**Introduction**

Before the testing process can be automated, it is necessary to have an understanding of how to manually uncover vulnerabilities. In order to achieve this, a few basic testing examples will be described. They will all be carried out without the assistance of pen-testing tools, using a purposely vulnerable web server.

For the purposes of these examples, a virtual machine hosting an Ubuntu 16.04 system will be used, but the basic procedure is applicable to other systems. The used vulnerable web server will be OWASP Mutillidae 2.6.43. It should be noted that the vulnerabilities found within this server are genuine, and should never be deployed outside a virtual machine.

**Injection**

An injection attack requires a field that accepts input data, and manipulating it in order to create an arbitrary query or command. The syntax will change depending on the interpreter. Error messages can help find what type of database the application uses, since many developers allow their applications to display error messages by default.

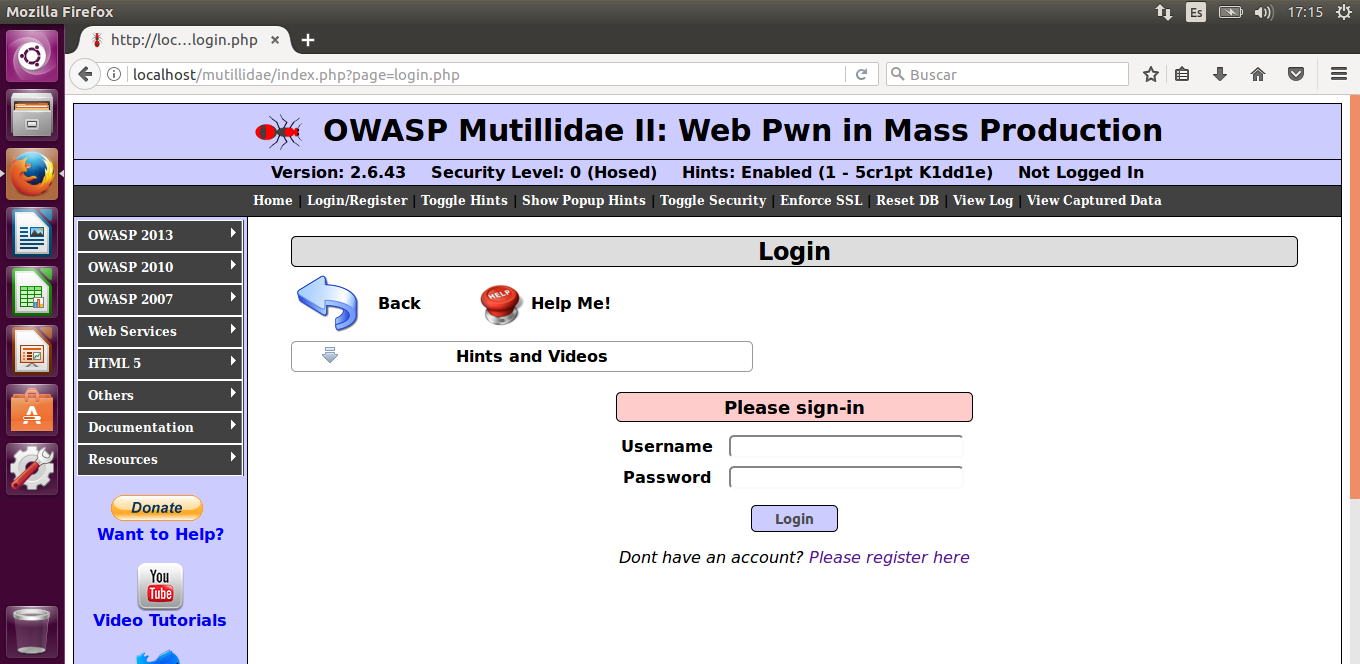
For example, let’s assume the server stores credentials in a MySQL table with this format:

|  |  |  |
| --- | --- | --- |
|  | **Username** | **Password** |
| **1** | User1 | Pass1 |
| **2** | User2 | Pass2 |
| **3** | User3 | Pass3 |

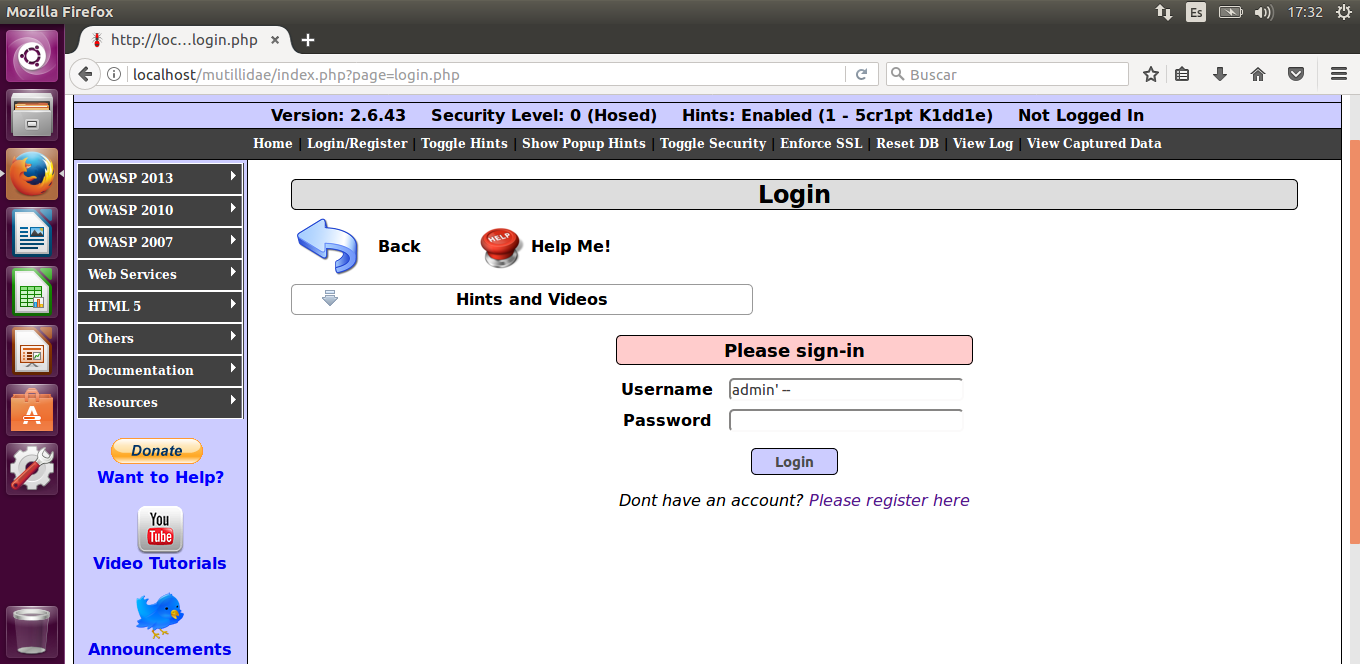
The table might contain any number of additional columns, and the passwords might or might not be encrypted or hashed. In order to retrieve information, the server will use MySQL queries to access the required information. For instance, inputting the username “admin” and the password “adminpass” might construct a query such as:

﻿SELECT \* FROM accounts WHERE username='admin' AND password='adminpass'

This will search through the ‘accounts’ table in the database for any row where both the username and the password correspond to the inputted data. The query is thus dynamically constructed, taking user input and sending it to the function as arguments. If the input is not sanitized, it’s possible to abuse this to force the server to execute an arbitrary query. An example of this might be a login page.



For instance, let’s assume that an attacker wants to log in as ‘admin’ without knowing the password. Since the input is passed on as arguments to the MySQL query, this would allow the attacker to modify the query. Knowing that comments in MySQL are inserted with a double hyphen (--) followed by a space, the attacker might input this in the username field:

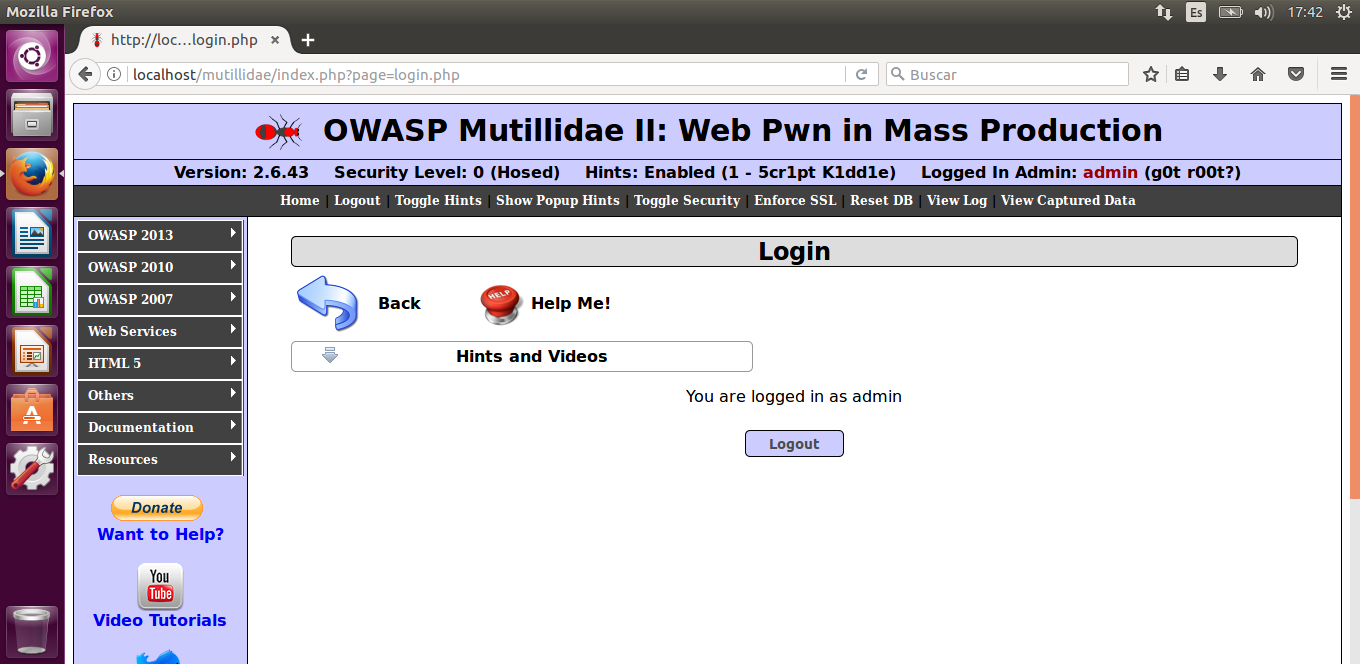


This will construct the following query:

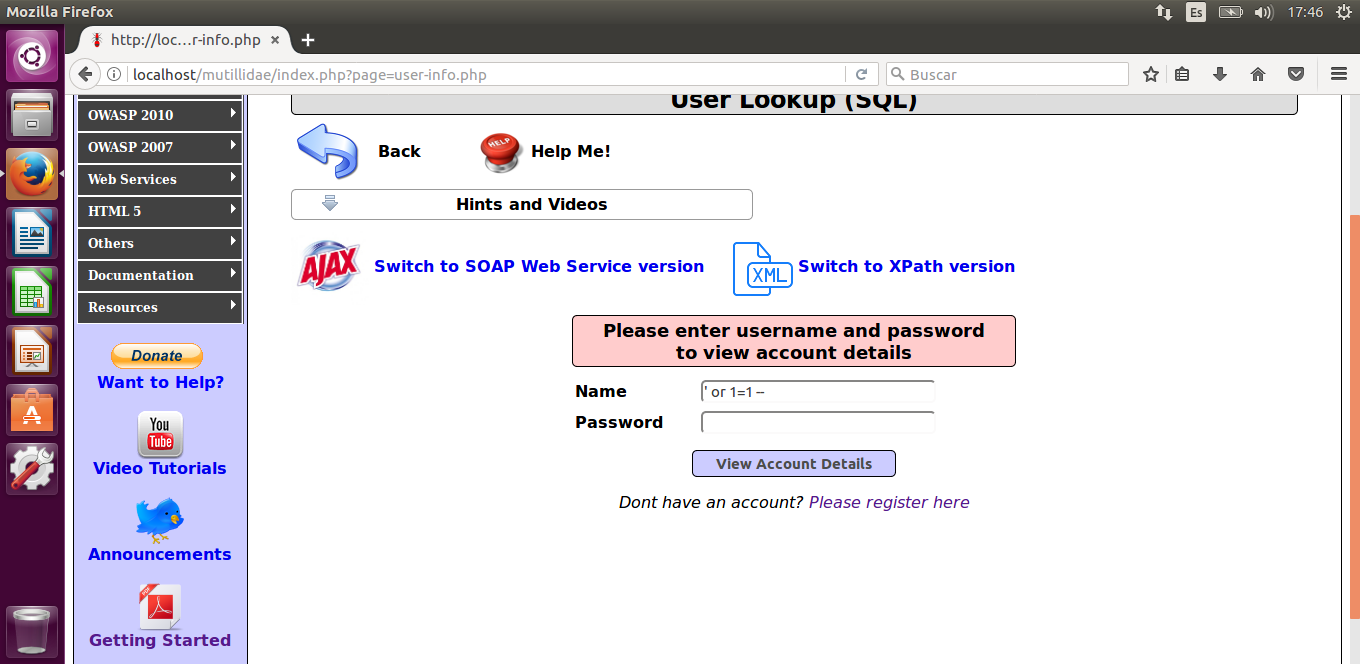
SELECT \* FROM accounts WHERE username='admin' *-- ' AND password=''* Since everything behind the double hyphen is commented out, this query is equivalent to:

SELECT \* FROM accounts WHERE username='admin'

This query will return the row with the username ‘admin’, regardless of the password, allowing an attacker to bypass authentication.



Expanding on this, it would be possible to force the server to dump the whole database by constructing a query that returns true for every row. For example, in a user lookup page, this input:



Would construct this query:

SELECT \* FROM accounts WHERE username='' OR 1=1 *-- ' AND password='adminpass'*

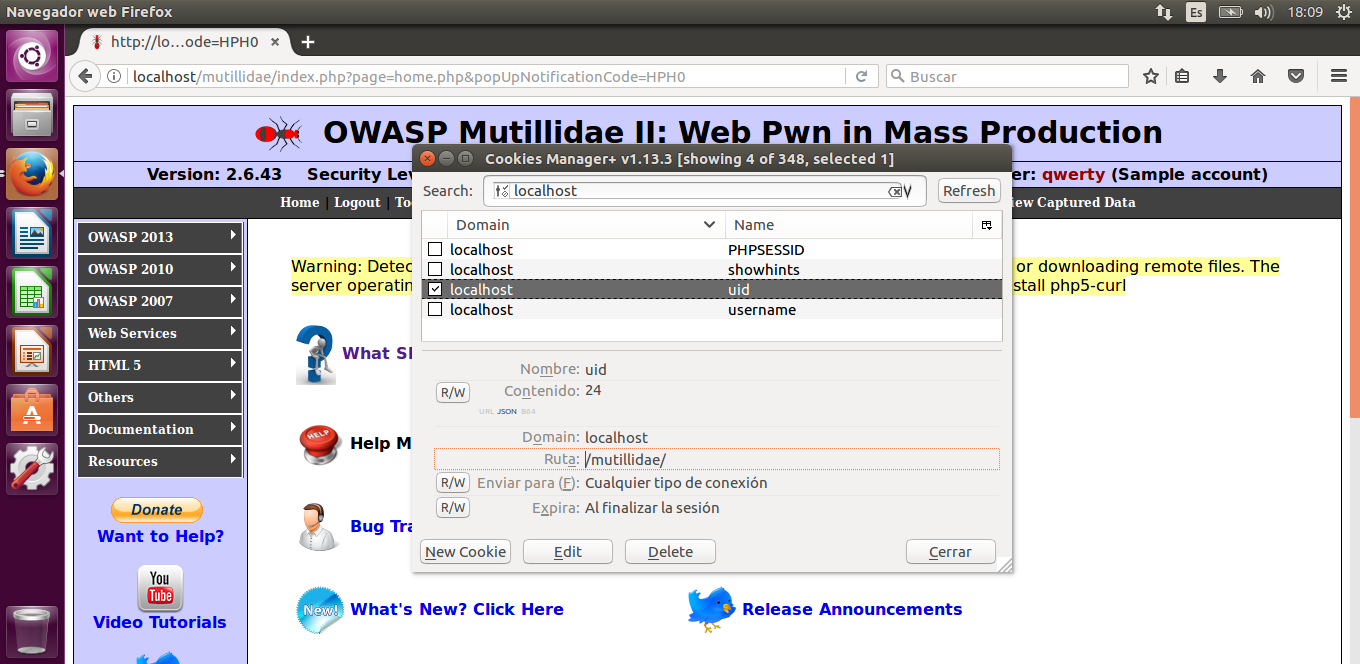
Ignoring the commented out part, this query will return every row where either of the two conditions evaluate to true. By inputting a trivial second condition that will always be true (such as 1=1), this query will return every row in the database, dumping the credentials of every user.

While these examples have been done in MySQL, the methodology is identical for other different interpreters, varying only in the syntax used to construct queries.

**Broken Authentication and Session Management**

There are multiple ways to hijack sessions depending on how the particular target web application is configured. As mentioned before, in an insecure page injection can be used to extract password lists or even bypass authentication altogether, but there are other ways to achieve this.

Depending on how a page works with cookies, they could be exploited to obtain administrative privileges or impersonate another user. There are many ways to edit cookies. For the purposes of this example, the ‘Cookies Manager+’ add-on for Mozilla Firefox will be used. First, it is necessary to understand how the target web application manages cookies. An easy way to begin is to actually register as a normal user, in order to make the web application generate a genuine session cookie.



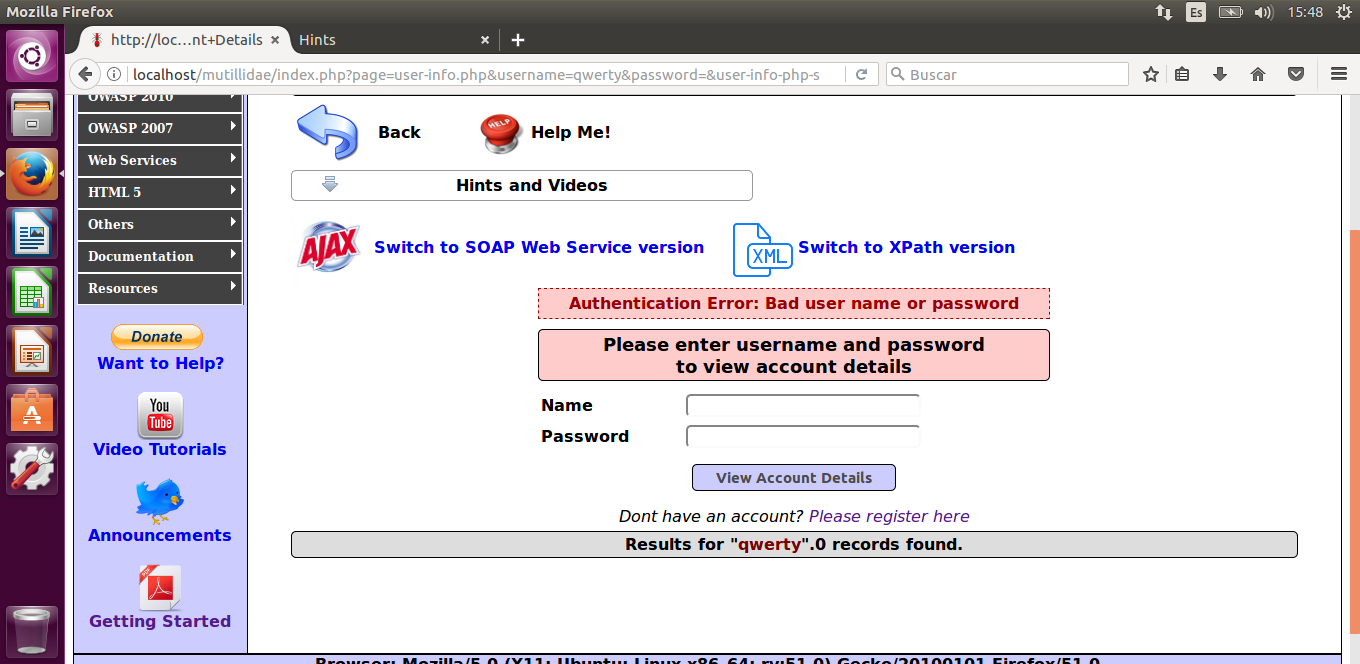
Analyzing the newly generated cookies can give insight on how to proceed. For this particular example, registering as a new user has created two new cookies, ‘uid’ and ‘username’, corresponding to the user’s id and the username. In most cases the relationship between the cookies and the session will not be as obvious, so it might be necessary to create more than one account to compare the cookies and see how they relate to their respective sessions. For this example, trial and error shows that changing ‘uid’ to 1 and ‘username’ to admin is equivalent to logging in as the web’s administrator.



**Cross-Site Scripting (XSS)**

Much like injection, cross-site scripting attacks make use of fields that allow the user to input data and then display it back in the output without proper encoding. Two variations exist, reflected XSS and stored XSS.

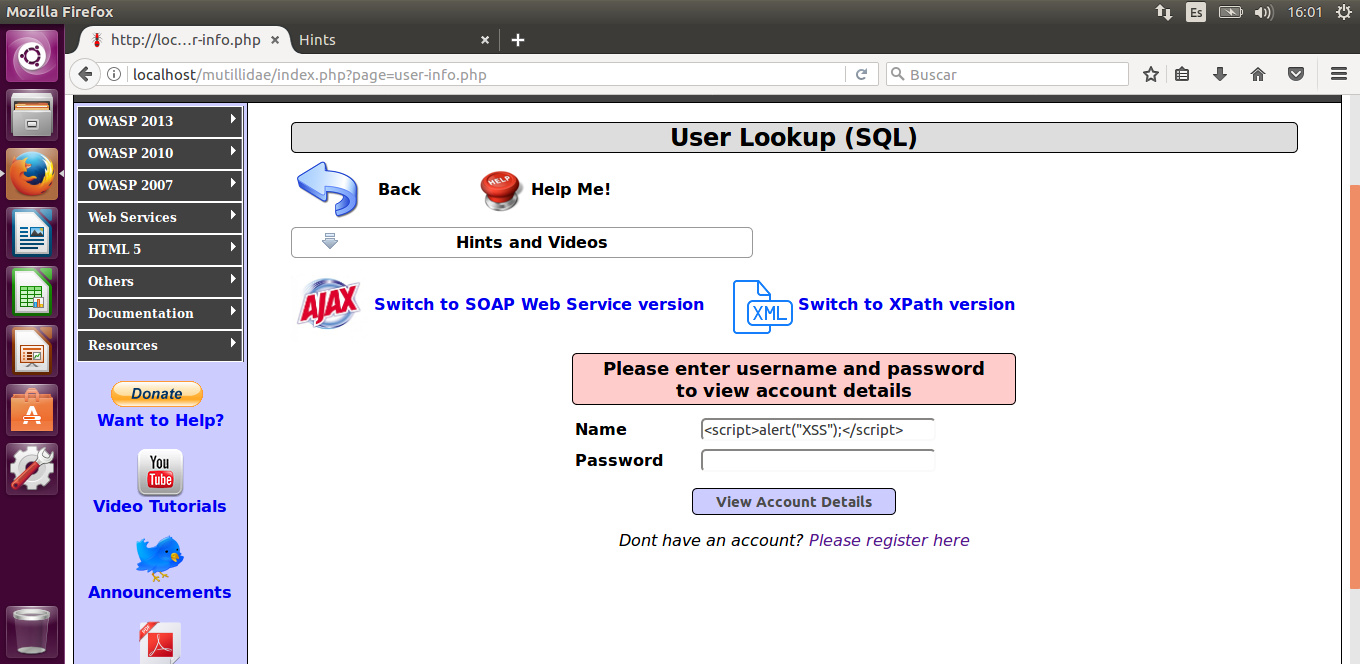
Reflected cross-site scripting is transient and is not stored within the server. Any page that directly outputs the user input without sanitizing it is vulnerable. For example, let’s assume a user lookup page takes in a username and a password and then displays the username back without sanitizing it. For instance, inputting ‘qwerty’ would return:



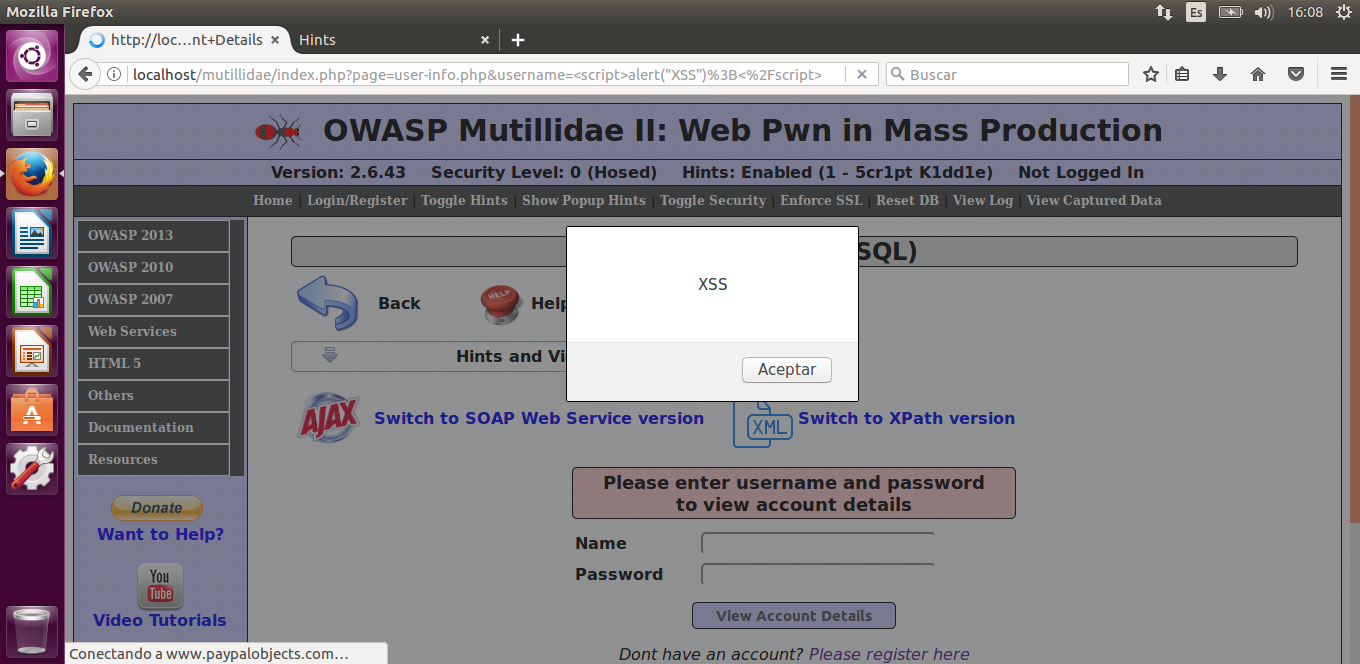
If the developer is careless and passes on the parameters directly through the URL, the URL might look like this:

*…/mutillidae/index.php?page=user-info.php&****username=qwerty****&password=123&user-info-php-submit-button=View+Account+Details*

If that is the case, it would be possible to input a malicious script in the username field in order to generate a malicious link that will execute that script when accessed. The attacker could then make use of social engineering to trick a victim into clicking such a link and executing the script. A very simple example of this could be a script that displays a warning message:



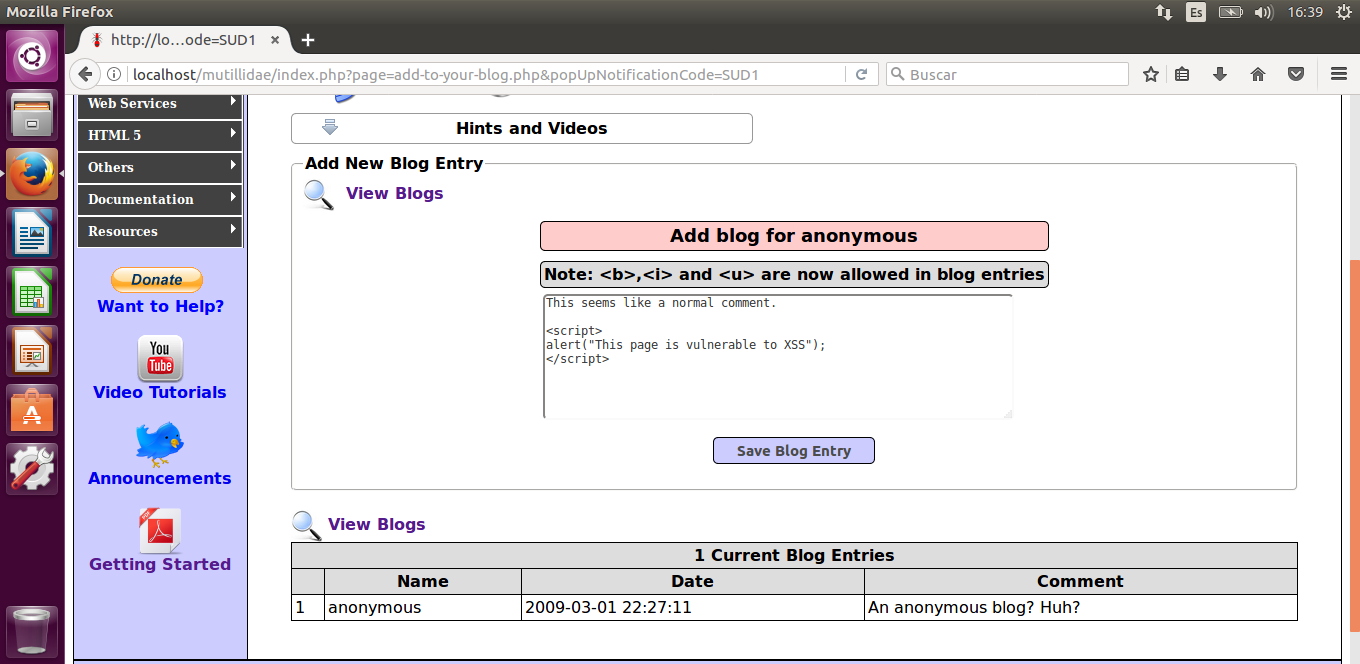
This would generate the URL: *…/mutillidae/index.php?page=user-info.php&****username=%3Cscript%3Ealert%28%22XSS%22%29%3B%3C%2Fscript%3E****&password=&user-info-php-submit-button=View+Account+Details*. The apparent garbling in the URL happens because all non-alphanumeric characters are substituted by their equivalent ASCII codes (such as ‘<’ becoming %3C). Upon accessing this URL, the script is executed and the warning message is displayed.

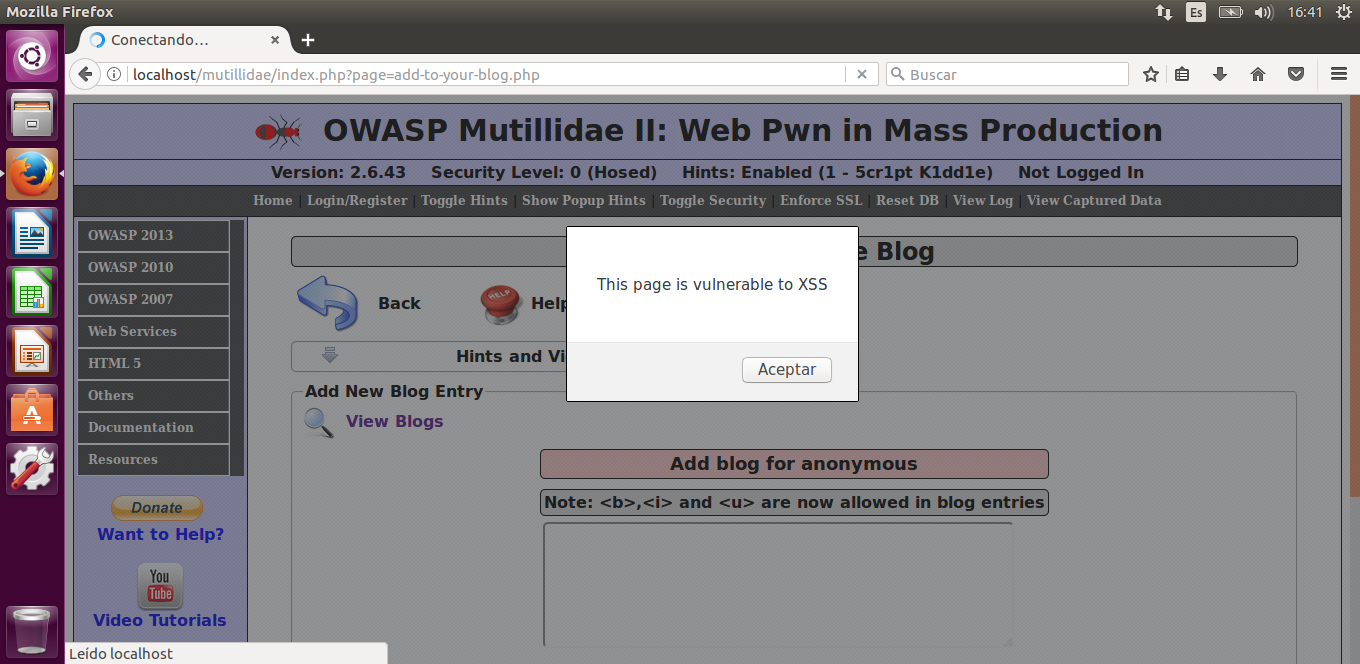


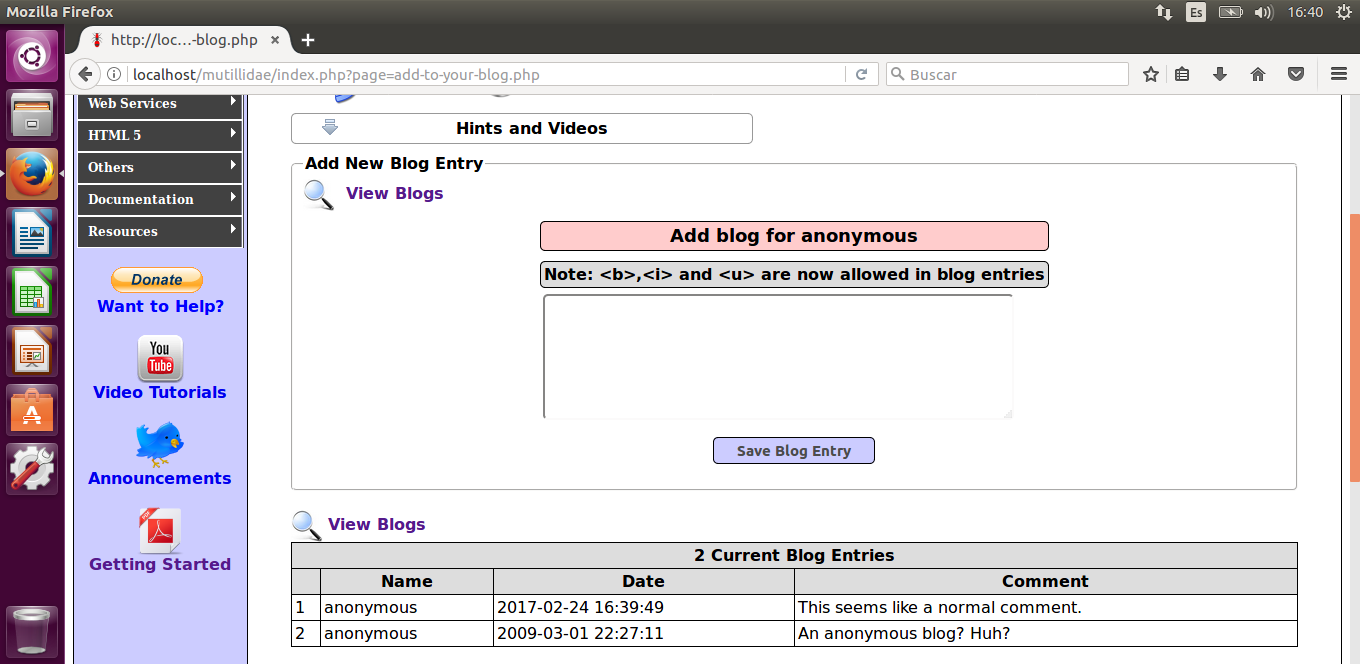
While this particular example is benign, an attacker could force the execution of any arbitrary script of their choice using this method.

Stored cross-site scripting differs from the reflected variant in that it doesn’t need to target a specific victim. Instead, as its name implies, the script is directly stored in the server and executed whenever any user accesses that page or performs a specific action. Pages vulnerable to stored XSS include all those that allow users to input data that is stored in the server without sanitizing it first. Common examples are forums, blogs or guestbooks.

An example could be a blog that allows users to leave comments that are then displayed to every other visitor. If the inputted comment was a malicious script, every visitor would accidentally execute it upon visiting the page.







This entry would be added as a permanent comment, and in most cases it would look no different than a normal comment, but it would execute the script whenever a user visits the page. Subtler scripts could execute without the user noticing anything wrong. The only way to solve this would be for the webmaster to delete the malicious entry from the server. In particularly pernicious cases with self-replicating scripts, the only solution might be to restore the database to a previous backup or reset it altogether.

**Insecure Direct Object References**

Insecure direct object references will take place whenever an application tries to access a resource using the resource itself as the reference to look up for. Through trial and error, it could then be possible to gain access to a confidential file without having the required privileges.

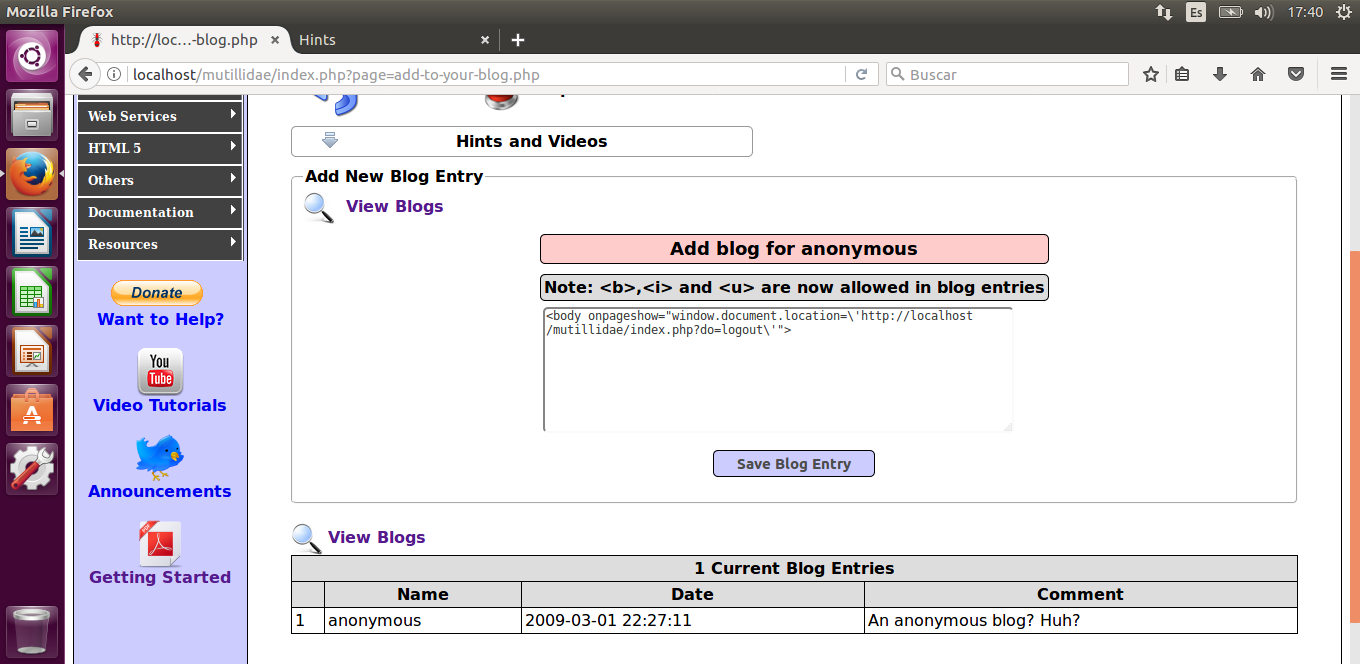
If the web application has a folder called ‘passwords’ that contains a text file of the name ‘accounts.txt’, it could be possible to access it directly through the URL */passwords/accounts.txt* ,even if the text file itself isn’t accessible through normal means in the webpage. Another example would be a page that shows the user’s account directly through the URL as a parameter, allowing attackers to just modify that parameter to access any account, such as */userInfo?account=****admin***.

**Cross-Site Request Forgery (CSRF)**

Before this kind of attack can take place, the target must be authenticated. Then, the attacker runs a script into the target’s browser to forge a request. The website has no way to know that the request isn’t genuine, since it comes from the user’s browser.

Cross-site request forgery can happen in multiple ways, and can be combined with many of the previous attacks. The final objective is to make the target’s browser execute a script, and this can be achieved by many different ways such as sending an URL or XSS.

As an example, in a forum or comment page that doesn’t sanitize input an attacker might use cross-site scripting to post an entry with the following HTML code that forces any visitor to logout:



These kinds of attacks are not effective if the user isn’t authenticated beforehand. Other possible approaches for attacks could include forcing a user to make a purchase or sending money to the attacker’s account.