LAB 18

SECURITY

What You Will Learn

- Some good development and administration practices to secure your sites.
- Some math behind encryption
- Some techniques hackers use and how to test your defenses.

Approximate Time

The exercises in this lab should take approximately 120 minutes to complete.

Fundamentals of Web Development, 2nd Ed

Randy Connolly and Ricardo Hoar

Textbook by Pearson
http://www.funwebdev.com

Date Last Revised: May 17, 2017

Preparing Directories

1. If you haven't done so already, create a folder in your personal drive for all the labs for this book.
2. From the main `labs` folder (either downloaded from the textbook's web site using the code provided with the textbook or in a common location provided by your instructor), copy the folder titled `lab0X` to your course folder created in step one.

Security has both high level and technical aspects that a web developer should be aware of. This lab covers some practical system administration practices, as well as some more theoretical concepts like modulo arithmetic. Many of these exercises require increased access (root) and may not be suitable for shared systems.

To get the most out of the exercises from this and subsequent chapter you will be applying these exercises at times to your own sites.

Good Practices

Exercise 18.1 — Website Backups

1. This first exercise should be done on a website you are currently using, or have created for a previous assignment.
2. If your site uses a database, it will need to be backed up. The MySQL dump command outputs a database as a file from which the database can be reconstituted. The format of the command is:

   ```
   mysqldump -u db_user -p[db_user_pass] [database_name] > outputfilename.sql
   ```

   Examine the .sql file that was output, you should see that it contains all the SQL commands to create a database, and all the insert commands for all the data.
3. Consider what file location you want to use to store backups and then create a backup script file (not visible from the web) named siteBackup.script.

   Place your `sqldump` command into the script as the first line.

   Now *chmod* that file to be executable by anyone:

   ```
   chmod 755 siteBackup.script
   ```

   and try running the script by executing the command (from inside the directory holding the file):

   ```
   ./siteBackup.script
   ```

You should see the sql dump file be regenerated if it's working correctly.

Note: Now you can easily execute the command from a cronjob, and later add any additional commands to that script you want to run when backing up.

4 If you have access to linux machine, you can now use a sync command to ensure that your computer has a mirror of all the files on the production machine (or vice versa). For our examples we are assuming the files for the domain are stored a on the production machine at /var/www/siteToBackup/ and that the database is stored there under /var/sqlBackups/database_name.sql

To retrieve the latest backup our rsync commands might resemble:

```
#sync the files
rsync -avz -e ssh remoteuser@productionMachine: /var/www/siteToBackup/
/var/backups/siteToBackup

#sync the database file.
rsync -avz -e ssh remoteuser@productionMachine:
/var/sqlBackups/database_name.sql /var/backups/siteToBackup/
```

These command can also be entered into a script file, and made to execute forma cron job.

Note: There are many ways to transfer files to a server. Some shared hosts limit you to web interfaces, and others require FTP. In the event you do not have access to a server with ssh access and rsync installed, you can manually compress the files on the host into a .tar.gz or zip archive, and transmit the archive to the mirror server where it can be decompressed. Cloud backup solutions, or hosting within a redundant cloud eliminates some of these worries. It is often when doing backups that you see the limitations of simple shared hosting and consider migrating to better hosting.

5 You know you have completed the task correctly if you can list the contents of the local directory and see the files that were transferred there from the server. Make a change on the production machine file, and see if the sync retrieves the changed file.

A proper backup strategy automate these tasks putting together as automated jobs (cronjob). One script on the production machine would export the database, perhaps every hour, while the backup machine would execute a sync hourly as well. Those additional talks are left as a suggested task for the reader in their production sites.

Exercise 18.2 — Self signed x.509 Certificate

1 This exercise requires that you have access to a command line, and likely that you have

root access to install the certificate. In Lab 19 you will install the certificate.

2 Put the following into a script (or run each command sequentially).

```
# generate key
openssl genrsa -des3 -out server.key 1024
# strip password
mv server.key server.key.pass openssl rsa -in server.key.pass -out
server.key
# generate certificate signing request (CSR)
openssl req -new -key server.key -out server.csr
# generate self-signed certificate with CSR
openssl x509 -req -days 3650 -in server.csr -signkey server.key -out
server.crt
```

3 Now answer the questions in the prompts. The most important to consider is the Fully Qualified Domain name you are making the certificate for. In our case we are using *.funwebdev.com to generate a wildcard certificate.

The following is the output from when we ran the script for our domain:

```
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:CA
State or Province Name (full name) [Some-State]:Alberta
Locality Name (eg, city) []:Calgary
Organization Name (eg, company) [Internet Widgits Pty Ltd]: Fundamentals of
Web Development
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:*.funwebdev.com
Email Address []:rhoar@mtroyal.ca
```

4 Your certificate will now be a file named server.crt. If you examine the file (say using the cat command) you will see something like this:

```
-----BEGIN CERTIFICATE-----
MIICMTCCAZoCCQDGTEWUdvEEFDANBgkqhkiG9w0BAQUFADBdMQswCQYDVQQGEwJD
QTETMBEGA1UECBMKU29tZS1TdGF0ZTEhMB8GA1UEChMYSW50ZXJuZXQgV2lkZ2l0
cyBQdHkgTHRkMRYwFAYDVQQDEw1mdW53ZWJkZXYuY29tMB4XDTE0MDMyMzE4MTQx
N1oXDTI0MDMyMDE4MTQxN1owXTELMAkGA1UEBhMCQ0ExEzARBgNVBAgTClNvbWUt
U3RhdGUxITAfBgNVBAoTGEludGVybmV0IFdpZGdpdHMgUHR5IEx0ZDEWMBQGA1UE
AxMNZnVud2ViZGV2LmNvbTCBnzANBgkqhkiG9w0BAQEFAAOBjQAwgYkCgYEAp3am
jzW39nOMTZbb4LfFMpSEvFL6Mm11LAWy8NFHYQ4tg0brKjDXmtExjcvjbCnr8QBs
FXWiGhY/X6FPpDScouS83cm5/VnVhvHhWoz0Z1LljUgflkF1iKBcznOHIP3k7UOa
yZvuYQIq7YZc9ikdEu+QmsLCLODVfVscBP/4Ap0CAwEAATANBgkqhkiG9w0BAQUF
AAOBgQAtKV7fD1+gw0zLjKUcPyDPmJtksV98O7O09tsD0kac2fBNjyIveSZz1ax5
ard0JZUifJ8FDVHYFNfBgGFPKg6MtdWbYXMrYsnaje74Xd+JTaQYGv+M5nvIdgtK
```

```
iyXlV1FpdRc/9f7Qk1eLNrPIcSG2iPHtPy9tVCH+8wLZq9Ol+g==
-----END CERTIFICATE-----
```

We will make use of this certificate in Lab 19.

Exercise 18.3 — PHP Unit Tests

1. Using the basic Artist classes from earlier chapters you will now write unit tests using the PHPUnit framework available from: http://phpunit.de/. You will just scratch the surface of what is possible with this powerful framework. If you are using a IDE such as Eclipse there are plugins that can help you generate and then run Unit Tests trough the IDE, though that is left as an exercise.
2. Download and install the latest build of the PHPUnit Framework, following their directions. We used 4 commands on the command line to get started although you can download and install manually as well.
3. You have been provided with Classes, and automatically generated stubs for unit tests. To learn about generating your own stubs from existing classes check out The PHPUnit Skeleton Generator. Using that tool initial test files can be generated from the existing source files as follows:

```
phpunit-skelgen generate-test Art
phpunit-skelgen generate-test Artist
```

If you try to run the test with the following command on the command line:

```
phpunit ArtistTest
```

You will see that all the tests fail (which is good because we haven't written them yet), and each contains code that purposefully flags tests as incomplete as follows.

```
public function testGetFirstName()
{
  // Remove the following lines when you implement
  $this->markTestIncomplete(
    'This test has not been implemented yet.'
  );
}
```

Over the next few steps we will replace the empty tests with meaningful ones.

4. In these generated files you have function to setUp(), which instantiates the object you

are testing. tearDown() destroys and objects after your test, and can be especially useful for database or network connections where you want to free resources after each test.

```
    /**
     * Sets up the fixture, for example, opens a network connection.
     * This method is called before a test is executed.
     */
    protected function setUp()
    {
        $this->object = new Artist;
    }

    /**
     * Tears down the fixture, for example, closes a network connection.
     * This method is called after a test is executed.
     */
    protected function tearDown()
    {
    }
```

The remainder of the file contains stubs for functions to test each method in those classes, each function starts with the name test. The function to test our serialize method, for example, looks like.

```
    /**
     * @covers Artist::serialize
     * @todo   Implement testSerialize().
     */
    public function testSerialize()
    {
        // Remove the following lines when you implement this test.
        $this->markTestIncomplete(
          'This test has not been implemented yet.'
        );
    }
```

5 Add code to the setUp() method to actually define an artist. In this case we will use an example illustrated back in chapter 10. Your setUp() function changes to accept the parameters to the constructor.

```
protected function setUp(){
  $this->object = new Artist("Pablo","Picasso","Malaga","Oct 25, 1881","Apl
8, 1973");
}
```

6 Begin by writing tests for the easy methods, the getters. So long as our constructor works, we can assert that certain properties have certain values. A few such tests are shown below:

```
    /**
     * @covers Artist::earliestDate
     * @todo   Implement testEarliestDate().
```

```
     */
    public function testEarliestDate()
    {
        $this->assertEquals($this->object->earliestDate(),"Oct 25, 1881");
    }

    /**
     * @covers Artist::getFirstName
     * @todo   Implement testGetFirstName().
     */
    public function testGetFirstName()
    {
            $this->assertEquals($this->object->getFirstName(),"Pablo");
    }

    /**
     * @covers Artist::getLastName
     * @todo   Implement testGetLastName().
     */
    public function testGetLastName()
    {
            $this->assertEquals($this->object->getLastName(),"Picasso");
    }
```

Continue implementing tests for the setters as well. A few setters are shown:

```
    public function testSetLastName()
    {
        $this->object->setLastName("Testy");
        $this->assertEquals($this->object->getLastName(),"Testy");
    }


    public function testSetFirstName()
    {
        $this->object->setFirstName("Testy");
        $this->assertEquals($this->object->getFirstName(),"Testy");
    }
```

7 Remove the code to make the test fail in the testSerialize() method and instead write code that creates a situation where you can test if the method is working properly. In this case, the serialize method works, if the object can subsequently be de-serialized back to the same state and serialized again.

```
/**
     * @covers Artist::serialize
     * @todo   Implement testSerialize().
     */
    public function testSerialize()
    {
      $serializedArtist = serialize($this->object);
```

```
        $deserializedObject = unserialize($serializedArtist);
        //assert that all the fields are the same.
        $this-
>assertEquals($serializedArtist,serialize($deserializedObject));
    }
```

8 Optionally there are some powerful mechanisms you can make use of in PHPUnit. One of them is to relate tests to one another, since some of our tests depaend on the other test being successful (like testing the setter using the getter).

You can say that one test depends on the other by adding a @depends tag in the comment block. For example to show that testUnserialize() depends on testserialize we would add

```
    /**
     * @depends testSerialize
     * @covers Artist::unserialize
     * @todo   Implement testUnserialize().
     */
    public function testUnserialize()
    {
     //...
    }
```

/* * Implement (or wipe out) any remaining tests that are difficult to automate, and run the command from step 3 (`phpunit ArtistTest`), to see if the unit test passed. Eventually you should output similar to:

```
Time: 10 ms, Memory: 7.50Mb

OK (15 tests, 13 assertions)
```

Exercise 18.4 — System Monitoring

1 Log into a live server (preferably), and log in as root so we can look at system access files.

2 Depending on your configuration, log files will be in different locations. To examine login attempts on our ssh server we might type:

```
tail –n 100 /var/log/secure/
```

And see output like:

```
Mar 23 14:39:36 funwebdev sshd[10234]: Invalid user admin from 192.168.1.3
Mar 23 14:39:36 funwebdev sshd[10235]: input_userauth_request: invalid user
admin
```

```
Mar 23 14:39:38 funwebdev sshd[10235]: Connection closed by 192.168.1.3
Mar 23 14:39:43 funwebdev sshd[10236]: Accepted password for ricardo from
192.168.1.3 port 58633 ssh2
Mar 23 14:39:43 funwebdev sshd[10236]: pam_unix(sshd:session): session
opened for user ricardo by (uid=0)
```

3 Look for any interesting entries. You might consider using egrep to highlight invalid logins as follows:

```
egrep "invalid" /var/log/secure*
```

Which shows a list of many login attempts that were un successful.

```
...

/var/log/secure-20140323:Mar 18 09:15:39 funwebdev sshd[19287]:
input_userauth_request: invalid user a
/var/log/secure-20140323:Mar 18 09:15:42 funwebdev sshd[19286]: Failed
password for invalid user a from 124.160.12.198 port 50091 ssh2
/var/log/secure-20140323:Mar 18 09:15:50 funwebdev sshd[19292]:
input_userauth_request: invalid user postgres
/var/log/secure-20140323:Mar 18 09:15:52 funwebdev sshd[19291]: Failed
password for invalid user postgres from 124.160.12.198 port 53444 ssh2
/var/log/secure-20140323:Mar 18 09:15:54 funwebdev sshd[19294]:
input_userauth_request: invalid user postgres
password for invalid user postgres from 124.160.12.198 port 59909 ssh2
/var/log/secure-20140323:Mar 18 16:38:04 funwebdev sshd[19564]:
input_userauth_request: invalid user cgi
/var/log/secure-20140323:Mar 18 16:38:06 funwebdev sshd[19563]: Failed
password for invalid user cgi from 219.138.60.51 port 23503 ssh2
/var/log/secure-20140323:Mar 18 19:16:30 funwebdev sshd[19660]:
input_userauth_request: invalid user minepeon
/var/log/secure-20140323:Mar 18 19:16:32 funwebdev sshd[19659]: Failed
password for invalid user minepeon from 201.70.88.146 port 36342 ssh2
/var/log/secure-20140323:Mar 18 19:16:36 funwebdev sshd[19662]:
input_userauth_request: invalid user minepeon
/var/log/secure-20140323:Mar 18 19:16:37 funwebdev sshd[19661]: Failed
password for invalid user minepeon from 201.70.88.146 port 36595 ssh2
/var/log/secure-20140323:Mar 19 04:39:34 funwebdev sshd[20243]:
input_userauth_request: invalid user admin
/var/log/secure-20140323:Mar 19 04:39:36 funwebdev sshd[20242]: Failed
password for invalid user admin from 116.10.191.181 port 4547 ssh2
/var/log/secure-20140323:Mar 19 04:39:38 funwebdev sshd[20242]: Failed
password for invalid user admin from 116.10.191.181 port 4547 ssh2
/var/log/secure-20140323:Mar 19 04:39:41 funwebdev sshd[20242]: Failed
password for invalid user admin from 116.10.191.181 port 4547 ssh2
/var/log/secure-20140323:Mar 19 04:39:44 funwebdev sshd[20242]: Failed
password for invalid user admin from 116.10.191.181 port 4547 ssh2
/var/log/secure-20140323:Mar 21 05:35:51 funwebdev sshd[22468]:
input_userauth_request: invalid user default
/var/log/secure-20140323:Mar 21 05:35:52 funwebdev sshd[22467]: Failed
password for invalid user default from 62.103.39.67 port 7176 ssh2
/var/log/secure-20140323:Mar 21 07:16:56 funwebdev sshd[22537]:
input_userauth_request: invalid user httpd
/var/log/secure-20140323:Mar 21 07:16:58 funwebdev sshd[22536]: Failed
```

```
password for invalid user httpd from 62.103.39.67 port 8546 ssh2
/var/log/secure-20140323:Mar 21 08:56:12 funwebdev sshd[22588]:
input_userauth_request: invalid user nginx
/var/log/secure-20140323:Mar 21 08:56:15 funwebdev sshd[22587]: Failed
password for invalid user nginx from 62.103.39.67 port 11724 ssh2
/var/log/secure-20140323:Mar 21 09:56:55 funwebdev sshd[22650]:
input_userauth_request: invalid user fluffy
/var/log/secure-20140323:Mar 21 09:56:57 funwebdev sshd[22649]: Failed
password for invalid user fluffy from 202.53.174.178 port 35881 ssh2
/var/log/secure-20140323:Mar 21 12:14:37 funwebdev sshd[22764]:
input_userauth_request: invalid user apple
/var/log/secure-20140323:Mar 21 12:14:39 funwebdev sshd[22763]: Failed
password for invalid user apple from 121.196.16.38 port 42354 ssh2
/var/log/secure-20140323:Mar 22 08:21:56 funwebdev sshd[23742]:
...
```

4 Try to look at these failed attempts and do some inquiries as to who as trying to log in. Some spot checks might include doing a whois on the IP address of suspect login attempts.

```
whois 201.70.88.146
```

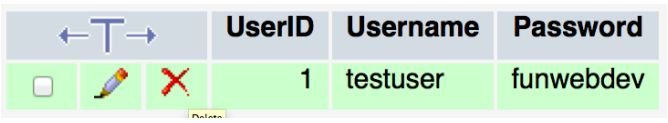
Tells us that thw attempt was assocaited with an IP address from Brazil, and with No employees working or travelling there, we now have an idea that there are people attacking your site and a call to your local police won't help.

Please, if you are running a production machine take it upon yourself to learn about installing and configuration a tool like: **blockhosts.py** (left as an exercise to the reader).

AUTHENTICATION

Exercise 18.5 — Build Better Authentication

1 Open, examine, and test lab18-exercise05.php. You will se that it contains a functioning authentication script for a user with the name "testUser" and password of "funwebdev" from back in Chapter 13. The script to create a simple database scheme is re-provided in lab13-exercise01.sql. Unfortunately, if you look at the database records, those fields are stored in plaintext, and so they are easily compromised by anyone with access to the database or database backup as illustrated in Figure 18.1.

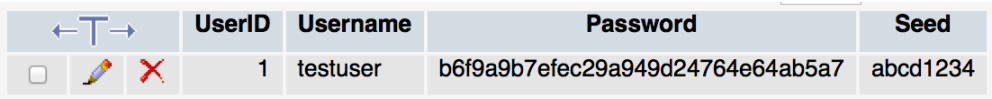


	UserID	Username	Password
	1	testuser	funwebdev

Figure 18.1 Credentials table and a record structure before changes

2 Modify the database table to hold an extra field, which should be a string and named seed. This can easily be done in phpMyAdmin.

3 Modify any records already in the table so that they store a unique seed and then use that unique seed when calculating the password. For the record with username password, the seed might be abcd1234 so you'd update the password to store md5(funwebdevabcd1234) as shown in Figure 18.2.



	UserID	Username	Password	Seed
	1	testuser	b6f9a9b7efec29a949d24764e64ab5a7	abcd1234

Figure 18.2 More secure database schema after modifications

4 Modify the **validLogin** function that checks for a valid login so that it also uses the seed in the validation check. Note that you only have to change the SELECT statement to make use of the seed in a sub query.

```
function validLogin(){
   $pdo = new PDO(DBCONNSTRING,DBUSER,DBPASS);
   //updated query to use the seed in checking login.
   $sql = "SELECT * FROM Credentials WHERE Username=:user and
Password=MD5(CONCAT(:pass,(SELECT Seed FROM Credentials WHERE
Username=:user)))";
   $statement = $pdo->prepare($sql);
   $statement->bindValue(':user',$_POST['username']);
   $statement->bindValue(':pass',$_POST['pword']);

   $statement->execute();
echo $sql;
   if($statement->rowCount()>0){
     return true;
   }
   return false;
}
```

Note: To see if your PDO queries are operating correctly, you need to access your MySQL log file which will have the "final" query after all bindings in PDO have occurred. Check my.cnf for the location of that file.

5 Now you have the basis for a secure authentication storage system. You should be able to log in as before, but now you are using the salted hash to validate logins. If you have any scripts to allow registrations, be sure to update them as well to include random seed creation, storage and usage as well.

Exercise 18.6 — Authenticate with Twitter

1. In the event of a change to the Twitter API, you should refer to https://dev.twitter.com/docs/auth/implementing-sign-twitter, for updated directions on the Twitter side.

2. Because Oath authentication is so common, there are libraries linked directly from Twitter's page https://dev.twitter.com/docs/twitter-libraries. Download and install the TwitterOAuth implementation from https://github.com/abraham/twitteroauth

   The file can be downloaded as a zip archive and extracted on a live server.

3. Now you must register as a developer on twitter and register an App to get the necessary secret tokens. https://apps.twitter.com/

   1. Login under the account you want to make the app
   2. Click "Create New App"
   3. Name the App and enter the necessary details.
      a. Be sure to set a value for the callback URL (it can be changed later)
      b. check the box to allow "user to log in with Twitter"
   4. Change permission on the new app to allow it to "Read and Write". This can be done in the permissions tab.
   5. Go to the API keys tab and click Generate Tokens. A set of unique values will be generated for you.

4. Unzip the source code from the TwitterOAuth onto a production server. This is important because to work correctly Twitter must be able to access the callback URL.

   Copy the config-sample.php file to config.php. Update the values to reflect the values from your twitter app that were generated in the last step.

```
define('CONSUMER_KEY', 'CONSUMER_KEY_HERE');
define('CONSUMER_SECRET', 'CONSUMER_SECRET_HERE');
define('OAUTH_CALLBACK', 'CALLBACK_URL_HERE');
```

5. In particular be cautious that the callback URL where your file is located is the same as the one in the twitter settings, and be certain that you are on a live site.

6. Connect to the test.php page, which should prompt you to log in, or recognize the credentials if you're already logged in. Once logged in you will see an output similar to that in Figure 18.3. From here you can see how you can access information to integrate the user with your site. Alternatively you can add the "Sign in with twitter" link anywhere

you want to let the user log in.

You can test our installation at http://examples.funwebdev.com/twitteroauth-master/

Figure 18.3 – Output from a successful Twitter OAuth login.

UNDERLYING THEORY

Exercise 18.7 — Modulo Arithmetic

1 This exercise requires pen, paper and calculator to help understand how modulo arithmetic is used.

Team up with a partner for this exercise, or use the suggested messages

2 Mutually agree to a prime number. Write that number down for all to see.

P=11

Now choose a generator for the cyclic group. A generator is a value such that when we raise g to the sequential powers we get all the values from 1, … p-1. On this case 2 makes a good generator for p = 11.

P=11, g=2

3 Now choose a random number a. Keep this number secret from everyone.

Calculate

$g^a \bmod p$

Note: If we chose a = 5. Then

$g^a \bmod p$ => $2^5 \bmod 11$ => $32 \bmod 11$ => **10**

Write your number down and pass it to your friend. It's ok if it's intercepted.

4 Take the value from your friend and raise it to the power of a (our secret)

Imagine our fiend passed us a value of 8, we would then take

$8^5 \bmod 11$=> $32768 \bmod 11$ => **10** $\bmod 11$

Remember modulo means the remainder when you divide a number by the mod value.

Note: Since you passed your friend 10, they will be calculating the same thing but backwards. Since powers are distributed the two numbers come out the same.

Since you would never transmit the key, continue working now with the symmetric key. If you want to be sure things are going right, ask your friend if they have the same value that you do.

5 Take a message and encode your message using a Caesar cipher. Pass the message to your friend, and receive one at the same time. Encode by adding the key from each character.

6 Decode the message by subtracting the key to each character

WHITE HAT SECURITY TESTING

All the exercises below make use of techniques that hackers use to compromise your sites. Therefore you should exercise caution that you are only using these techniques on websites and systems that you own. These skills are necessary to understand and protect your sites, but can land you in prison if used for illegal purposes.

Exercise 18.8 — Go Phishing

1 Masquerading as another person or phishing for information is illegal and should be avoided. This exercise illustrates some of the techniques that would go into a phishing email and then uses that example to illustrate how phishing can be detected in the wild.

This exercise assumes you have a working sendmail daemon running, so that` you can use the PHP mail command.

2 Open up lab18-exercise08.php. Note that this script will send an email to a placeholder email address every time it is run. Change the email address to an email you own and run the script to send yourself an email from president@funwebdev.com

3 The email may or may not arrive depending on your local spam filters. Since this email is masquerading as a domain you do not own, it would be reasonable for the SPAM filters to block the email from arriving.

4 If the email did arrive, we can see the path it took, and why it might not be valid. Examine the headers, where we can see which server received it and where it received it from (there will be a chain of these

```
Received: from verdande.mtroyal.ca ([142.109.1.33])
by notes2cluster.mtroyal.ca (Lotus Domino Release 8.5.3FP6)
with ESMTP id 2014032323365686-231947 ;
Sun, 23 Mar 2014 23:36:56 -0600
Received: from fakedomain.com (S0106602ad08217d8.cg.shawcable.net
[68.148.186.67])
by verdande.mtroyal.ca (8.13.1/8.13.1) with ESMTP id s2O5aq44000535
(version=TLSv1/SSLv3 cipher=DHE-RSA-AES256-SHA bits=256 verify=NO)
for <rhoar@mtroyal.ca>; Sun, 23 Mar 2014 23:36:52 -0600
Received: from fakedomain.com (hariseldon [127.0.0.1])
by fakedomain.com (8.13.8/8.13.8) with ESMTP id s2O5aqxk013059
for <rhoar@mtroyal.ca>; Sun, 23 Mar 2014 23:36:52 -0600
Received: (from apache@localhost)
by fakedomain.com (8.13.8/8.13.8/Submit) id s2O5aqDP013058;
Sun, 23 Mar 2014 23:36:52 -0600
Date: Sun, 23 Mar 2014 23:36:52 -0600
Message-Id: <201403240536.s2O5aqDP013058@fakedomain.com>
To: rhoar@mtroyal.ca
Subject: Exercise 18-8 Phishing Attack
From: president@funwebdev.com
X-Mailer: PHP/5.1.6
X-Greylist: Delayed for 00:19:31 by milter-greylist-4.4.3
(verdande.mtroyal.ca [142.109.1.33]); Sun, 23 Mar 2014 23:36:53 -0600 (MDT)
X-MIMETrack: Itemize by SMTP Server on Notes-2/Servers/MRC(Release
8.5.3FP6|November 21, 2013) at
03/23/2014 11:36:57 PM,
Serialize by HTTP Server on Notes-1/Servers/MRC(Release 8.5.3FP6|November
21, 2013) at 03/23/2014 11:54:11 PM,
Serialize complete at 03/23/2014 11:54:11 PM
X-TNEFEvaluated: 1
```

Look for headers identifying the domain it purports to be from. Often the email address does not match the host being sent (and some span filters catch this). You can often identify the server that originated the email, and perform WHOIs and other queries to get more info about the sender.

5 *Note: A real phishing scam is only a little more sophisticated than this super simple script. The HTMl in the email will be nicer, and the link will often hide the real url.*

Exercise 18.9 — Injection Tests

1. Open, examine, and test lab18-exercise09.php. Like Exercise 5 it is a simple login form. Unlike Example 5 it passes user input directly to the database making SQL injection possible.

2. Since you have likely modified your database to handle authentication more securely in Exercises you will have to revert to the previous, bad way of storing credentials as defined in the file lab13-exercise01.sql

3. Now to ensure the form is working properly log inwith *testUser* and password *funwebdev*. If successful the page is working as expected.

4. Now try to create a SQL query which will exploit the fact that the query values are being passed on in an unsanitized format.

   In the username field type a username and in the password field enter

   ```
   '; DROP TABLE Credentials;#
   ```

   This results in two queries, the first to test for login and the second to drop the table.

   If you run this query you will have dropped the Credentials table form the database altogether, thereby breaking logins for everyone, and possibly losing the data for good if the site was not backed up!

5. Since you just lost your table you will have to run lab13-exercise01.sql to recreate it.

   Now to fix the SQL injection vulnerability you have to handle the potentially bad user input by either sanitizing your user inputs or using prepared statements. Since we have illustrated prepared statements elsewhere, we will illustrate sanitized inputs. Modify the SQL statement provided in the file to the following:

   ```
   function validLogin(){
      $pdo = new PDO(DBCONNSTRING,DBUSER,DBPASS);
      //very simple (and insecure) check of valuid credentials.
      $sql = "SELECT * FROM Credentials WHERE Username='"
             .$pdo->quote($_POST['username'])."' and Password='"
             .$pdo->quote($_POST['pword'])."'";
      echo $sql;
      $statement = $pdo->prepare($sql);
       //The solution with prepared statements prevents injection as well!
      //$statement->bindValue(':user',$_POST['username']);
      //$statement->bindValue(':pass',$_POST['pword']);
      $statement->execute();
      if($statement->rowCount()>0){
        return true;
      }
      return false;
   }
   ```

6. If you add a echo statement to output the SQL command being run you will see that the

attempt was escaped and so the query did not drop the table. Instead that DROP query was tested as the password as is appropriate.

```
SELECT * FROM Credentials WHERE Username=''testUser'' and Password=''\';
DROP TABLE Credentials;#''
```

Exercise 18.10 — Cross Site Scripts

1. Open, examine, and test lab18-exercise10.php. This page allows users to click on links for every registered user to get access to information about them. Those links are then used to generate pages for that user (Note: for this exercise that functionality is not implemented),
2. Try clicking on the usernames and see that for each link a different message is displayed in the webpage identifying that this is the profile page for that person.
3. Now try editing the URL manually to make an alert box box up in lieu of a person's name as shown in Figure 18.4

   The url might resemble.

   lab18-exercise10.php?user=`<script>alert('bad');</script>`
4. Now with the knowledge that the page can be exploited using a script, a malicious user could craft a more sophisticated attack and use email to try and harvest data from logged in users.
5. To prevent the attack, user input must be distrusted. Using the HTMLPurifier library available from http://htmlpurifier.org/ we can add some code to our page so that reflected user inputs are first filtered and thus scripts are not executed.

```
include_once("htmlpurifier-4.6.0/library/HTMLPurifier.auto.php");
$config = HTMLPurifier_Config::createDefault();
$purifier = new HTMLPurifier($config);
$clean_name = $purifier->purify($_GET['user']);
echo "<h1>User: ".$clean_name."</h1>";
```

6. If you try to attack the site again the JavaScript alert will not pop up and thus JS can not be executed now.

   Note: this site was poorly designed in many ways, but it does allow you to see how to prevent user input from being used to execute random scripts. You should also be careful to apply these techniques to stored user data where a Stored XSS attack can be even more effective and devastating.

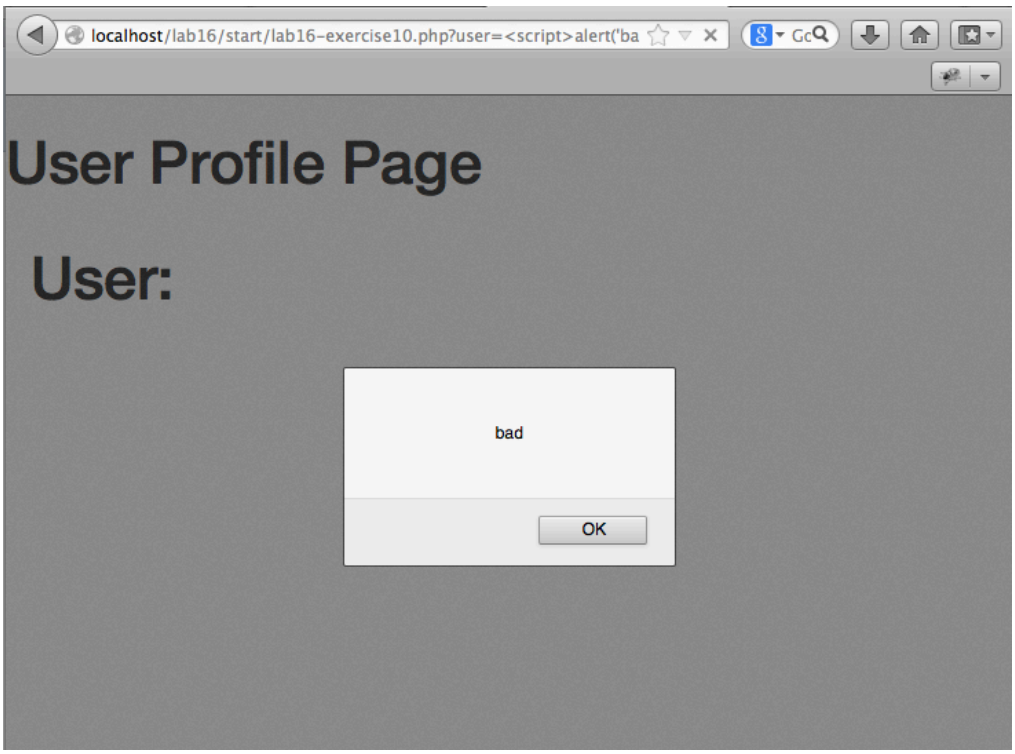


Figure 18.4 – Screenshot showing the script being executed based on the GET parameter!