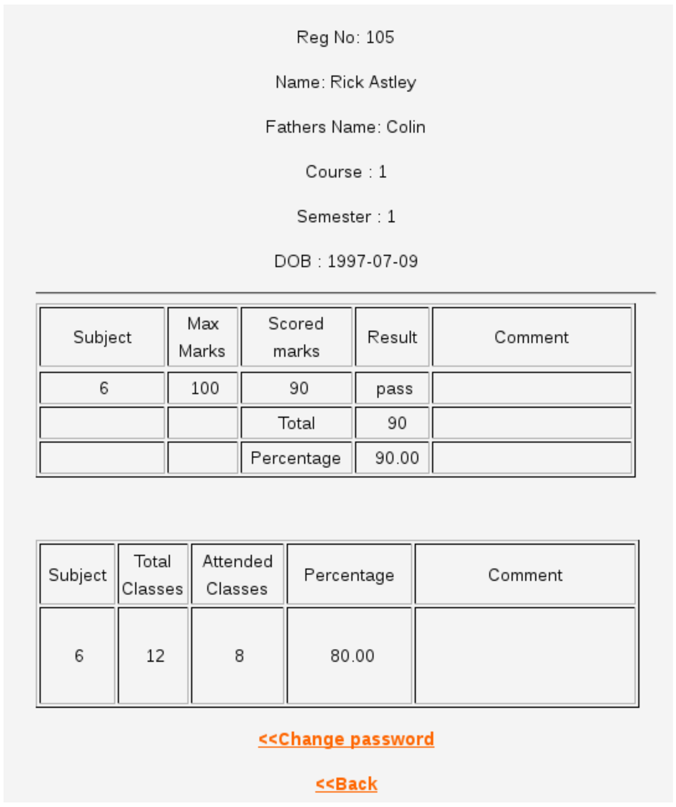

```
<!-- Ahaha Note to self:- Change the MySQL root password from Hacklab12323 to something better. -->


<!doctype html>

<html lang="en-US">
<head>
<meta charset="UTF-8" />
<title>Student Information System</title>

<link href="style.css" rel="stylesheet" type="text/css">
<link href="styles/print/main.css" rel="stylesheet" type="text/css" media="print">
<!--[if IE]>
<script src="http://html5shiv.googlecode.com/svn/trunk/html5.js"></script>
<![endif]-->
<!--[if IE 6]>
<script src="js/belatedPNG.js"></script>
<script>
    DD_belatedPNG.fix('*');
</script>
<![endif]-->
```

Figure 4: HTML Source Code

1.2 ANALYSIS OF FUNCTIONALITY

The application was a website for a college that had three tiers of access: students, lecturers and admins. Students could view their personal details, grades and update their passwords as shown in *Figure 5*.



Reg No: 105

Name: Rick Astley

Fathers Name: Colin

Course : 1

Semester : 1

DOB : 1997-07-09

Subject	Max Marks	Scored marks	Result	Comment
6	100	90	pass	
		Total	90	
		Percentage	90.00	

Subject	Total Classes	Attended Classes	Percentage	Comment
6	12	8	80.00	

<<Change password

<<Back

Figure 5: Student's Portal

Lecturers could view courses and subjects they were registered to teach, view their own personal details such as address and phone number, view student details and track attendance of students as shown in *Figure 6*.

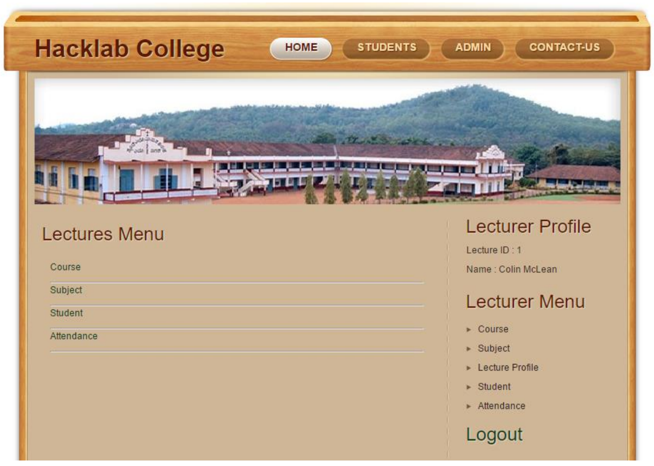
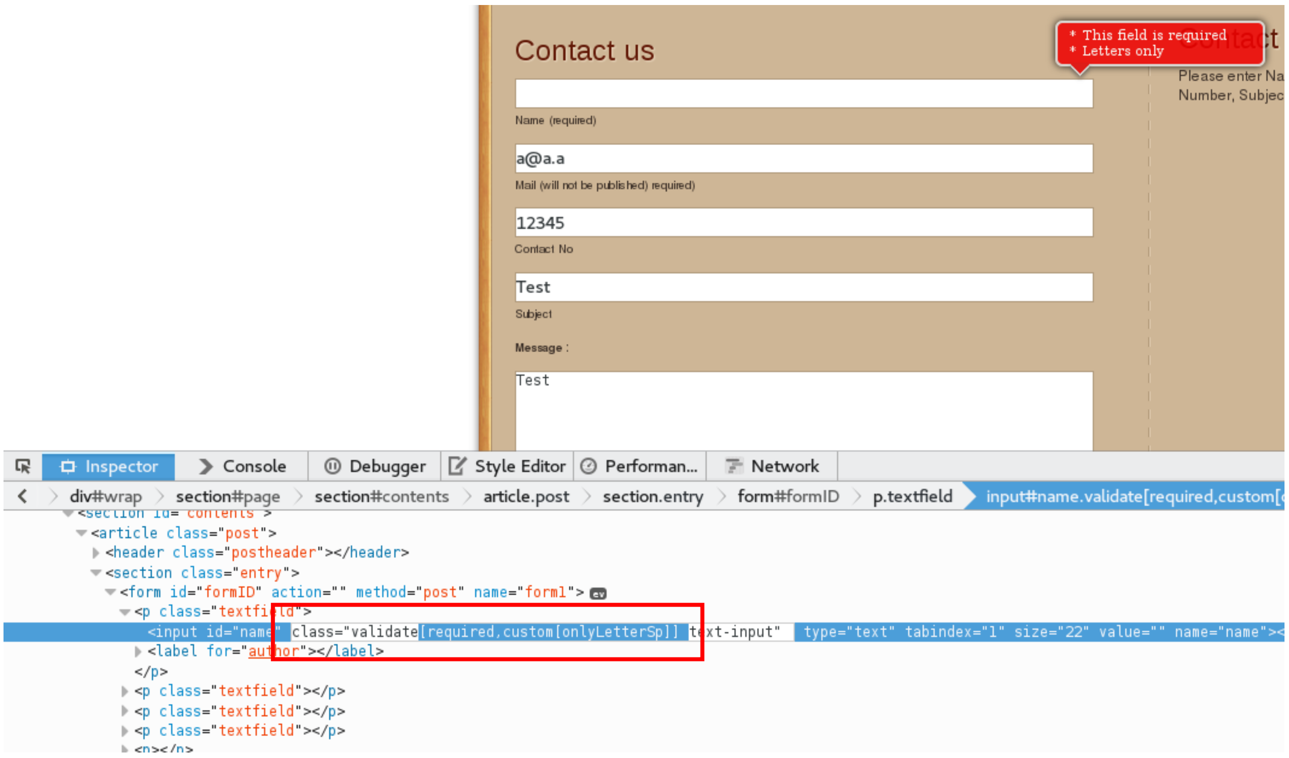


Figure 6: Lecturer's Portal

Administrators had the highest privileges and has full control over the website. As well as the three logins for the various tiers of user, there was also a publicly-facing contact form that required name, email address, phone number, subject and message which that had fairly strict input validation (which could be circumvented). The contact page can be previewed in *Figure 7*.

1.3 TESTING CLIENT-SIDE CONTROL

As mentioned in the previous section, almost all input was validated across the website. The website was validating input on the client (as opposed to the server) which allowed for tampering either by inspecting the text-box element in the browser and removing the validation or by modifying the data being sent using a proxy such as *Burpsuite*. An example of removing the input validation can be viewed in *Figure 7*.



a@a.a
12345
Test
Test

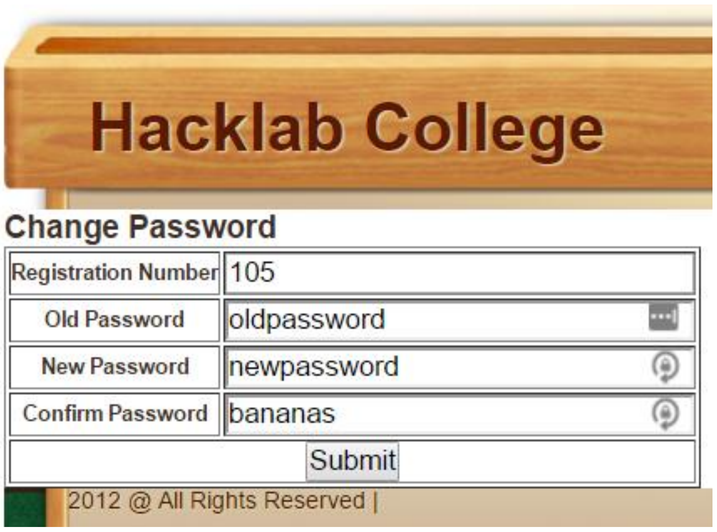
```
<section id="contents">
  <article class="post">
    <header class="postheader"></header>
    <section class="entry">
      <form id="formID" action="" method="post" name="form1">
        <p class="textfield">
          <input id="name" class="validate[required,custom[onlyLetterSp]] text-input" type="text" tabindex="1" size="22" value="" name="name"><
          <label for="author"></label>
        </p>
        <p class="textfield"></p>
        <p class="textfield"></p>
        <p class="textfield"></p>
        <p></p>
```

Figure 7: Using Element Inspector to Remove Input Validation

1.4 TESTING LOGIC FLAWS

1.4.1 Password Update

It was observed that on the page to update a student's password (*/Changepassword.php*) there was no authentication on the password confirmation textbox. This could either be left blank, or an entirely different password from the first password box could be entered and the password would still update. Using *Figure 8* as an example, updating the password with the given credentials would update the password to "newpassword", "bananas" does absolutely nothing.



Registration Number	105
Old Password	oldpassword
New Password	newpassword
Confirm Password	bananas

Figure 8: Example Password Update

1.5 INPUT-BASED VULNERABILITIES

1.5.1 Manual SQL Injection

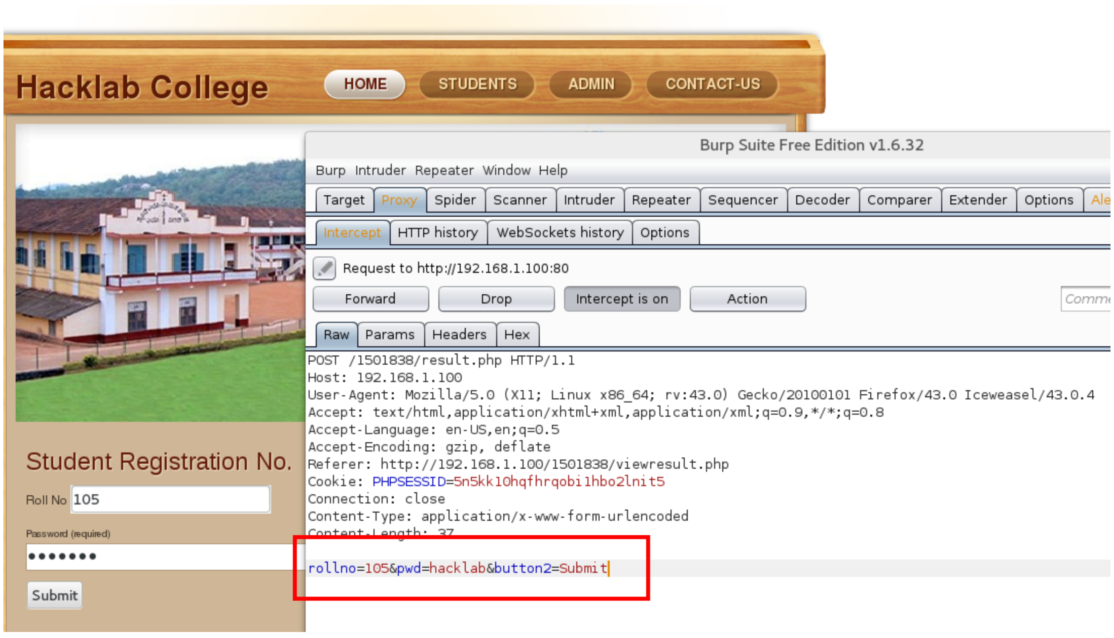
The SQL command below was used in the *Roll No* field on the student login page to dump the administrator ID and password. The first two variables after the *SELECT* statement represent the two fields required for logging in as an administrator, the “null” variables are used as padding since (when the system is not being abused) the database is querying to try and find eleven variables for a student login. This was determined by analysing the database schema found in *Section 1.1.1*.

‘ UNION SELECT adminid, password, null, null, null, null, null, null, null, null, null FROM administrator #

This method could also be used to dump the lecturer and student usernames and passwords, and after learning the username generation method, a WHERE statement could be added to the query to find specific lecturer’s passwords. Realistically, data from any table could be dumped, so long as the syntax for the SQL query was correct.

1.5.2 Automated SQL Injection

The above method was successful for finding specific users, however it was fairly tedious. The entire SQL injection process could be automated using a tool called SQLmap to dump the entire database. This was achieved by logging in to the website using the provided student login credentials and intercepting the HTTP header using Burpsuite, as shown in *Figure 9.*


Hacklab College
HOME
STUDENTS
ADMIN
CONTACT-US
Student Registration No.
Roll No 105
Password (required)
•••••••
Submit
Burp Suite Free Edition v1.6.32
Burp Intruder Repeater Window Help
Target Proxy Spider Scanner Intruder Repeater Sequencer Decoder Comparer Extender Options Ale
Intercept HTTP history WebSockets history Options
Forward
Drop
Intercept is on
Action
Comme
Raw Params Headers Hex

Request to http://192.168.1.100:80

```
POST /1501838/result.php HTTP/1.1
Host: 192.168.1.100
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:43.0) Gecko/20100101 Firefox/43.0 Iceweasel/43.0.4
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://192.168.1.100/1501838/viewresult.php
Cookie: PHPSESSID=5n5kk10hqfhrqobi1hbo2lnit5
Connection: close
Content-Type: application/x-www-form-urlencoded
Content-Length: 37

rollno=105&pwd=hacklab&button2=Submit
```

Figure 9: Intercepting HTTP Traffic with Burpsuite

The next step was to copy the contents of the header to a file (in this case called *header.txt)* and tell SQLmap to read the file using the following command:

SQLmap -r header.txt –dump

The process took some time because of the size of the database, but eventually the entire database was dumped. This provided details on courses, grades, attendance, student details and passwords, lecturer names and passwords. It was noteworthy that students' passwords were stored in plain text and lecturers' passwords were hashed in MD5, however the passwords were very simple and known hashes.

A faster alternative could be to dump specific tables. The schema of the database was known from the schema in *Section 1.1.1*, so the database name and specific table names could be entered into the command to expedite the process. (Teach The Net, 2011). The command used to dump the lecturers table is shown below:

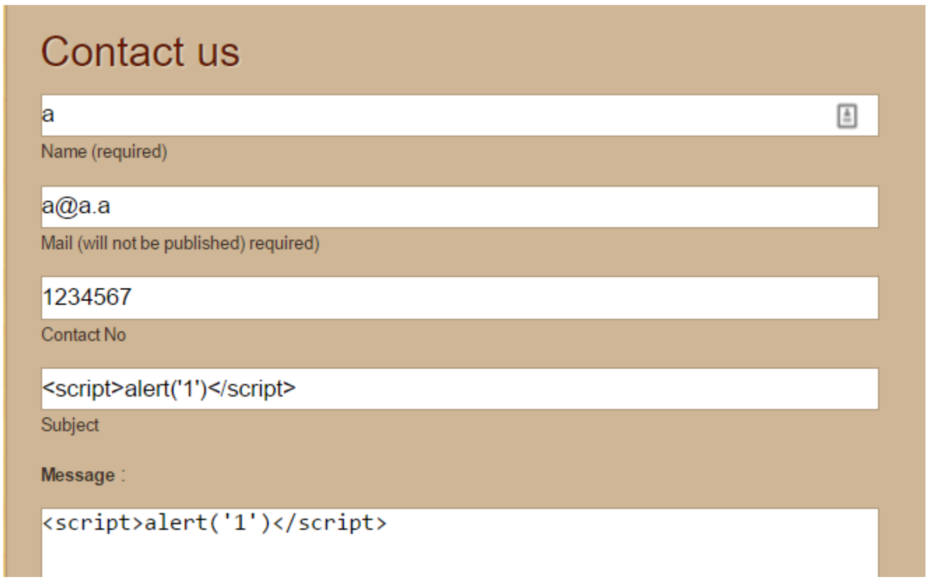
SQLmap -r header.txt -D studentinfo -T lectures

This entire database dump also revealed the administrator details including: name, ID number, address, phone number and password for the one (default) user with administrator credentials. The hash for the administrator password was: *f1c70784cef9f1986cf78c56f32e6cd.*

However, the default administrator account could not be accessed as the encoding method could not be determined. It was not encoded in Base64, Hexadecimal, HTML, SHA1 or even stored in plain-text, nor was it encoded using multiple methods. An attempt was also made to decode the hash from MD5, however, an MD5 hash must be 32 characters and the given hash was 31 characters. A brute-force attack was also attempted to crack the hash by padding the start and the end of the hash with numbers and letters (to bring it to the required 32-character length) but this was also unsuccessful.

1.5.3 Cross-Site Scripting

The next test was to find potential cross-site scripting (XSS) vulnerabilities. The first attempt was a simple test to make an alert box appear using the JavaScript code shown in *Figure 10* on the */contact.php* page. The result of the code injection can be previewed in *Figure 11*.



Contact us

a
Name (required)

a@a.a
Mail (will not be published) required)

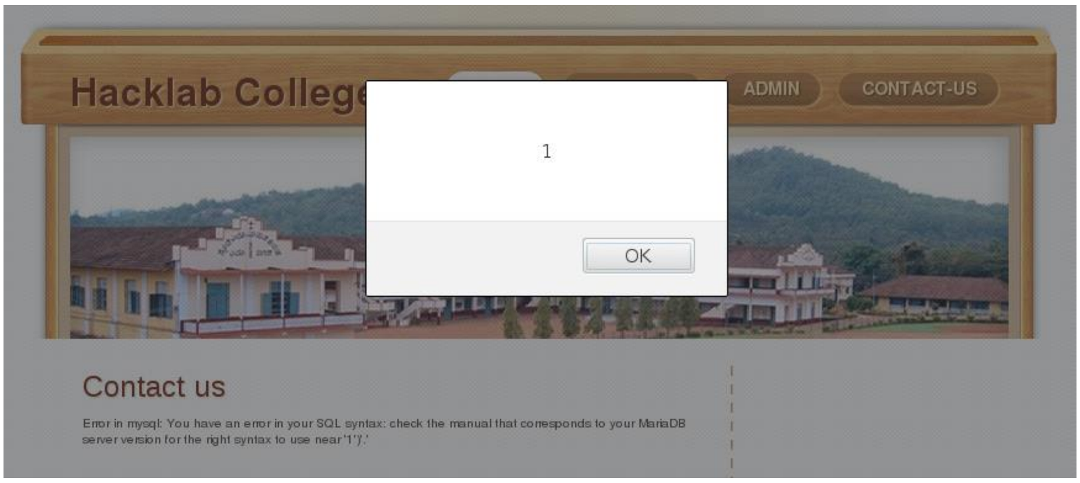
1234567
Contact No

<script>alert('1')</script>
Subject

Message :

```
<script>alert('1')</script>
```

Figure 10: XSS Code



Contact us

Error in mysql: You have an error in your SQL syntax: check the manual that corresponds to your MariaDB server version for the right syntax to use near '1')'.'

Figure 11: XSS Code Injection Result

The input boxes on the student login page (*/viewresult.php*) were also tested for cross-site scripting and an identical result was found using a slight modification of the above command in the *Roll No* input box:

‘”<script>alert(1);</script>

1.6 SESSION MANAGEMENT

1.6.1 Cookies

The cookie appeared to be consistent across every user and read as *1477598359%3ANzM2ZjclNzI2MzY1MmU3YTY5NzA%3D* which was encoded multiple times. The first step to decoding was to decode from URL to text which resulted in:

1477601999:NzM2Zjc1NzI2MzY1MmU3YTY5NzA=

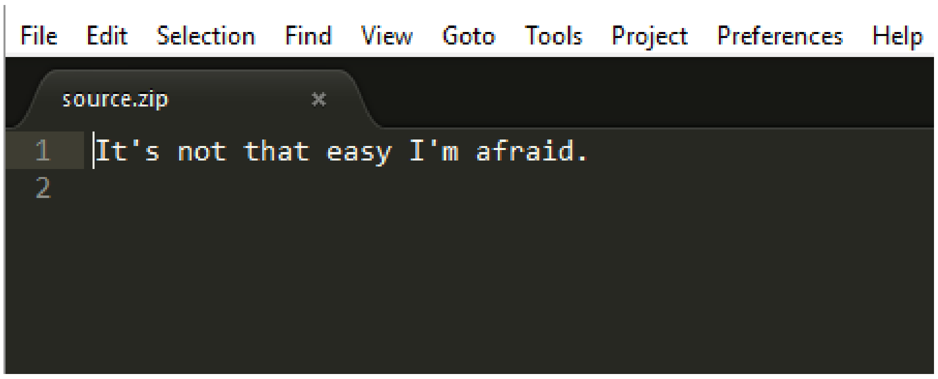
Then, decode the text after the colon from Base 64 to text:

736f7579v3652e7a6970

Finally, decode from hexadecimal to text:

source.zip

Navigating to /source.zip in the browser downloaded a "corrupted" .zip archive. Opening the .zip archive in a text editor showed the following:


File Edit Selection Find View Goto Tools Project Preferences Help
source.zip

```
It's not that easy I'm afraid.
```

Figure 12: Source.zip Viewed in Text Editor

1.7 INVESTIGATING AUTHENTICATION

1.7.1 Password Quality

From analysing the dumped database, it was observed that the password quality was generally sub-par, with most, if not all, passwords being susceptible to a brute-force password attack.

1.7.2 Username Enumeration

There was no way to directly enumerate usernames from the front-end of the website, however, the database dump provided the required information to determine that student usernames auto-increment from 100 and lecturer usernames auto-increment from 1. This may be advantageous if an attacker wanted to run a brute-force attack against a new user such as a lecturer. The attacker would be able to predict that the new user's user ID would be 4 and negate the need to run a brute-force attack against multiple user accounts.

1.7.3 Brute-Force Resilience

The website had no means in place to prevent a brute-force password attack, combining this with the generally weak passwords found means that an attacker could easily brute-force student and lecturer passwords.

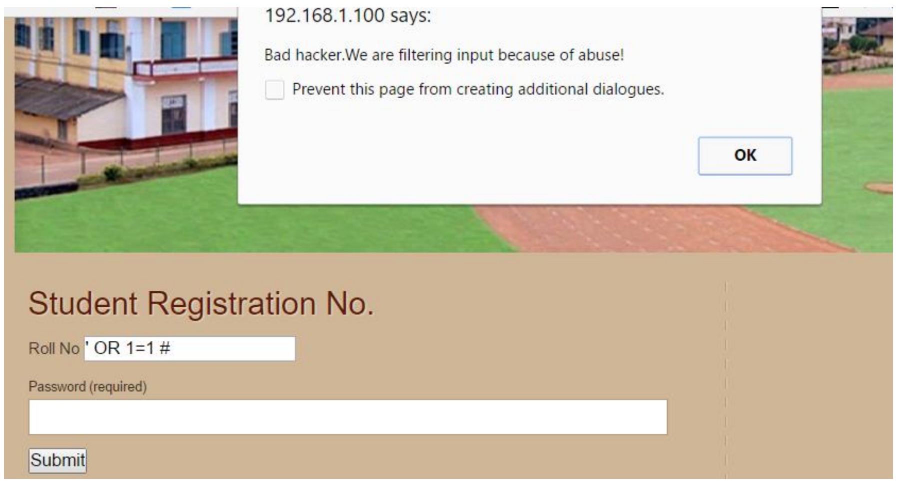
1.7.4 Unsafe Credential Transmission

The website was lacking an SSL certificate or HTTPS support, meaning that any data sent to and from the website was sent unencrypted. This would be susceptible to a man-in-the-middle attack where an attacker could intercept connections to the website and steal data such as database entries, usernames and passwords.

2. DISCUSSION AND CONCLUSIONS

The report presented in this paper has found the website to be vulnerable to attacks from a multitude of methods. This includes displaying information that should not necessarily be public (such as the SQL schema) and allowing unauthenticated access to add entries to the database (through the *courseinsert.php* page). The authentication was poor throughout the website. Not only did no password policy seem to be in place to enforce secure passwords, there was no brute-force attack protection and input validation was mostly done client-side, meaning any attempt to sanitise inputs was fairly simple to bypass as mentioned in *Section 1.3*.

There was some authentication in place to prevent SQL injection, however it was sub-par and only appeared to pick up certain SQL queries entered into input boxes. One such example was on the student login page (*/viewresult.php*). Entering the command *' OR 1=1 #* would display a JavaScript alert shown in *Figure 11* and stop the command being sent to the server. However, as mentioned in *Section 1.5.1* SQL injection was possible using a UNION SELECT command which was not filtered.


192.168.1.100 says:

Bad hacker.We are filtering input because of abuse!

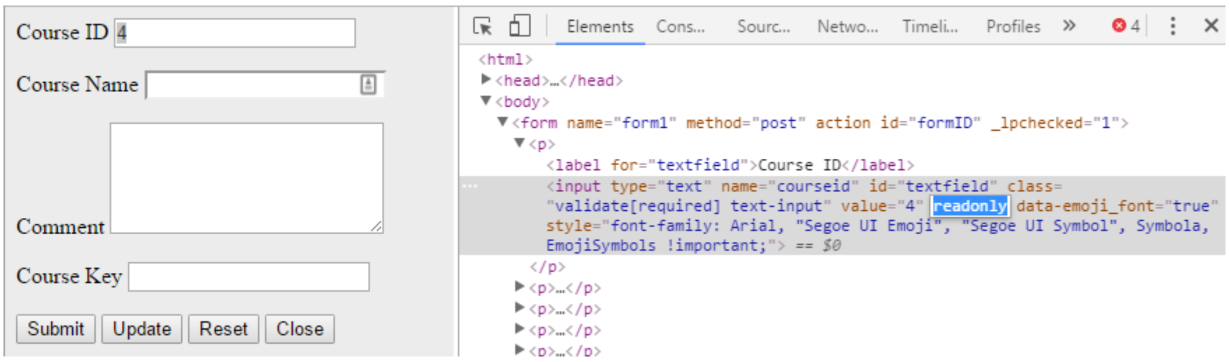
Prevent this page from creating additional dialogues.

Figure 13: Website Input Filtering

The website appeared to have no input sanitising to prevent against cross-site scripting attacks. The demonstration in *Section 1.5.3* offered a proof of concept to show that XSS attacks were possible, however it was a non-persistent pop-up box that was demonstrated. Persistence could have resulted in phishing attacks by redirecting users to a different website to steal passwords, or even just to send advertising using the JavaScript pop-up box. Furthermore, it could allow for cookie theft by sending a remote user's cookie to the hacker and allowing the hacker to log in as that user, however, persistent cookies, or even displaying the local user's cookie was not achieved.

While investigating the cookies, it was discovered that the cookie was not necessarily used to identify specific users and that every user shared the same cookie. In some cases, cookies can be decoded and may represent a username and password, however in this case it simply pointed to a decoy file pretending to offer the entire source code to the website.

The course insert page (*/courseinsert.php)* allowed for courses to be added to the database, as mentioned above. There was also an update button that allowed for existing courses to have their details updated. This was achievable by intercepting the HTTP traffic in Burpsuite, or by removing the read-only option in the element inspector, as demonstrated in *Figure 14.*

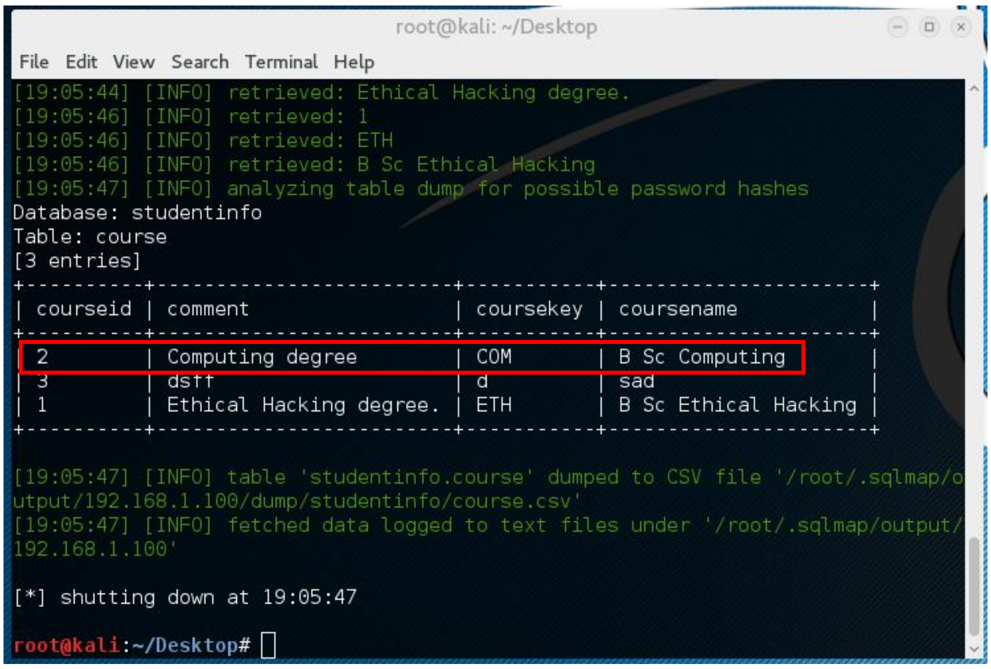


Course ID	4
Course Name	
Comment	
Course Key	

```
<html>
▶<head>…</head>
▼<body>
  ▼<form name="form1" method="post" action id="formID" _lpchecked="1">
    ▼<p>
        <label for="textfield">Course ID</label>
        <input type="text" name="courseid" id="textfield" class=
        "validate[required] text-input" value="4" readonly data-emoji_font="true"
        style="font-family: Arial, "Segoe UI Emoji", "Segoe UI Symbol", Symbola,
        EmojiSymbols !important;"> == $0
      </p>
    ▶<p>…</p>
    ▶<p>…</p>
    ▶<p>…</p>
    ▶<p>…</p>
```

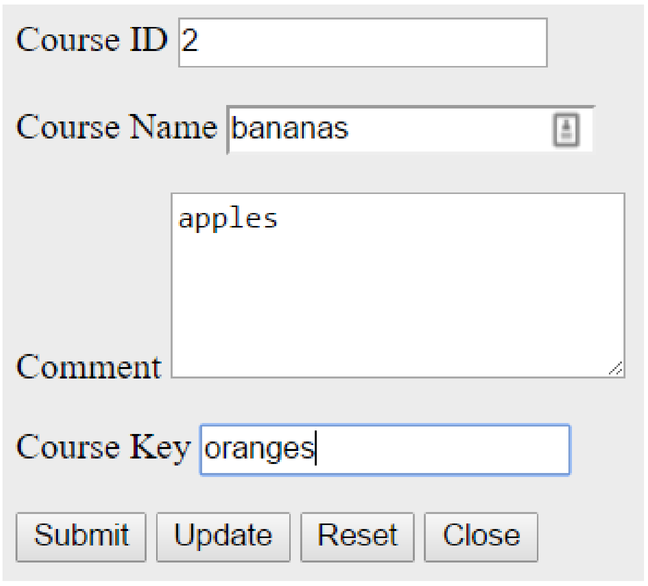
Figure 14: Removing Read-Only Functionality

To apply this practically, course ID #2 was updated as proof the method worked. *Figure 15* shows the courses table before the update (dumped in SQLmap), *Figure 16* shows the changes that were being sent to the database and *Figure 17* shows the table dump after sending the update query.



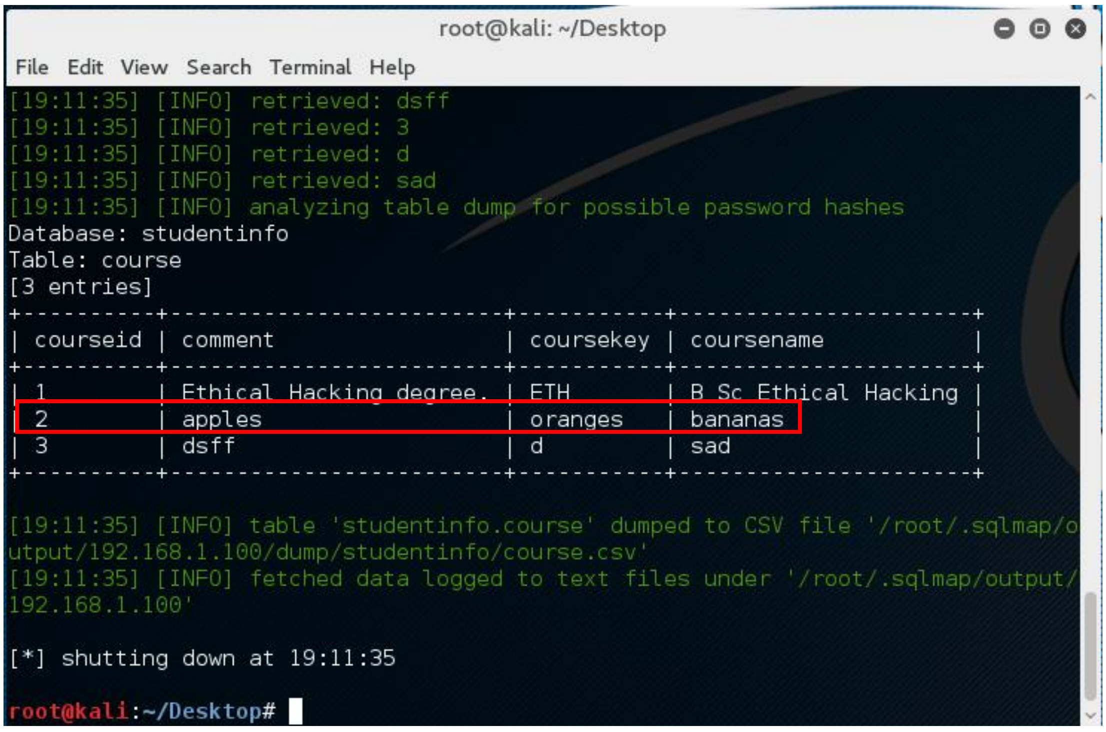
```
[19:05:44] [INFO] retrieved: Ethical Hacking degree.
[19:05:46] [INFO] retrieved: 1
[19:05:46] [INFO] retrieved: ETH
[19:05:46] [INFO] retrieved: B Sc Ethical Hacking
[19:05:47] [INFO] analyzing table dump for possible password hashes
Database: studentinfo
Table: course
[3 entries]
+----------+------------------------+----------+---------------------+
| courseid | comment                | coursekey | coursename          |
+----------+------------------------+----------+---------------------+
| 2        | Computing degree       | COM      | B Sc Computing      |
| 3        | dsff                   | d        | sad                 |
| 1        | Ethical Hacking degree. | ETH      | B Sc Ethical Hacking |
+----------+------------------------+----------+---------------------+

[19:05:47] [INFO] table 'studentinfo.course' dumped to CSV file '/root/.sqlmap/o
utput/192.168.1.100/dump/studentinfo/course.csv'
[19:05:47] [INFO] fetched data logged to text files under '/root/.sqlmap/output/
192.168.1.100'

[*] shutting down at 19:05:47

root@kali:~/Desktop#
```

Figure 15: Courses Table Before Changes



Course ID	2
Course Name	bananas
Comment	apples
Course Key	oranges

Figure 16: Changes Sent to the Database

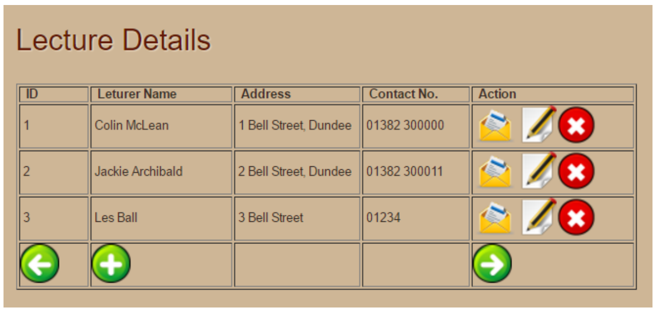


```
[19:11:35] [INFO] retrieved: dsff
[19:11:35] [INFO] retrieved: 3
[19:11:35] [INFO] retrieved: d
[19:11:35] [INFO] retrieved: sad
[19:11:35] [INFO] analyzing table dump for possible password hashes
Database: studentinfo
Table: course
[3 entries]
+----------+-------------------------+----------+---------------------+
| courseid | comment                 | coursekey | coursename          |
+----------+-------------------------+----------+---------------------+
| 1        | Ethical Hacking degree. | ETH      | B Sc Ethical Hacking |
| 2        | apples                  | oranges  | bananas             |
| 3        | dsff                    | d        | sad                 |
+----------+-------------------------+----------+---------------------+

[19:11:35] [INFO] table 'studentinfo.course' dumped to CSV file '/root/.sqlmap/o
utput/192.168.1.100/dump/studentinfo/course.csv'
[19:11:35] [INFO] fetched data logged to text files under '/root/.sqlmap/output/
192.168.1.100'

[*] shutting down at 19:11:35

root@kali:~/Desktop#
```

Figure 17: Courses Table After Changes

A user finding this page without having administrative access may wonder why there was an update button where the primary key was left open. The reason is that the page is meant to be opened from the administrative portal where an authenticated administrator has permissions to change the primary key. However, an unauthenticated user could bypass the authentication mechanism (as demonstrated) allowing for an attacker to essentially destroy course records by overwriting course information with meaningless data.

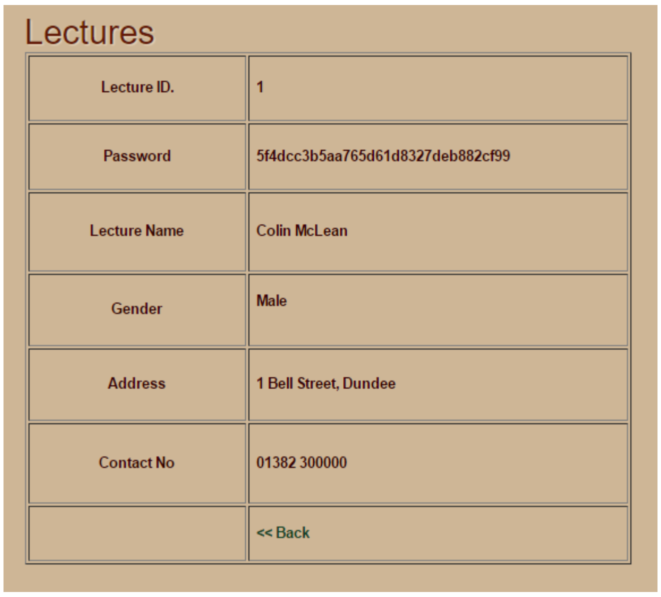
The next area to be analysed was the admin portal, after inserting a new administrative user with the *addadmin.php* page found in *Section 1.1.2*, the new administrator user account had full control over the database. Details for courses, subjects, lecturers, students, attendance and other administrators could be viewed, edited or deleted. Messages sent using the contact form could also be viewed in the admin portal, but as mentioned in *Section 1.1.2* this could be achieved unauthenticated anyway. The page for viewing lecturer details can be viewed in *Figure 18*, and the page for editing/adding lecturers can be viewed in *Figure 19*, below.



Lecture Details

ID	Leturer Name	Address	Contact No.	Action
1	Colin McLean	1 Bell Street, Dundee	01382 300000	✉ ✎ ✖
2	Jackie Archibald	2 Bell Street, Dundee	01382 300011	✉ ✎ ✖
3	Les Ball	3 Bell Street	01234	✉ ✎ ✖
←	⊕			→

Figure 18: All Lecturers Details on Admin Portal

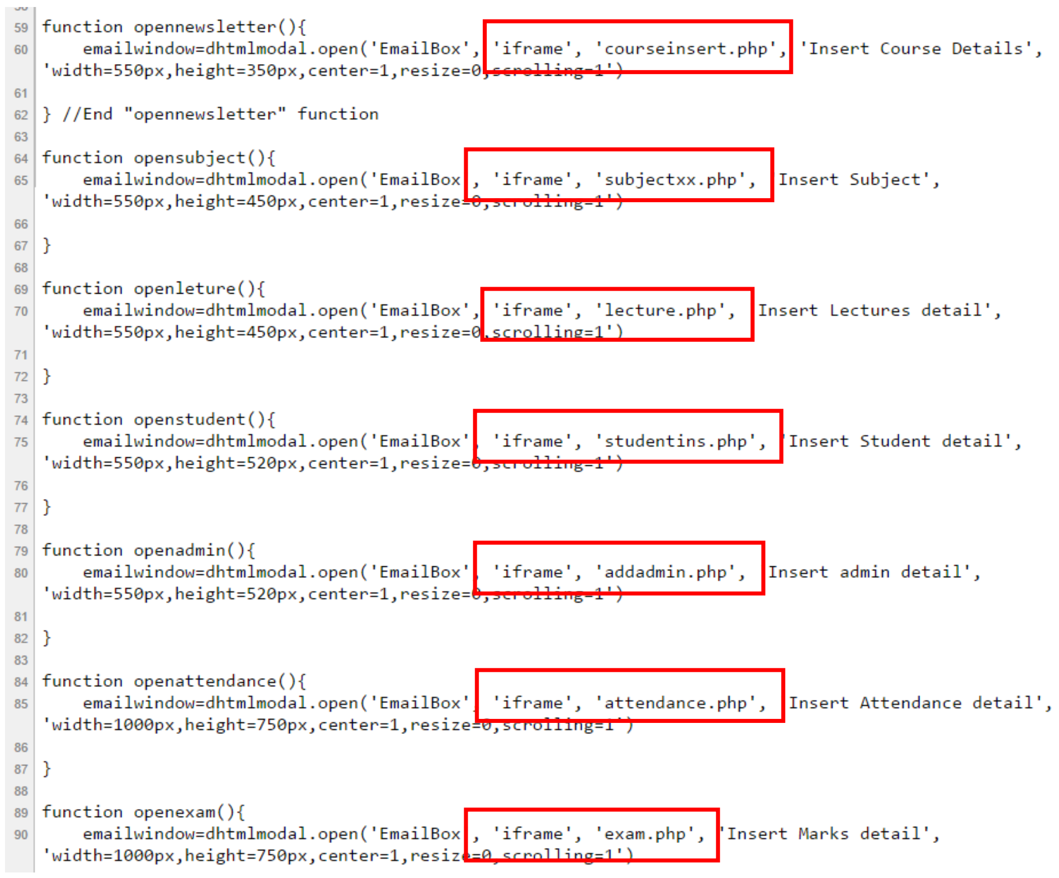


Lectures

Lecture ID.	1
Password	5f4dcc3b5aa765d61d8327deb882cf99
Lecture Name	Colin McLean
Gender	Male
Address	1 Bell Street, Dundee
Contact No	01382 300000
	<< Back

Figure 19: Individual Lecturer Details on Admin Portal

Analysing the HTML source for the various pages where lecturers, students etc. could be added found that the pages were using an iframe (embedding one page within another) as shown in *Figure 20*. Some of the pages that were referenced included *addadmin.php and courseinsert.php* which dirbuster found. Additionally, the HTML source included iframe references for *subjectxx.php, lecture.php, studentins.php, attendance.php* and *exam.php*, which were more pages that allowed the for subjects, lecturers, students, attendance and exam fields, respectively, to be modified. Dirbuster did not find these pages as the words were not in the wordlist. These pages allowed for all of the above to be inserted into the database unauthenticated. This means details could be inserted or updated for any table in the database by any user with no authentication.



```
function opennewsletter(){
    emailwindow=dhtmlmodal.open('EmailBox', 'iframe', 'courseinsert.php', 'Insert Course Details',
'width=550px,height=350px,center=1,resize=0,scrolling=1')

} //End "opennewsletter" function

function opensubject(){
    emailwindow=dhtmlmodal.open('EmailBox', 'iframe', 'subjectxx.php', 'Insert Subject',
'width=550px,height=450px,center=1,resize=0,scrolling=1')

}

function openleture(){
    emailwindow=dhtmlmodal.open('EmailBox', 'iframe', 'lecture.php', 'Insert Lectures detail',
'width=550px,height=450px,center=1,resize=0,scrolling=1')

}

function openstudent(){
    emailwindow=dhtmlmodal.open('EmailBox', 'iframe', 'studentins.php', 'Insert Student detail',
'width=550px,height=520px,center=1,resize=0,scrolling=1')

}

function openadmin(){
    emailwindow=dhtmlmodal.open('EmailBox', 'iframe', 'addadmin.php', 'Insert admin detail',
'width=550px,height=520px,center=1,resize=0,scrolling=1')

}

function openattendance(){
    emailwindow=dhtmlmodal.open('EmailBox', 'iframe', 'attendance.php', 'Insert Attendance detail',
'width=1000px,height=750px,center=1,resize=0,scrolling=1')

}

function openexam(){
    emailwindow=dhtmlmodal.open('EmailBox', 'iframe', 'exam.php', 'Insert Marks detail',
'width=1000px,height=750px,center=1,resize=0,scrolling=1')
```

Figure 20: iframe References in HTML Source Code

Although full administrative access was achieved very early on in the testing, it was still important to follow the methodology to find as many vulnerabilities as possible. Hiding the various pages that should be authenticated would prevent a hacker from gaining admin access, but a hacker would have still been able to dump the database and execute XSS attacks if the rest of the methodology was not followed.

2.1 FUTURE WORK

Some additional work could have been done within this paper to suggest possible counter-measures to many of the attacks demonstrated. The largest part of this includes improving the authentication mechanisms to prevent brute-force password attacks. This could include methods such as a temporary account lockout with too many incorrect guesses, or by blocking an IP address that has taken too many incorrect guesses. The latter of the two would still allow a legitimate user to log in, but block a hacker from continuing an attack. (OWASP, [no date]).

Methods of moving pages such as *courseinsert.php* and *addadmin.php* behind a login page could have also been suggested which would prevent unauthenticated users from editing the database and gaining administrative access to the website.

Some counter-measures to smaller flaws could have also been suggested, including removing the SQL database password from the HTML source, and ensuring that password confirmation boxes are actually authenticating the contents of the textbox.

Recommendations for authentication could have also been made, such as ensuring all input authentication was taking place server-side to stop a malicious user from manipulating the input with a proxy.

3. REFERENCES

Robots.txt. About /robots.txt. [no date]. Accessed 26 October 2016 at http://www.robotstxt.org/robotstxt.html

OWASP. OWASP DirBuster Project. [no date]. Accessed 26 October 2016 at https://www.owasp.org/index.php/Category:OWASP_DirBuster_Project

PC Mag. Definition of: Web Application. [no date]. Accessed 26 October 2016 at http://www.pcmag.com/encyclopedia/term/54272/web-application

OWASP. SQL Injection. 4 October 2016. Accessed 26 October 2016 at https://www.owasp.org/index.php/SQL_Injection

The Web Application Security Consortium. 2011. Accessed 26 October 2016 at http://projects.webappsec.org/w/page/13246920/Cross%20Site%20Scripting

OWASP. Testing Guide v4.0. [no date]. Accessed 15 November 2016 at https://www.owasp.org/images/1/19/OTGv4.pdf

MDSec. WAHH > Task Checklist. [no date]. Accessed 27 November 2016 at http://mdsec.net/wahh/tasks.html

Teach The Net. My sqlmap cheatsheet. 4 April 2011. Accessed 27 November 2016 at http://teachthe.net/?p=1481

Rest API Tutorials. HTTP Status Codes. [no date]. Accessed 28 November 2016 at http://www.restapitutorial.com/httpstatuscodes.html

OWASP. Blocking Brute Force Attacks. [no date]. Accessed 28 November 2016 at https://www.owasp.org/index.php/Blocking_Brute_Force_Attacks

Searchcode. Wfuzz /wordlist/fuzzdb/Discovery/PredictableRes/raft-large-files.txt. [no date]. Accessed 29 November 2016 at https://searchcode.com/codesearch/view/7456137/

4. APPENDICES

4.1 APPENDIX 1: DATABASE SCHEMA

```
-- MySQL dump 10.13  Distrib 5.6.30, for linux-glibc2.5 (x86_64)
--
-- Host: localhost    Database: studentinfo
-- ------------------------------------------------------
-- Server version	5.6.30


/*!40101 SET @OLD_CHARACTER_SET_CLIENT=@@CHARACTER_SET_CLIENT */;
/*!40101 SET @OLD_CHARACTER_SET_RESULTS=@@CHARACTER_SET_RESULTS
*/;
/*!40101 SET @OLD_COLLATION_CONNECTION=@@COLLATION_CONNECTION */;
/*!40101 SET NAMES UTF8 */;
/*!40103 SET @OLD_TIME_ZONE=@@TIME_ZONE */;
/*!40103 SET TIME_ZONE='+00:00' */;
/*!40014 SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0
*/;
/*!40014 SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS,
FOREIGN_KEY_CHECKS=0 */;
/*!40101 SET @OLD_SQL_MODE=@@SQL_MODE,
SQL_MODE='NO_AUTO_VALUE_ON_ZERO' */;
/*!40111 SET @OLD_SQL_NOTES=@@SQL_NOTES, SQL_NOTES=0 */;


--
-- Table structure for table `administrator`
--


DROP TABLE IF EXISTS `administrator`;
/*!40101 SET @saved_cs_client     = @@character_set_client */;
```

```
/*!40101 SET character_set_client = utf8 */;
CREATE TABLE `administrator` (
  `adminid` bigint(4) NOT NULL AUTO_INCREMENT,
  `password` varchar(50) NOT NULL,
  `adminname` varchar(80) NOT NULL,
  `address` text NOT NULL,
  `contactno` varchar(25) NOT NULL,
  PRIMARY KEY (`adminid`)
) ENGINE=InnoDB AUTO_INCREMENT=124 DEFAULT CHARSET=latin1;
/*!40101 SET character_set_client = @saved_cs_client */;

--
-- Table structure for table `attendance`
--

DROP TABLE IF EXISTS `attendance`;
/*!40101 SET @saved_cs_client     = @@character_set_client */;
/*!40101 SET character_set_client = utf8 */;
CREATE TABLE `attendance` (
  `attid` bigint(4) NOT NULL AUTO_INCREMENT,
  `studid` varchar(20) NOT NULL,
  `subid` bigint(4) NOT NULL,
  `totalclasses` int(2) NOT NULL,
  `attendedclasses` int(2) NOT NULL,
  `percentage` double(4,2) NOT NULL,
  `comment` text NOT NULL,
  PRIMARY KEY (`attid`),
  KEY `studid` (`studid`),
  KEY `subid` (`subid`),
  CONSTRAINT `attendance_ibfk_1` FOREIGN KEY (`subid`) REFERENCES `subject` (`subid`) ON DELETE CASCADE ON UPDATE CASCADE,
  CONSTRAINT `attendance_ibfk_2` FOREIGN KEY (`studid`) REFERENCES `studentdetails` (`studid`) ON DELETE CASCADE ON UPDATE CASCADE
) ENGINE=InnoDB AUTO_INCREMENT=18 DEFAULT CHARSET=latin1;
/*!40101 SET character_set_client = @saved_cs_client */;

--
-- Table structure for table `contact`
--

DROP TABLE IF EXISTS `contact`;
/*!40101 SET @saved_cs_client     = @@character_set_client */;
/*!40101 SET character_set_client = utf8 */;
CREATE TABLE `contact` (
  `contactid` bigint(4) NOT NULL AUTO_INCREMENT,
  `name` varchar(25) NOT NULL,
  `emailid` varchar(30) NOT NULL,
  `contactno` varchar(20) NOT NULL,
  `subject` varchar(20) NOT NULL,
  `message` text NOT NULL,
  PRIMARY KEY (`contactid`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
/*!40101 SET character_set_client = @saved_cs_client */;

--
-- Table structure for table `course`
--
```

```
DROP TABLE IF EXISTS `course`;
/*!40101 SET @saved_cs_client     = @@character_set_client */;
/*!40101 SET character_set_client = utf8 */;
CREATE TABLE `course` (
  `courseid` bigint(4) NOT NULL AUTO_INCREMENT,
  `coursename` varchar(40) NOT NULL,
  `comment` text NOT NULL,
  `coursekey` varchar(15) NOT NULL,
  PRIMARY KEY (`courseid`)
) ENGINE=InnoDB AUTO_INCREMENT=3 DEFAULT CHARSET=latin1;
/*!40101 SET character_set_client = @saved_cs_client */;

--
-- Table structure for table `examination`
--

DROP TABLE IF EXISTS `examination`;
/*!40101 SET @saved_cs_client     = @@character_set_client */;
/*!40101 SET character_set_client = utf8 */;
CREATE TABLE `examination` (
  `examid` bigint(4) NOT NULL AUTO_INCREMENT,
  `studid` varchar(20) NOT NULL,
  `subid` bigint(4) NOT NULL,
  `courseid` bigint(4) NOT NULL,
  `internaltype` varchar(20) NOT NULL,
  `maxmarks` int(2) NOT NULL,
  `scored` int(2) NOT NULL,
  `percentage` float NOT NULL,
  `result` text NOT NULL,
  PRIMARY KEY (`examid`),
  KEY `subid` (`subid`),
  KEY `studid` (`studid`),
  KEY `courseid` (`courseid`),
  CONSTRAINT `examination_ibfk_1` FOREIGN KEY (`studid`) REFERENCES
`studentdetails` (`studid`) ON DELETE CASCADE ON UPDATE CASCADE,
  CONSTRAINT `examination_ibfk_2` FOREIGN KEY (`courseid`) REFERENCES `course`
(`courseid`) ON DELETE CASCADE ON UPDATE CASCADE,
  CONSTRAINT `examination_ibfk_3` FOREIGN KEY (`subid`) REFERENCES `subject`
(`subid`) ON DELETE CASCADE ON UPDATE CASCADE
) ENGINE=InnoDB AUTO_INCREMENT=9 DEFAULT CHARSET=latin1;
/*!40101 SET character_set_client = @saved_cs_client */;


--
-- Table structure for table `lectures`
--


DROP TABLE IF EXISTS `lectures`;
/*!40101 SET @saved_cs_client     = @@character_set_client */;
/*!40101 SET character_set_client = utf8 */;
CREATE TABLE `lectures` (
  `lecid` bigint(4) NOT NULL AUTO_INCREMENT,
  `password` varchar(50) NOT NULL,
  `courseid` bigint(4) NOT NULL,
  `lecname` varchar(50) NOT NULL,
  `gender` varchar(50) NOT NULL,
  `address` varchar(100) NOT NULL,
  `contactno` varchar(15) NOT NULL,
  PRIMARY KEY (`lecid`),
  KEY `courseid` (`courseid`),
  CONSTRAINT `lectures_ibfk_1` FOREIGN KEY (`courseid`) REFERENCES `course`
(`courseid`) ON DELETE CASCADE ON UPDATE CASCADE
) ENGINE=InnoDB AUTO_INCREMENT=4 DEFAULT CHARSET=latin1;
/*!40101 SET character_set_client = @saved_cs_client */;


--
-- Table structure for table `semester`
--


DROP TABLE IF EXISTS `semester`;
/*!40101 SET @saved_cs_client     = @@character_set_client */;
/*!40101 SET character_set_client = utf8 */;
CREATE TABLE `semester` (
  `semid` bigint(4) NOT NULL AUTO_INCREMENT,
  `semester` varchar(25) NOT NULL,
  `comment` text NOT NULL,
  PRIMARY KEY (`semid`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
/*!40101 SET character_set_client = @saved_cs_client */;


--
-- Table structure for table `studentdetails`
--


DROP TABLE IF EXISTS `studentdetails`;
/*!40101 SET @saved_cs_client     = @@character_set_client */;
/*!40101 SET character_set_client = utf8 */;
CREATE TABLE `studentdetails` (
  `studid` varchar(25) NOT NULL,
  `studfname` varchar(20) NOT NULL,
  `studlname` varchar(20) NOT NULL,
  `fathername` varchar(25) NOT NULL,
  `gender` varchar(20) NOT NULL,
  `address` varchar(100) NOT NULL,
  `contactno` varchar(20) NOT NULL,
  `courseid` bigint(4) NOT NULL,
  `semester` varchar(20) NOT NULL,
  `dob` date NOT NULL,
  `password` varchar(20) NOT NULL,
  PRIMARY KEY (`studid`),
  KEY `courseid` (`courseid`),
  CONSTRAINT `studentdetails_ibfk_1` FOREIGN KEY (`courseid`) REFERENCES `course` (`courseid`) ON DELETE CASCADE ON UPDATE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
/*!40101 SET character_set_client = @saved_cs_client */;

--
-- Table structure for table `subject`
--

DROP TABLE IF EXISTS `subject`;
/*!40101 SET @saved_cs_client     = @@character_set_client */;
/*!40101 SET character_set_client = utf8 */;
CREATE TABLE `subject` (
  `subid` bigint(4) NOT NULL AUTO_INCREMENT,
  `subname` varchar(20) NOT NULL,
  `courseid` bigint(4) NOT NULL,
  `lecid` bigint(4) NOT NULL,
  `subtype` varchar(25) NOT NULL,
  `semester` varchar(25) NOT NULL,
  `comment` text NOT NULL,
  PRIMARY KEY (`subid`),
  KEY `courseid` (`courseid`),
  CONSTRAINT `subject_ibfk_1` FOREIGN KEY (`courseid`) REFERENCES `course` (`courseid`) ON DELETE CASCADE ON UPDATE CASCADE
) ENGINE=InnoDB AUTO_INCREMENT=7 DEFAULT CHARSET=latin1;
/*!40101 SET character_set_client = @saved_cs_client */;
/*!40103 SET TIME_ZONE=@OLD_TIME_ZONE */;

/*!40101 SET SQL_MODE=@OLD_SQL_MODE */;
/*!40014 SET FOREIGN_KEY_CHECKS=@OLD_FOREIGN_KEY_CHECKS */;
/*!40014 SET UNIQUE_CHECKS=@OLD_UNIQUE_CHECKS */;
/*!40101 SET CHARACTER_SET_CLIENT=@OLD_CHARACTER_SET_CLIENT */;
/*!40101 SET CHARACTER_SET_RESULTS=@OLD_CHARACTER_SET_RESULTS */;
/*!40101 SET COLLATION_CONNECTION=@OLD_COLLATION_CONNECTION */;
/*!40111 SET SQL_NOTES=@OLD_SQL_NOTES */;

-- Dump completed on 2016-10-18 13:31:31
```