Secure Computer Systems I: Lab 1

Ren Li Tianyao Ma Samuel Pettersson

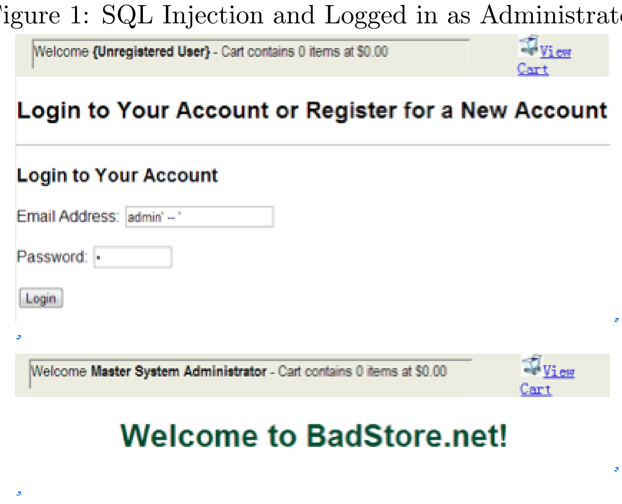
March 23, 2014

Task 1: SQL injections

Exercises:

(a) The result is shown below:

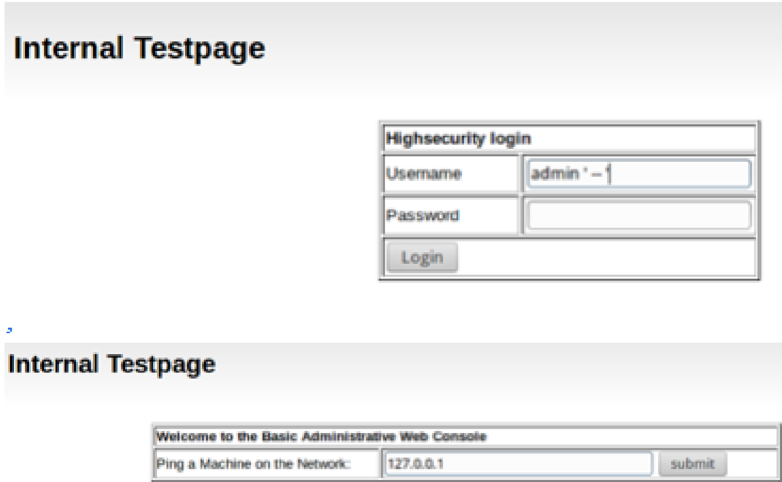
Figure 1: SQL Injection and Logged in as Administrator



Explanation Without injection, the SQL code is believed to be `SELECT * FROM user WHERE EmailAddress = 'emailaddress' AND Password = 'password'`. Then during the injection, we input `admin '--'` in the text box for the email address and some arbitrary password in the text box for the password. The WHERE condition is altered into `WHERE EmailAddress = 'admin' -- ' AND Password = 'password'`, where the part following the two dashes is interpreted as a SQL comment. In other words, the WHERE condition is turned into `WHERE EmailAddress = 'admin'`. Thus there is no need for a password and we can directly log into the admin account.

(b) The result is shown in Figure 2
Explanation The underlying SQL code for the login screen for host 192.168.2.134 is believed to be the same as before: without injection, the SQL code is thought to be `SELECT * FROM user WHERE UserName = 'username' AND Password = 'password'`. Then during the injection, we input `admin '--'` in the text box for the username and some arbitrary password in the text box for the password. The WHERE condition is altered into `WHERE UserName = 'admin' -- ' AND Password = 'password'` (or equivalently, `WHERE UserName = 'admin'`). So there is no need for a password and we can directly log into the admin account.

Figure 2: SQL Injection and the Result



(c) **Prepared Statement** A prepared statement, which is also called a parametrized statement, is a SQL query with variables inside of it. That is to say, we prepare the query with blank spots to fill and it will automatically protect the query from SQL injection.

Here is a code snippet for using a prepared statement in PHP[1]:

```
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 a prepared statement as follows:

```
prepare("SELECT * FROM user WHERE UserName = :username AND
   Password = :password");
bind(':username', username);
bind(':password', password);
```

Explanation There are two variables in the prepared statement: username and password. Without injection, for instance with the username `admin` and the password `test`, the SQL query for logging on would be `SELECT * FROM user WHERE username = 'admin' AND password = 'test'`. During an injection attempt, for instance with the username `admin' --` and password `123`, the SQL query becomes `SELECT * FROM user WHERE username = 'admin\' --' AND password = '123'`. This is because the binding system will automatically change the input to protect the query. Thus, 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. The SQL statement intended to be executed can be assumed to be of the form `INSERT INTO Students VALUES ('Robert');`. The SQL injection had this query altered into the following two statements and comment: `INSERT INTO Students VALUES ('Robert'); DROP TABLE Students ; --');`. When the query is executed, the table Students will be dropped.

What the school should do is to sanitize the database inputs by utilizing prepared statements as mentioned above. They can create the prepared statement as follows:

```
prepare("INSERT INTO Students VALUES (':name')");
bind(':name',name);
```

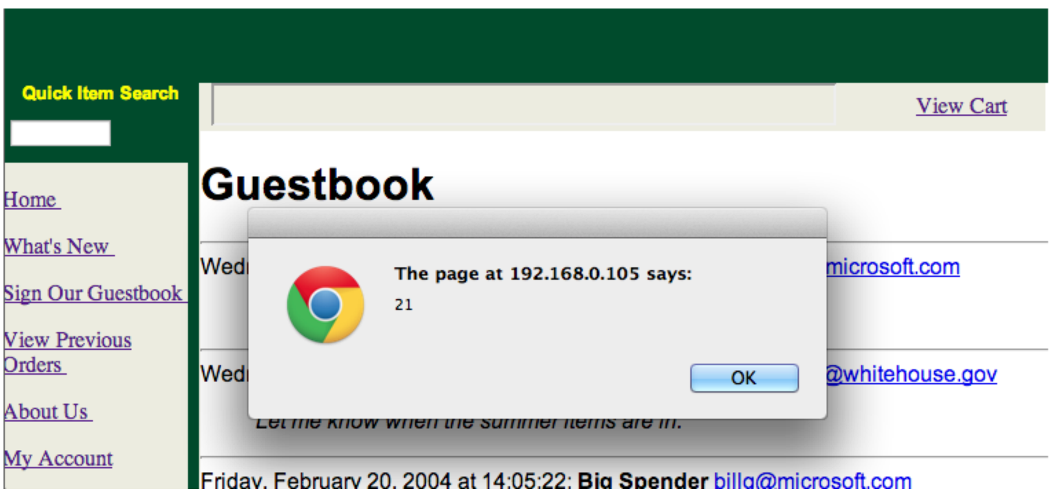
Task 2: XSS and CSRF

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:

Figure 3: Alert Made by XSS Attack



Explanation: In the example, we write a piece of Javascript code enclosed in script tags in the comment area and submit it to the guest book. It works because the website has an XSS vulnerability which enables us to insert Javascript into the website and have it executed by other users. When the page reloads, the server will just paste the tag-embraced Javascript code into the HTML code in such a way that it is interpreted as a Javascript snippet rather than the text `<script>alert("1")</script>`.

(b) To perform a CSRF attack, we need to create a custom web page containing deceptive information. The objective of the attack is to have an unsuspecting user make an XSS comment in the guestbook for generating countless alert messages. The key point necessary to execute the attack is to guide the victim to our page and have him or her click on the button. As we demonstrate it just as an example, we can suppose that the victim will click on it. The custom page is so simple that it is just written as a demo. Below are the steps for carrying out the attack:

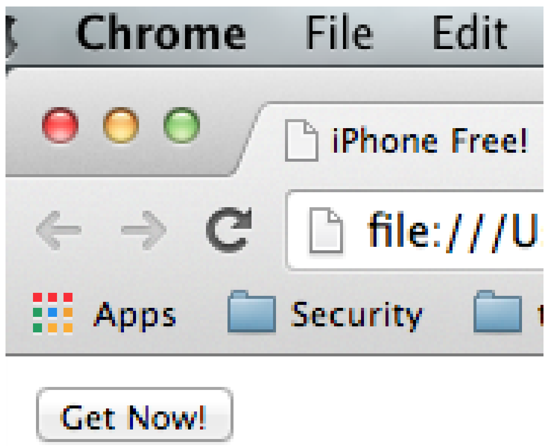
- As the attacker, write a simple page containing this code:

```
<!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>
</FORM>
</body>
</html>
```

- The page shows like this:

Figure 4: Fake Page

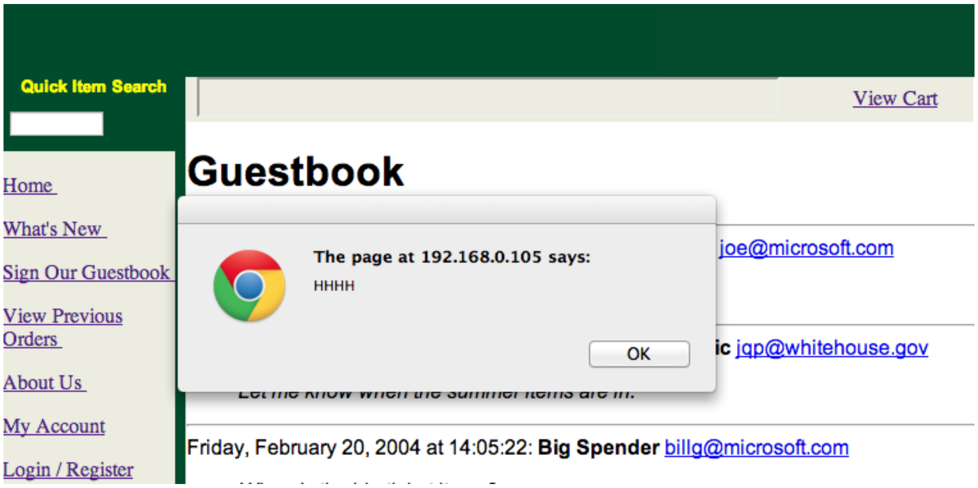


- As the victim, open BadStore and log in as a user.
- Visit the malicious page and click the "Get Now!" button.
- The victim will then be redirected into BadStore, as is shown in Figure 5.
- The alert will repeat again and again as well as leaving many comments.
- Note: There might be some different 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 the CSRF attack. As mentioned above, the key point is that the attacker needs to persuade the victim to visit the deceived page which contains the evil code. The code works as follows: The code sends a request to the server (BadStore), which could be as serious as submitting an order or transferring money to another account; and it works if the victim does not 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 the related operation.

(c) The difference is that, in an XSS attack, the attacker makes the attack by inserting Javascript code (or other code) into the site. Because the site does not have strict checks on user inputs, the code can be planted into HTML and executed when the page is loaded. On the contrary, CSRF attacks do not necessarily need Javascript [2]. The most important part is that the attacker must acquire trust from the user and have him operate the deceived request. Also, the user cannot quit the page because the code will only work when the cookie of the target page stays in the browser. It is not the attacker himself that makes it work.

Figure 5: Result after Performing CSRF Attack



(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[3]. 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[5] 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[4]. 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.
- The principle of authenticating with something you have could be leveraged to increase the security of the phone unlocking mechanism drastically. One could imagine that the presence of a physical device—a *token*—capable of communicating with the phone in a wireless manner would be required to unlock the phone. With a challenge-response system, the token could prove its presence without opening up the possibility of a replay attack.

  In this, the time of smart devices, the token need not be a piece of special-purpose hardware with the only purpose of authenticating the possessor to the phone; for instance, a smartwatch could be assigned the task of authentication on the phone.

  While requiring a separate device of some sort to unlock the phone is obviously beneficial in the case that the phone but not the device is lost, a downside of the system is apparent in the opposite scenario: losing the device but not the phone. In that case, the phone is rendered unusable. There being ramifications of an authentication system is not unique to those based on the principle of what you have though; the situation of losing one's device can be likened to that of forgetting one's PIN code, which also renders the phone unusable.

References

[1] php.net "Prepared statements and stored procedures" accessed 2014–03–22. [Online] Available: `http://www.php.net/pdo.prepared-statements`

[2] Wikipedia, "Cross-site request forgery", accessed 2014–02–01. [Online] Available: `http://en.wikipedia.org/wiki/Cross-site_request_forgery`

[3] M. Bishop, "Computer Security: Art and Science", *"Attacking a Password System"*

[4] M. Bishop, "Computer Security: Art and Science", *"Countering Password Guessing"*

[5] Wikipedia, "Rainbow table", accessed 2014–01–29. [Online] Available: `http://en.wikipedia.org/wiki/Rainbow_table`