# Secure Computer Systems I: Lab 1

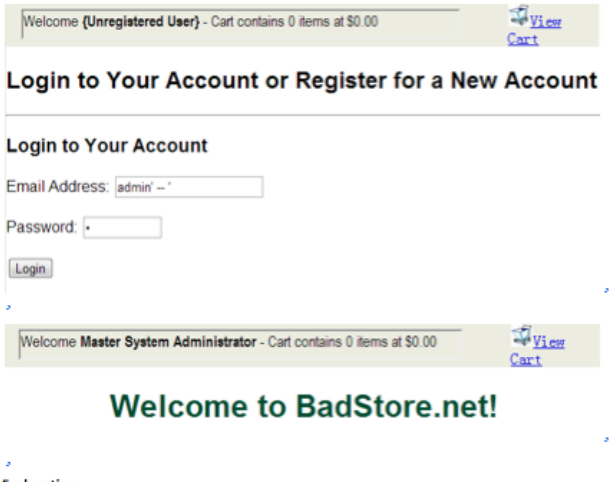
Ren Li Tianyao Ma Samuel Pettersson

January 29, 2014

## Task 1: SQL injections

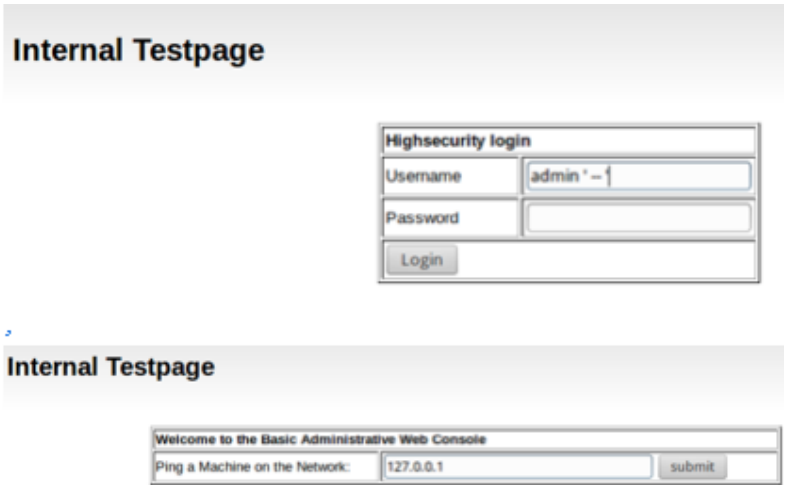
### Exercises:

(a) The result shows below:




Explanation: Before injection, the SQL code should be SELECT * FROM user WHERE EmailAddress = 'emailaddress' AND Password = 'password'. Then during the injection, we input admin '–' in the text of Email Address and whatever password is in the text of Password. The WHERE condition is altered into WHERE EmailAddress = 'admin' – ' AND Password = 'password' (WHERE EmailAddress = 'admin'). So there is no need of password and we can directly log into admin account.

(b) The result shows below (Continued on next page):




Explanation:Before injection, the SQL code should be SELECT * FROM user WHERE UserName = 'username' AND Password = 'password'. Then during the injection, we input admin '–' in the text of Username and whatever password is in the text of Password. The WHERE condition is altered into WHERE UserName = 'admin' – ' AND Password = 'password' (WHERE UserName = 'admin'). So there is no need of password and we can directly log into admin account.

(c) **Prepared Statement**

Prepared statement is also called parametrized statement, which is a SQL query with variables inside of it. That is to say, we prepare the query with blank spot to fill and it will automatically protect the query from SQL injection.

Here is the example:

*stmt = dbh->prepare("INSERT INTO REGISTRY (name, value) VALUES (:name, :value)");*

*stmt->bindParam(':name', name);*

*stmt->bindParam(':value', value);*

In our case, we can create prepared statement as follows: prepare("SELECT * FROM user WHERE UserName = :username AND Password = :password"); bind(':username', username); bind(':password', password);

**Explanation**

Before injection, we have two variables, admin(username) and test(password). The SQL query should be SELECT * FROM user WHERE username = 'admin' AND password = 'test'. Then during the injection, these two variables are changed into admin' –(username) and 123(password) and the SQL query becomes SELECT * FROM user WHERE username = 'admin'--' AND password = '123'. It's because the binding system will automatically change the input to protect the query. So the SQL injection doesn't work in this situation.

(d) The school lost all the student records because the table Students in the database is dropped by SQL injection. Before injection, the SQL code should be INSERT INTO Students VALUES ('Robert'). Then during the injection, the query is altered into INSERT INTO Students VALUES ('Robert'); DROP TABLE Students; – . So after the query is executed, the table Students will be dropped automatically.

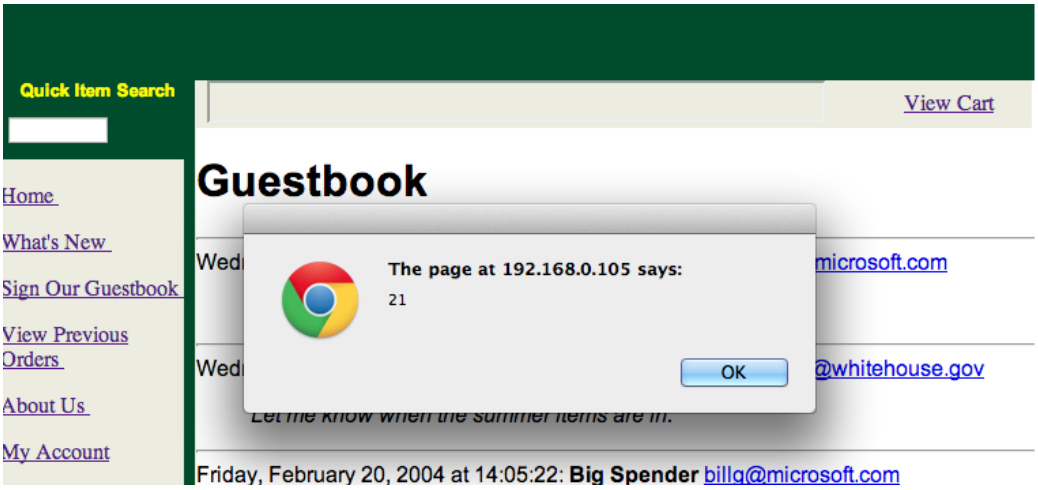
What the school should do is to sanitize the database inputs by utilizing prepared statement as mentioned above. They can create prepared statement as follows: prepare("INSERT INTO Students VALUES (':name')"); bind(':name',name);

# Task 2: XSS and CSRF

## Exercises:

(a) The steps are as follows:

- Open BadStore in your browser.
- Click "Sign Our GuestBook" on the left side of the page.
- In the "Your Name" field, type whatever you want.
- In the "Email" field, type whatever you want.
- In the "Comments" field, type *<script>alert("1")</script>* .
- Click "Add Entry", the page should like this below:




Explanation: In the example, we write a piece of Javascript code in the comment area and submit it. It can work because the website has XSS vulnerability which enable us to insert and perform Javascript in the website. When it reloads, it will generate the current HTML code which contains the code we just injected. And it works as a result.

(b) To perform CSRF attacks, we need to create a custom web page contains deceived information. The function is to let user comment in the guestbook and receive countless alert. The key point is to guide victim to visit your page and click on the information. As the reason that we demonstrate it just as an example, we can suppose that the victim will click it. The custom page is so simple that it is just written as a demo. Below are the steps:

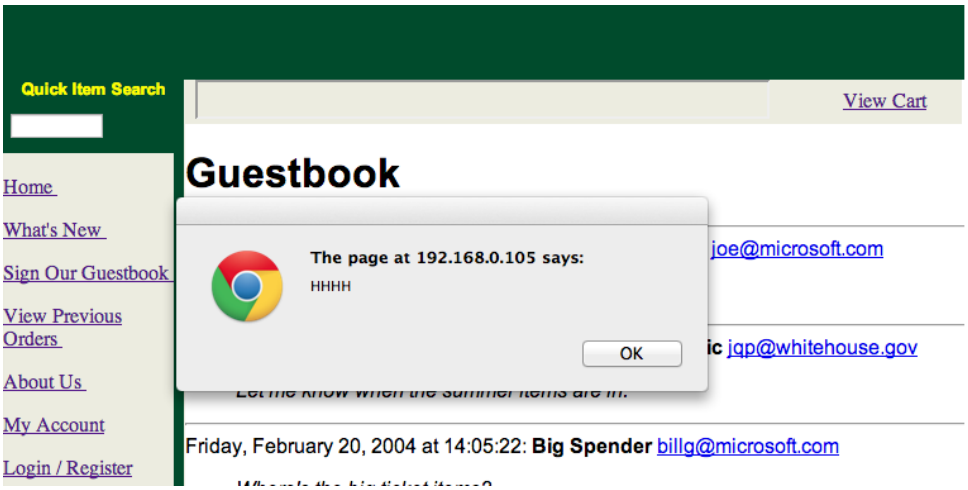
- Write a simple page containing these code:

```
<!DOCTYPE html>
<html>
<head>
        <title>iPhone Free!</title>
</head>
<body>
        <FORM METHOD="POST" ACTION="http://10.0.2.5/cgi-bin/badstore.cgi?action=doguestbook">
                <INPUT TYPE=hidden NAME=name value="<script>for(var␣i=0;i<10000;i++){alert('HHHH');}</script>" SIZE=30
                <INPUT TYPE=hidden NAME=email value="1" SIZE=40>
                <TEXTAREA style="display:none" NAME=comments COLS=60 ROWS=4 value="aaa"> </TEXTAREA>
                <INPUT TYPE=submit VALUE="Get␣Now!">
                <INPUT style="display:none" TYPE=reset></Center>
```

```
13          </FORM>
14  </body>
15  </html>
```

- The page shows like this:


- Open BadStore and login in as an user.
- Click the "Get Now!" button in the custom page.
- The page will be redirected into BadStore and shows like this:




- The alert will repeat again and again as well as leaving many comments.
- Note: There might be some different performance in different browsers. i.e. Safari will operate the code after you quit it and reopen it.

Explanation: In the example, we use a self-written page to perform CSRF attack. As mentioned above, the key point is that the attacker need to persuade the victim to visit the deceived page which contains evil code. The code can work with following reasons. First, the code will perform action in the server (BadStore) such as submitting an order or transferring money to another account. Second, the code will work because the victim doesn't close the server website. To make it clear, the cookie of the server website still exists in the browser. As a result, when the server receives the deceived request, it will think that the request is made by the victim and perform related operation.

(c) The difference is that, in an XSS attack, attacker makes the attack by inserting the Javascript code inside the site. Because the site do not have strict check on user inputs, the code can be planted into HTML which will be operated on loading. On the contrary, CSRF attack don't necessarily need Javascript. What count most is that attacker must acquire trust from user and let him to operate the deceived request. Also, the user can't quit the page because the code will only work when the cookie of the target page stays. It's not the attacker himself that make it work.

(d) [**higher grades only**]

(e) [**higher grades only**]

# Task 3: Authentication

The third and last task of the lab was about authentication, and the exercises were of a theoretical rather than practical nature.

### Exercises:

(a) Hashing a list of passwords with a one-way hash function, say $f$, rather than having the passwords stored in cleartext is a method for reducing the impact of a database leak. Authentication given a password $p$ in such a system is done by computing the hash $f(p)$ of the password and comparing it with the hash for the specific user stored in the database.

In the event that the password hashes are exposed to an attacker, there is no trivial way for him or her to retrieve the passwords corresponding to the compromised password hashes, by the definition of a one-way hash function. Thus, there is no trivial way of authenticating despite having access to both the hash function and the hashes. If an attacker finds the passwords stored in cleartext, on the other hand, he or she can authenticate as any of the affected users on the system. Furthermore, because passwords are frequently reused in more than one system, the attacker might be able to authenticate falsely on other systems as well. In other words, the impact of a cleartext password leak could encompass more than just the attacked system.

While there is no *trivial* way of finding a password corresponding to a specific hash value, there is a cumbersome but in practice feasible way of doing so: password guessing. The idea is to repeatedly come up with a guess for the password, hash it, and compare the result with the hash value in question, continuing until a password guess with the specific hash is found. The hash values for guesses for common passwords can be precomputed into a so-called rainbow table that allow for these passwords to be looked up given their hash value.

Salting is a method for making password guessing more difficult. In addition to storing a hash value for each user, a random string $s$—the salt—is stored for each user as well. Authentication given a password $p$ is done by computing the hash of a combination of the password and the salt, such as $s+p$ where "+" denotes a concatenation. The resulting hash, $f(s+p)$, is then compared with the hash value in the database.

If an attacker gets hold of the database, which contains a salt and hash for each user, and aims for finding the password of *any* of the users, the addition of the salt will make password guessing more difficult. Let $n$ be the number of users. Under the assumption that each user has a unique salt, the attacker will have to compute $n$ hashes for each password guess, compared to the single computation that is required with hashing alone. Salting offers increased security also against attacks with precomputed tables, seeing as one table would be needed for each possible salt in order to cover the same passwords as without salting.

(b) Assuming that the salt and salted hash are known to an attacker, salting may make the attack on a specific account more difficult, depending on whether precomputed tables are used.

If precomputed tables are used and the attacker did not choose the specific user because the attacker had a precomputed table for that specific salt, salting will make the attack more difficult in that more precomputed tables will be required.

Without precomputation, however, each password guess will require one computation of a hash value, assuming that the hash is computed like mentioned before: $f(s+g)$ for hash function $f$, salt $s$, password guess $g$, and concatenation operator $+$. The only difference in the case that salting is not used is that no salt has to be concatenated; the hash function still has to be evaluated once for each guess. Under the assumption that concatenating the string takes negligible time compared to evaluating the hash function, salting does not make the attack in question harder.

(c) [**higher grades only**]

- 
- 

(d)
- While the fingerprint sensor of the iPhone 5s can evidently be deceived in a variety of ways, there are at least two reasons that it is a good addition to the iPhone security. The first reason is that the use of the fingerprint sensor in conjunction with more traditional authentication methods such as PIN codes adds another layer for the attacker to get past. The other reason, while not as convincing as the first, is that the sensor might provide a strong enough protection for some users, by the principle of adequate protection.
- 