00103: Analyze SQL Injection Attack

Objective

- Use Wireshark to analyze network traffic and identify targets
- Use Wireshark to identify methods used by the attacker
- Analyze exploit to determine specific attack vector and vulnerabilities exploited

Scenario

The Corporate Information Assurance Division has received a packet capture showing some suspicious behavior occurring on the network between the corporate SQL database server, a mystery web page and an unknown attacker. You have been called to analyze this capture and to help figure out what might be going on.

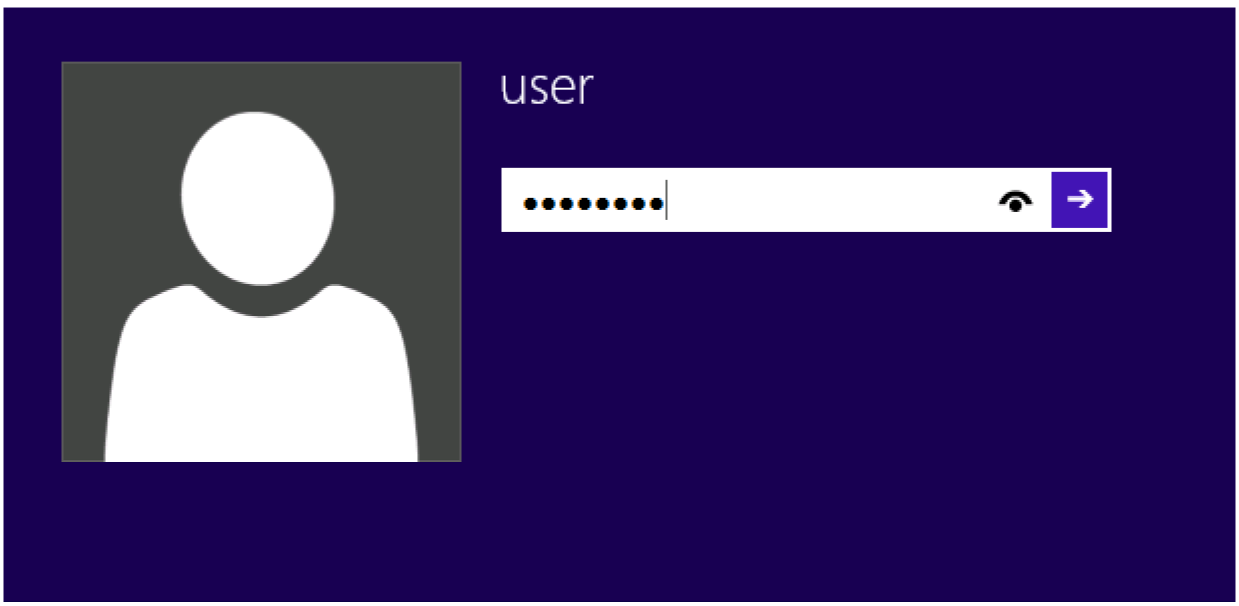
Analyze Network Traffic

Scenario

Students will begin by looking at the network traffic capture provided by the Corporate Information Assurance Division. The student will then analyze the given traffic capture to identify an SQL Injection.

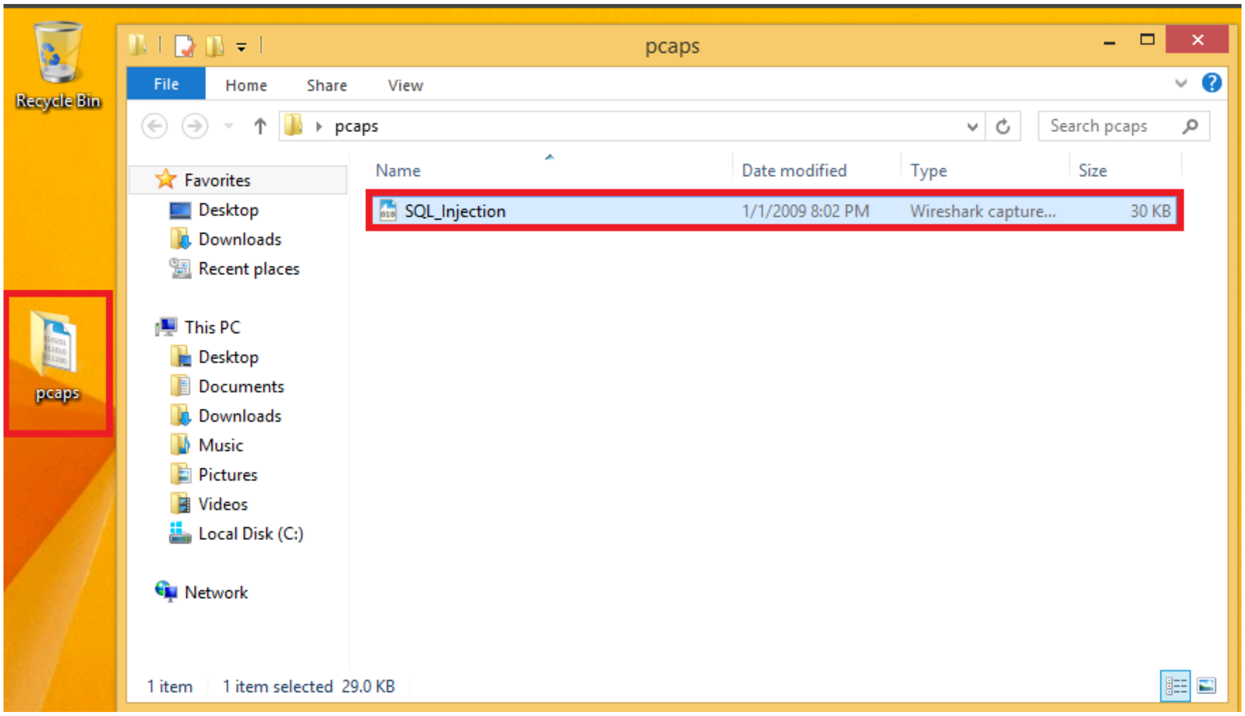
☐ 1. Log into the **Windows 8** machine using the username **user** and the password **password**.

> ⚠ This lab is equipped with an auto-scoring technology designed to track your progress as you successfully complete the lab. A short-cut is available on the Desktop to check your score. Double-click on the short-cut and a web browser window will open displaying your current score. You can periodically refresh the page as you complete more of the lab to see your current score.
>
> The final score is calculated and recorded when you either complete or cancel the lab. If you save your lab, the score is held until you resume the lab and cancel or complete it.



Switch to Win 8.1

☐ 2. Open the **pcaps** folder on the Desktop and select the file called **SQL_Injection.pcap** located inside. Wireshark will open the file for analysis.

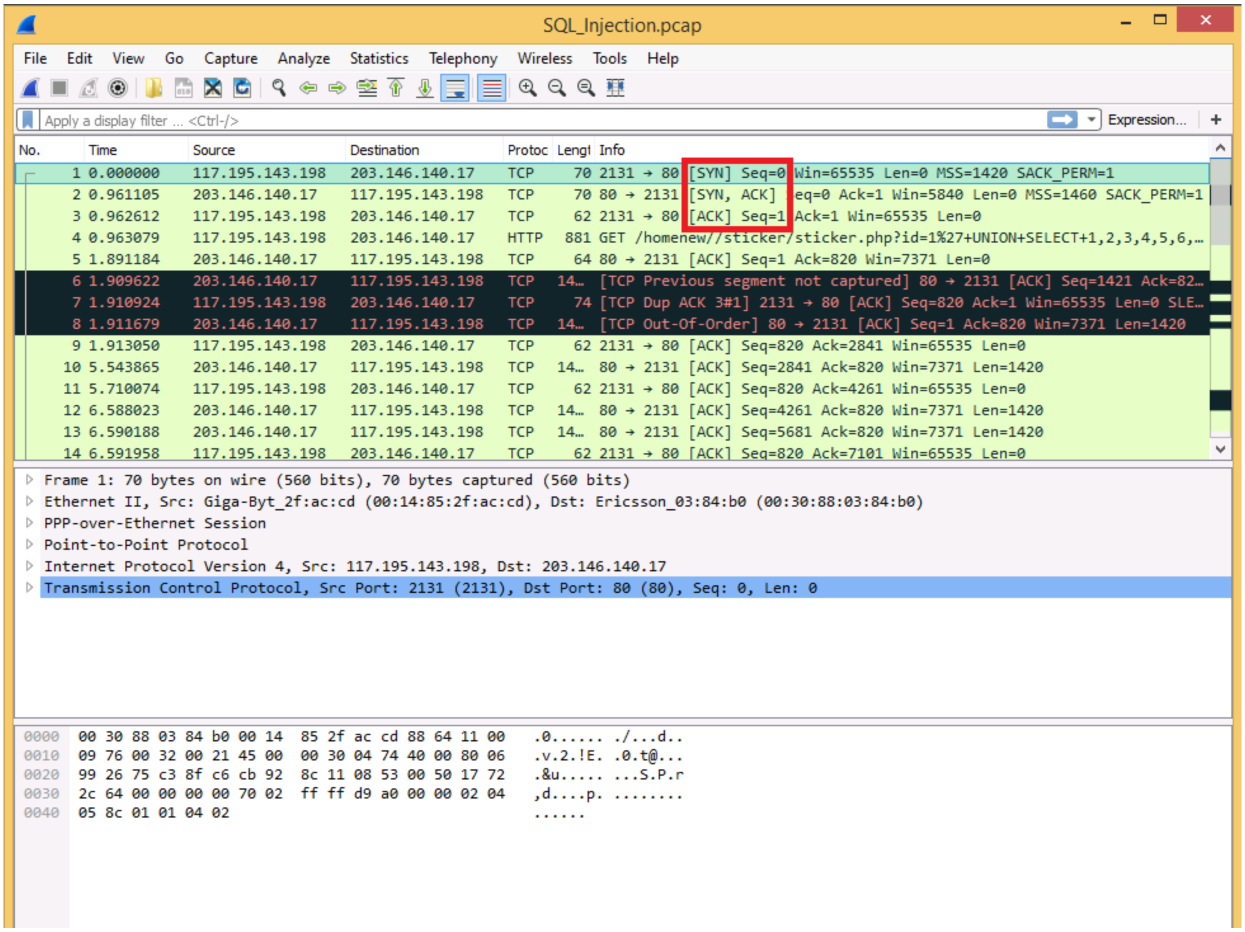


Switch to Win 8.1

☐ 3. This traffic capture is very short. In the first few frames, we see a 3-way TCP handshake occur.

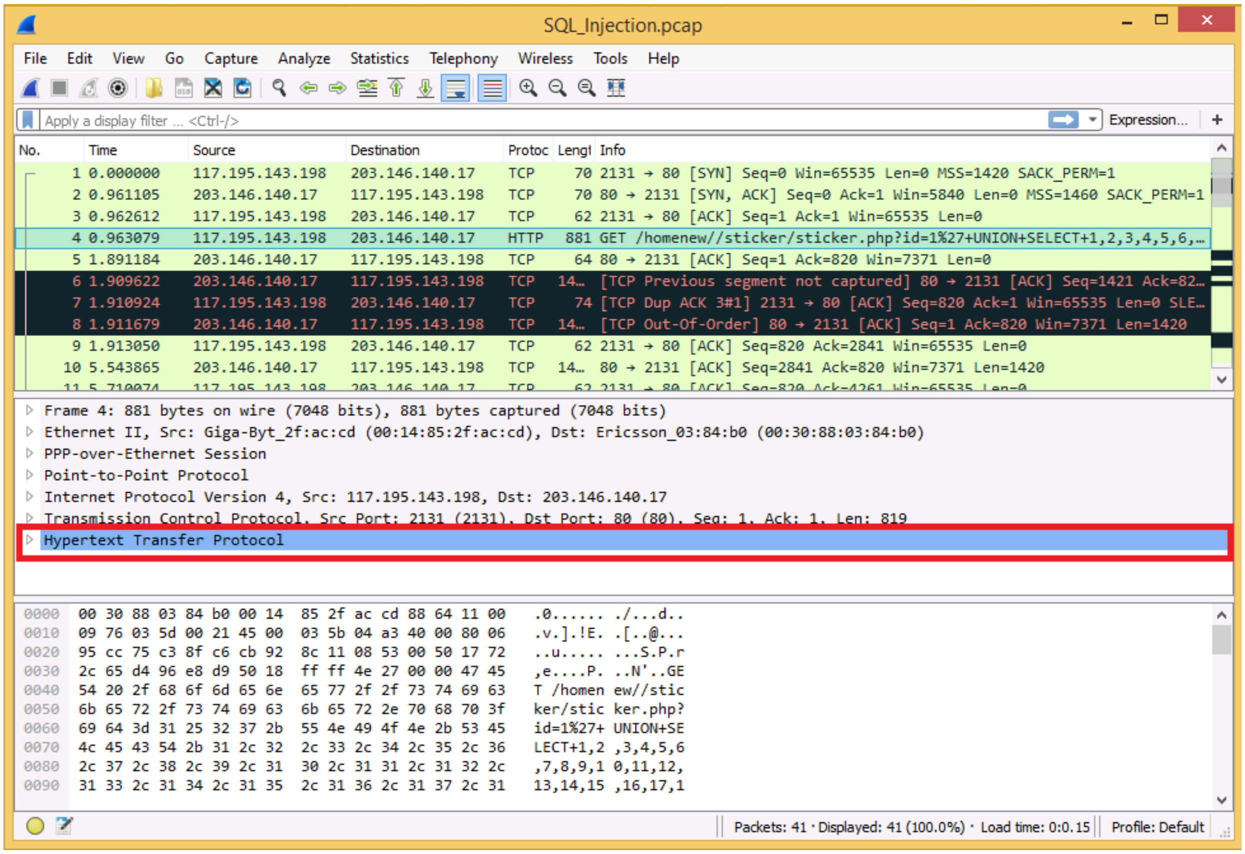
SYN > SYN/ACK > ACK

This sets up our reliable TCP connection for follow-on actions. In the next step, we will look at an **HTTP GET** request for a URL that contains an SQL Injection attack.



Switch to Win 8.1

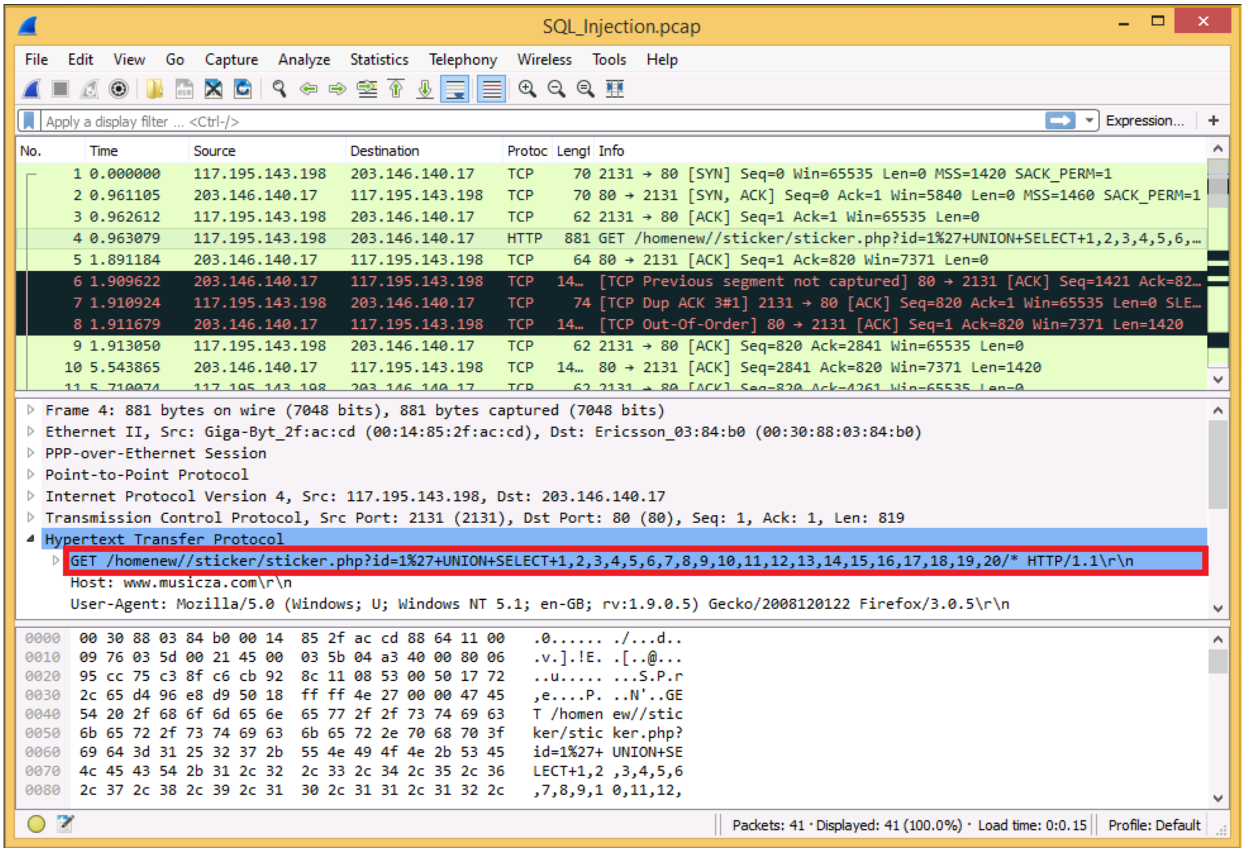
☐ 4. Click on packet **#4.** Open the **Hypertext Transfer Protocol** subtree in the middle frame by clicking the sideways triangle.



Switch to Win 8.1

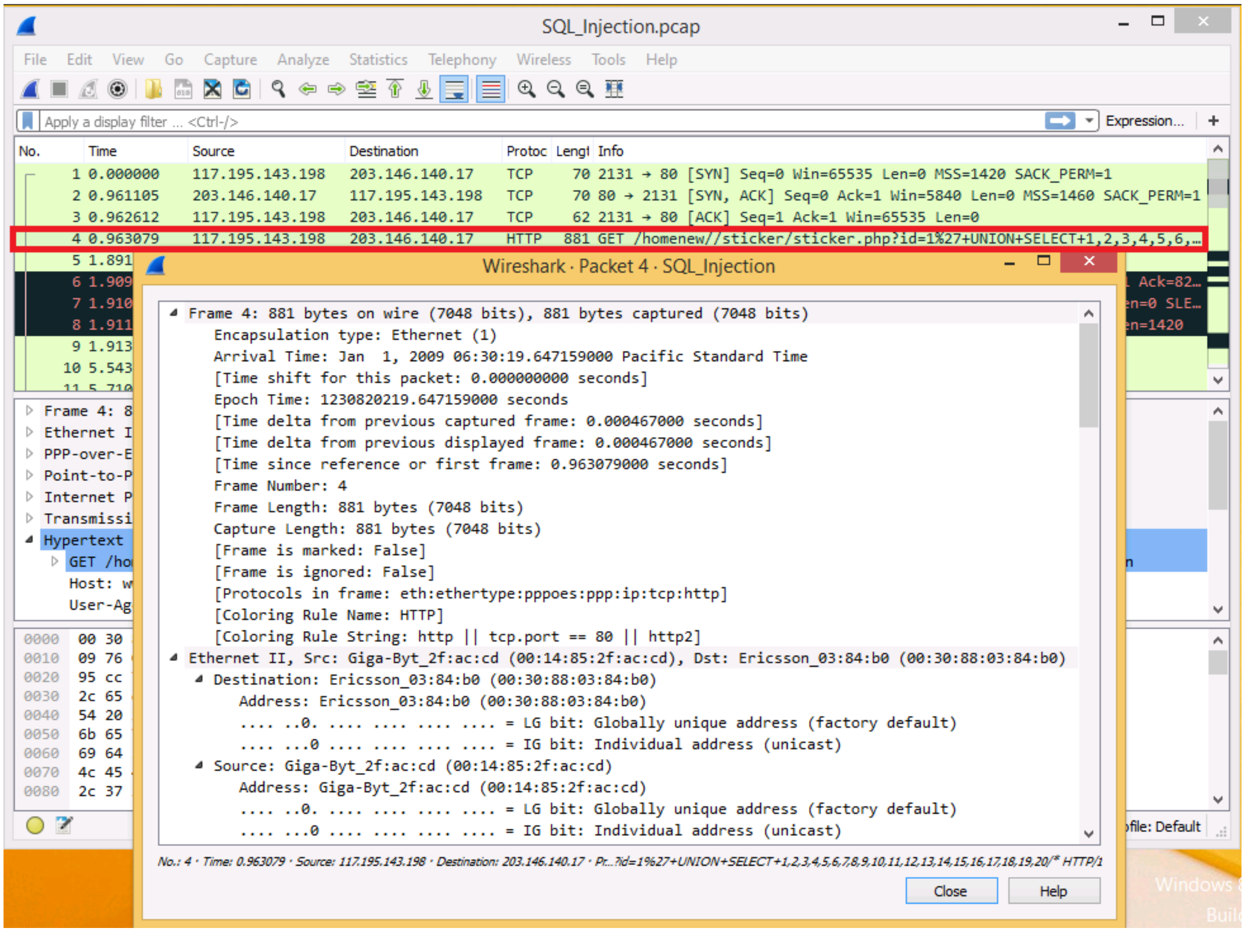
☐ 5. We see the following GET request for the resource **/homenew//sticker/sticker.php?id=1%27+UNION+SELECT** and a whole list of numbers.

After a client browser sends an HTTP GET request to a web server for a specific resource or page as we've seen here, the web server will respond with a status code and dynamically generate the client-side code (HTML, Javascript) and push it back over the TCP connection to the requesting entity.



☐ 6. Double click on **packet #4** and open it in a new window. Spend a few minutes digging into this specific packet and try to understand its various elements.

In the next step, we will look at the actual exploit downloaded from the web server. The exploit will be in the form of an HTML page that was used to conduct an SQL Injection attack against the victim website.

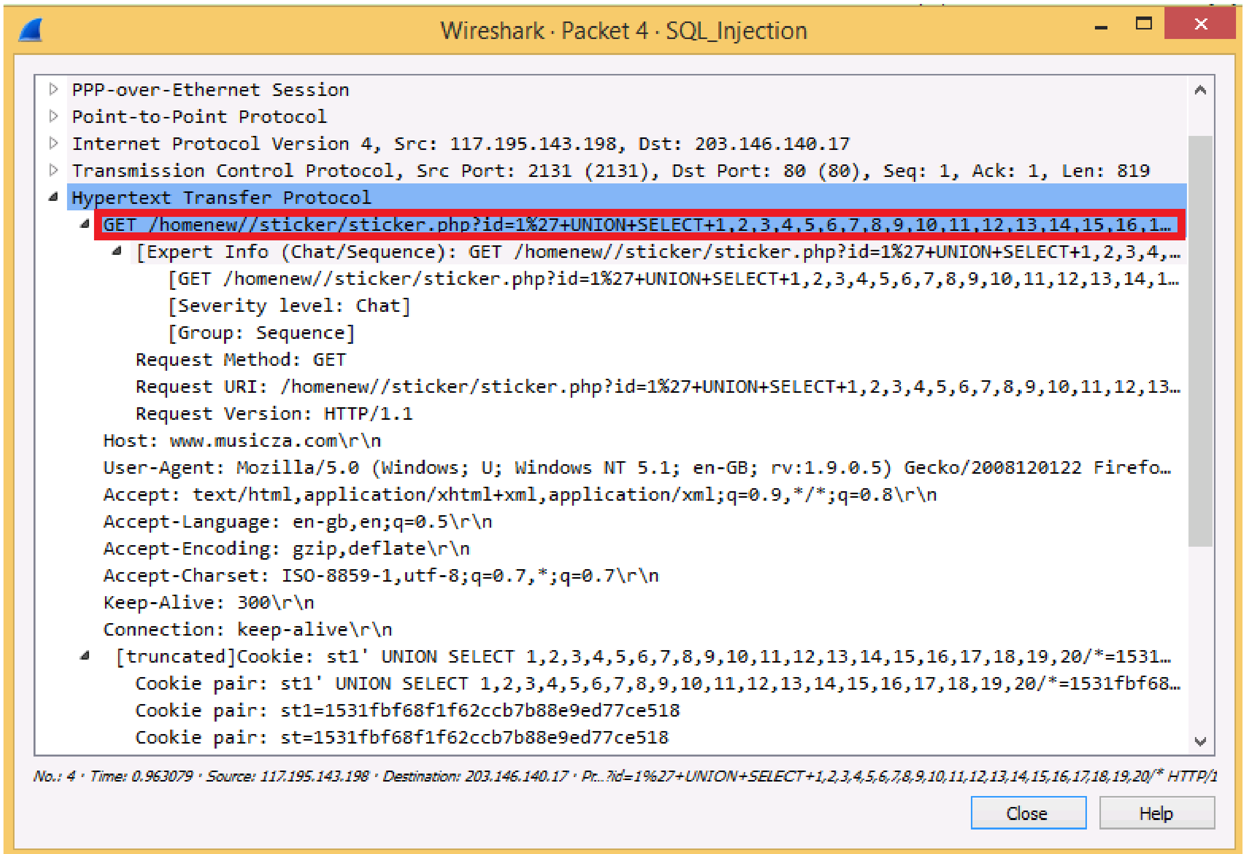


☐ 7. In the new **Packet #4** window, scroll down until you reach the following section: **Hypertext Transfer Protocol**

Pay specific attention to the very next line which shows the **HTTP GET** request that was sent to the victim web server in an attempt to conduct an SQL Injection attack.

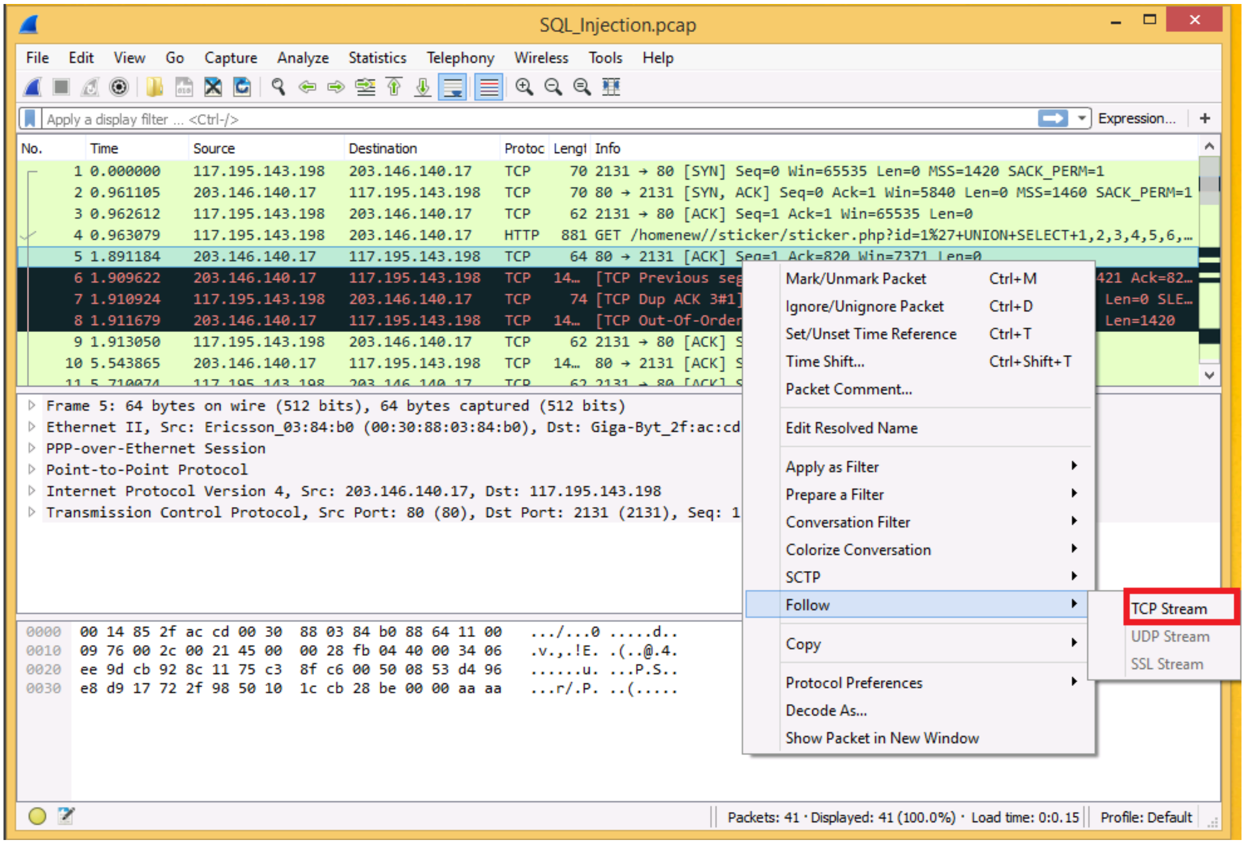
The **%27** is a URL-encoded single quote (') which is the simplest way to conduct an SQL Injection attack. By sending a URL-encoded single quote input into a web application that accepts user input and interacts with a backend database, an attacker can try to break the SQL database's parsing engine and cause an error by making the number of single quotes it uses for delimiting data uneven. After sending the vulnerable SQL database a single quote, an attacker will frequently also send additional legitimate SQL queries in order to get the database to disclose internal information.

When you're done reviewing the information, close the window and go back to the main Wireshark interface.



Switch to Win 8.1

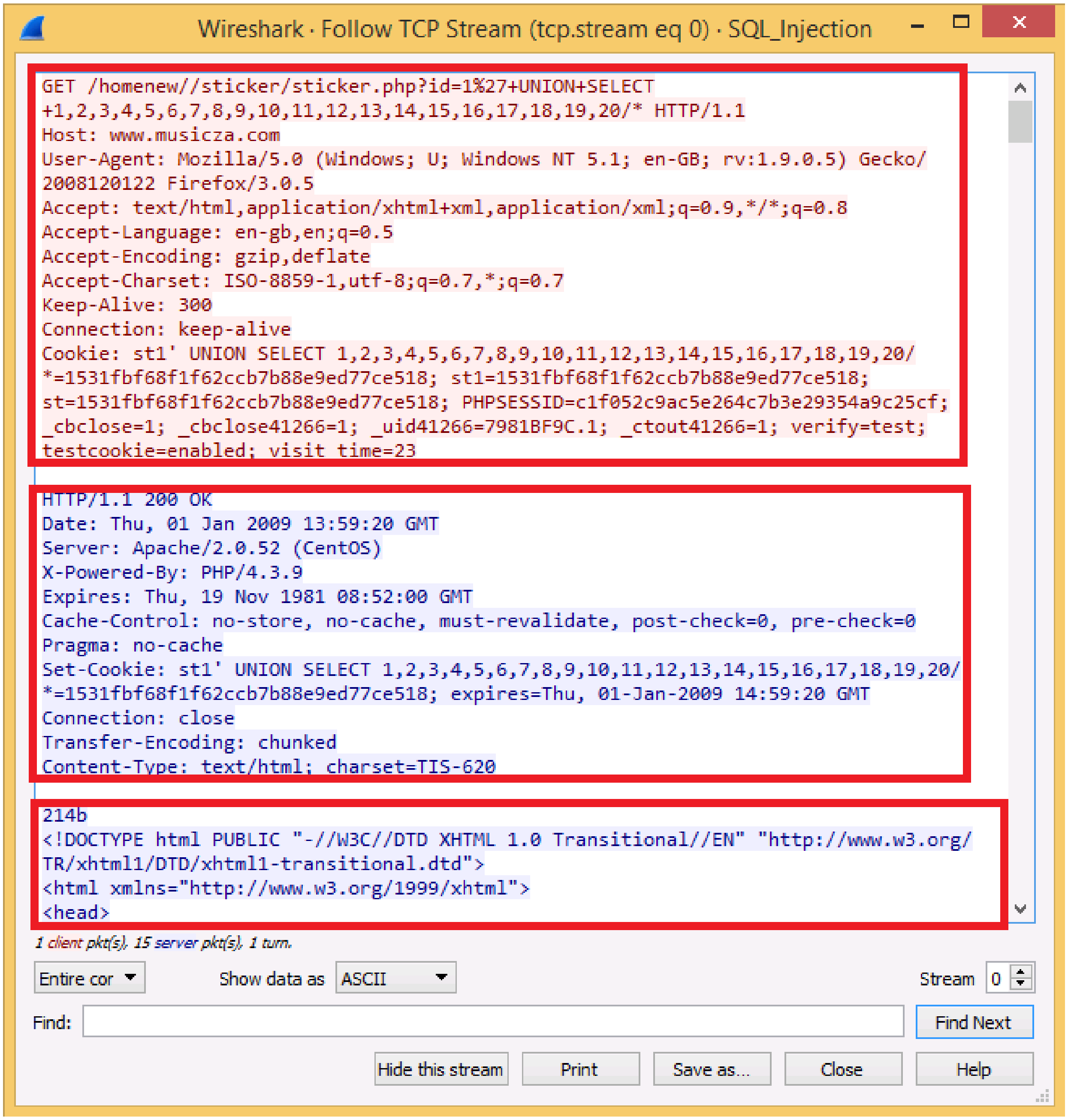
☐ 8. We are going to create a single TCP session from all the packets and view it in its entirety. To do this, click on any packet in the main Wireshark window, then right click and select **Follow TCP Stream**. A new window will pop up. In the next step, we'll begin to analyze the TCP Stream



Switch to Win 8.1

☐ 9. In the **Follow TCP Stream** window that just opened, we see an initial **GET** request for the resource on the victim web server and the embedded **SQL Injection** attack in the URI.

After that, we a see a **200 OK** response code followed by some HTML and Javascript. This code was generated by the web server and sent back in response to the GET request. The request itself begins with **214b**, as seen shortly before the head of the HTML document.



Switch to Win 8.1

☐ 10. Let's look at that Javascript:
Nothing strange here as we see some basic HTML and some Javascript, as annotated in the screenshot. There is also a Javascript pop-up window function near the bottom.

Let's keep looking and see if the SQL Injection attack worked and whether the database server responded with any information it should not have.

Script Tag - Signals beginning of Javascript

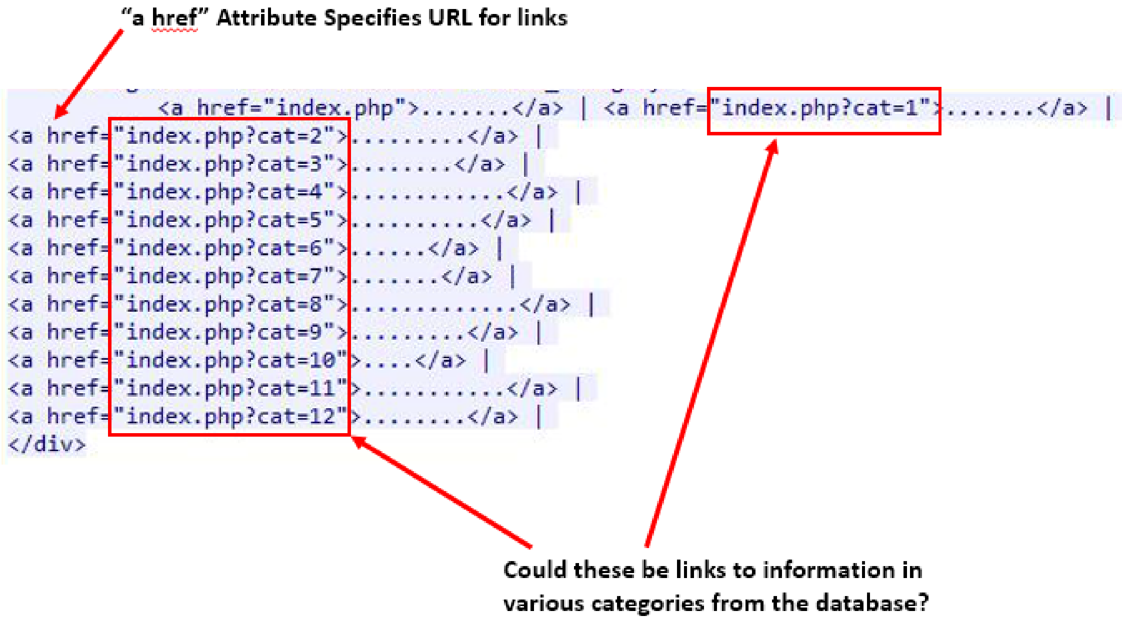
```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/
TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=windows-874" />
<title>Musicza  Sticker Extreme edition</title>
<link href="style.css" rel="stylesheet" type="text/css" />
<META HTTP-EQUIV="Pragma" CONTENT="no-cache">
<meta name="Keywords"
content="free,thailand,thai,sticker,album,musicza,tosdn,php,mysql" />
<script language="JavaScript" type="text/JavaScript">
<!--
function MM_reloadPage(init) {  //reloads the window if Nav4 resized
  if (init==true) with (navigator) {if
((appName=="Netscape")&&(parseInt(appVersion)==4)) {
    document.MM_pgW=innerWidth; document.MM_pgH=innerHeight; onresize=MM_reloadPage;
}}
  else if (innerWidth!=document.MM_pgW || innerHeight!=document.MM_pgH)
location.reload();
}
MM_reloadPage(true);
function MM_openBrWindow(theURL,winName,features) { //v2.0
  window.open(theURL,winName,features);
}
function setsmile(what)
{
            document.Postcomment.CommentText.value =
document.Postcomment.elements.CommentText.value+" "+what;
            document.Postcomment.CommentText.focus();
}
function PopupPic(sPicURL) {
            window.open( "popup.html?"+sPicURL, "",
            "resizable=1,HEIGHT=200,WIDTH=200");
```

Javascript Pop-up Function

☐ 11. As we continue to scroll down and look at the HTML and Javascript nothing really stands out. There's a few buttons, some ads, a few hidden iframes, but it's not until we see twelve **"*<*a href*"** open tags (about a quarter of the way down) that we get our first hint that something odd is going on.

What do you notice about the information contained in these tags?
Does it look similar to anything from the GET Request and the SQL Injection attack?
Review the attached screenshot for a hint.
Do you think the attack was successful?

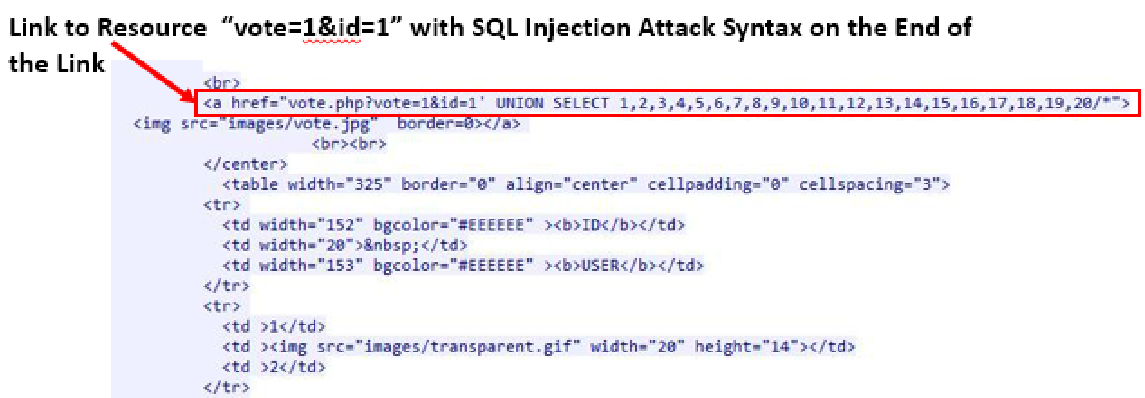
> Looking for specific information from each cat (what a cat is).
>
> the AQL injection/GetRequest was trying to get the web server to disclose one of 20 different items. The web server then attempted to resolve 12 of the 20 requests.
>
> Yes, the attack was successful.



Switch to Win 8.1

☐ 12. If we continue to scroll down we come across an interesting link enclosed in an **"a href"** attribute. It appears to be pointing to a PHP document on the web server and using it to look for a specific vote number and ID number. However, appended to the end of the URI is the SQL Injection attack syntax reflected back to us in client-side code!

Review the attached screenshot for more information.



Switch to Win 8.1

☐ 13. Continuing on through the HTML and Javascript we come across one more instance of the SQL inject / query.

See if you can find the last piece of injected code. See the attached screenshot for a hint.

> The SQL Injection was trying to receive the hidden names of 20 values on the web server.

Hidden Field with name “IdMem” and Value of “1 ‘ UNION SELECT <1-20>

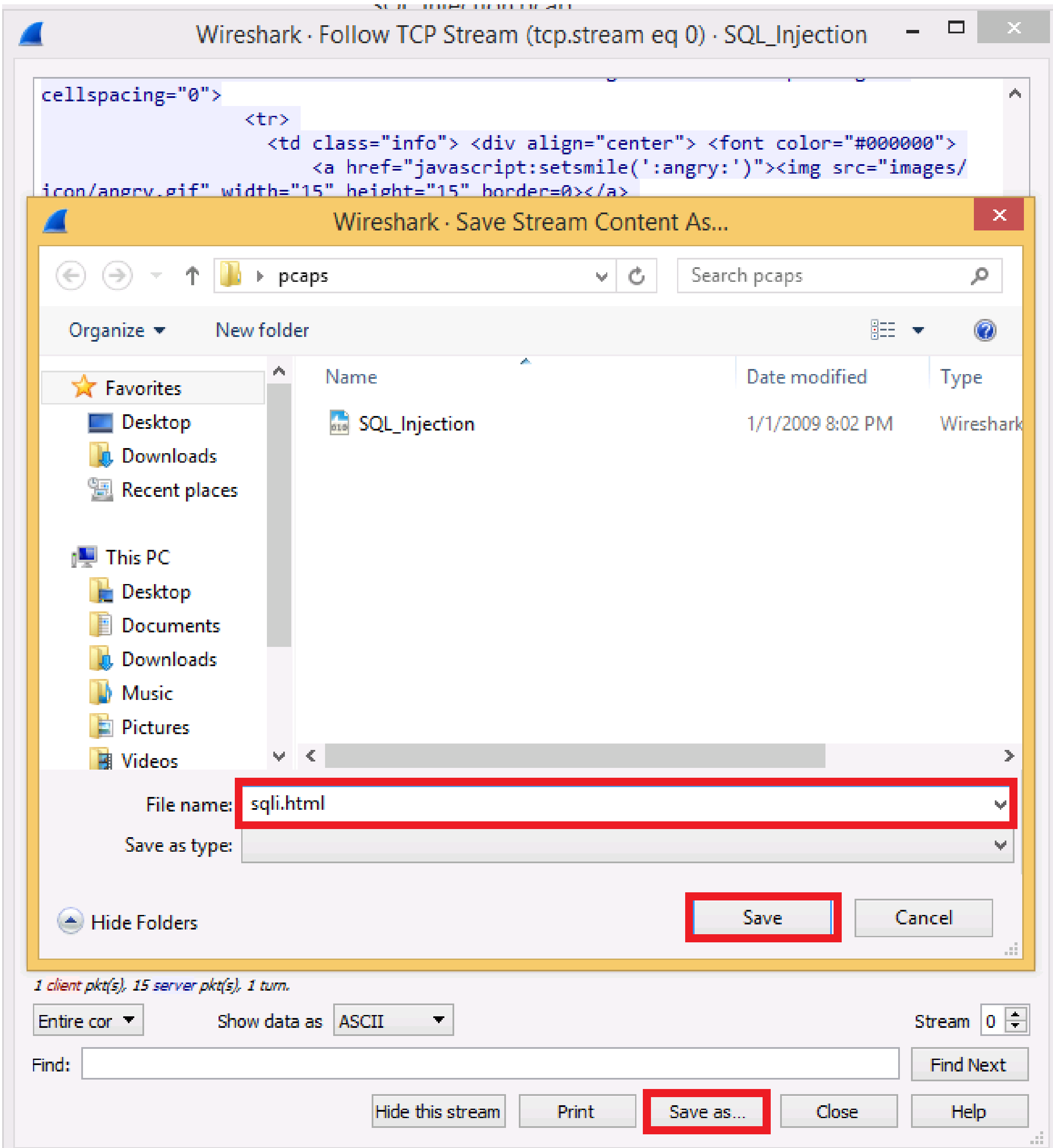
```
maxlength=50></td>                    </tr>
    <tr>
      <td width="30%" valign="top" class="info"><div align="right"><font color="#000000"><b>........</b></font></div></td>
      <td width="70%" align=left><textarea name="CommentText" wrap="physical" cols="40" rows="8" class="info"></textarea>
        <span class="info"><font color="#FF0000"><em>!</em></font></span></td>
    </tr>
    <tr>
      <td class="info"><div align="right"><a name="post"></a></div></td>
      <td><INPUT TYPE="hidden" name="IdMem" Value="1' UNION SELECT 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20/*">
                                   <INPUT TYPE="hidden" name="CatMem" Value="">
        <input name="Submit" type="submit" class="form" value="    OK    ">                    </td>
```

Switch to Win 8.1

☐ 14. Let's save this output as a **.html** file and try to open it in a browser to see what happens. To do this, at the bottom of the **TCP Stream** window click the **Save As** button.

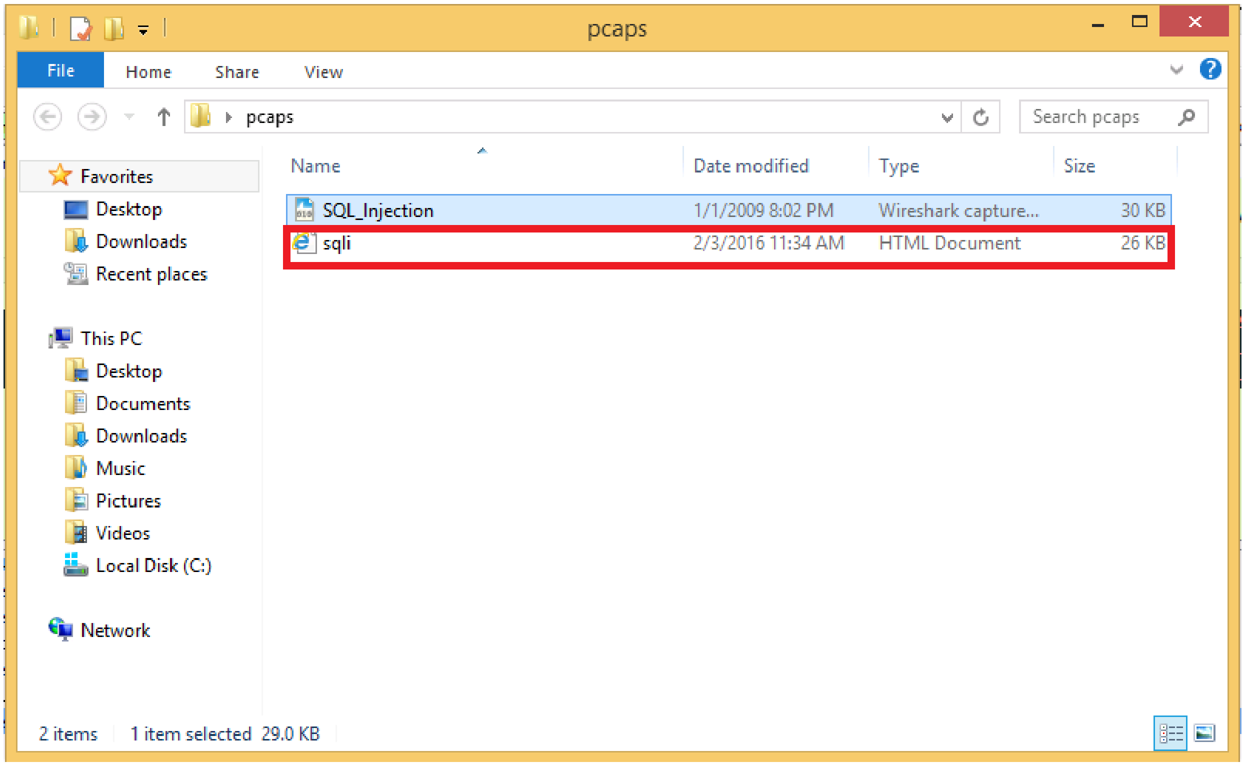
In the File Name field type **sqli.html** to save the data as web page file.

Go ahead and close the TCP Stream window and minimize Wireshark at this time.



Switch to Win 8.1

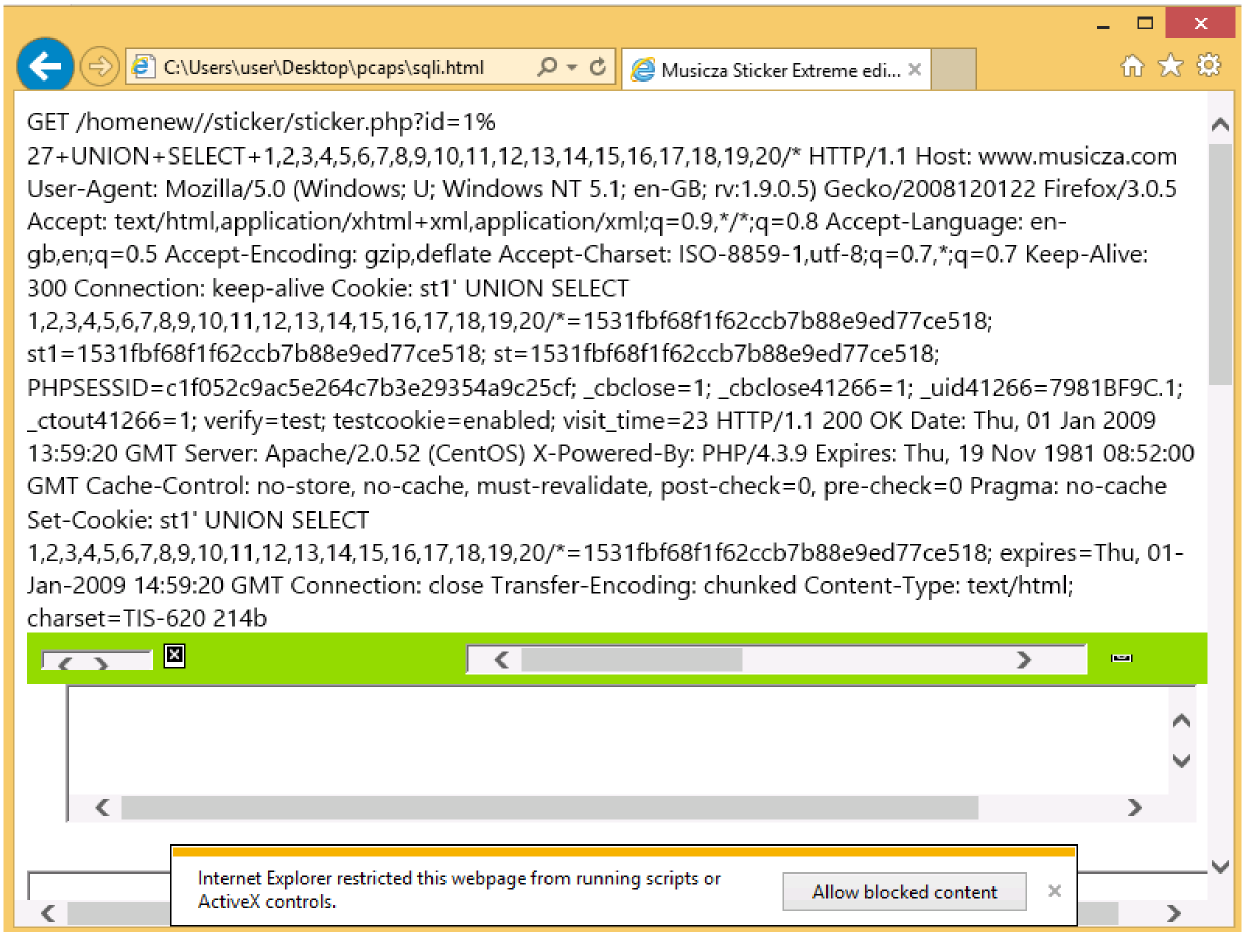
☐ 15. Now go to the **pcaps** folder and open the **sqli.html** file you just made.



Switch to Win 8.1

☐ 16. Look at how the Browser renders the HTML and Javascript from the file we saved. Do you see the SQL Injection attack syntax at the top of the page?

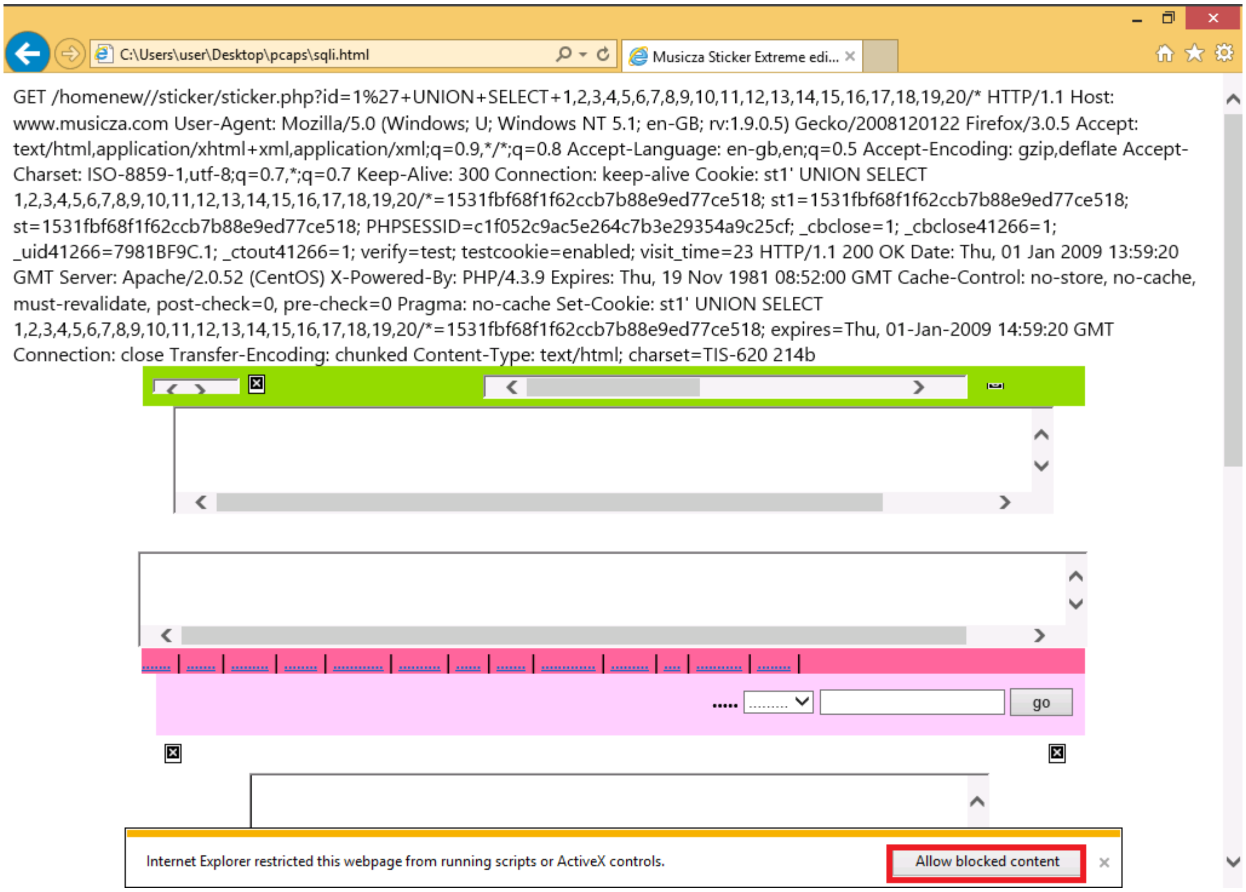
Notice how the browser stops scripts from running because it considers them dangerous.



Switch to Win 8.1

☐ 17. Let's allow the scripts to run by clicking **Allow Blocked Content** button at the bottom. If you don't see the button there, close out the browser, and open the **sqli.html** file again. As soon as the blocked content alert pops up click to click the **[****Allow blocked content]** button.

Do you see the difference?

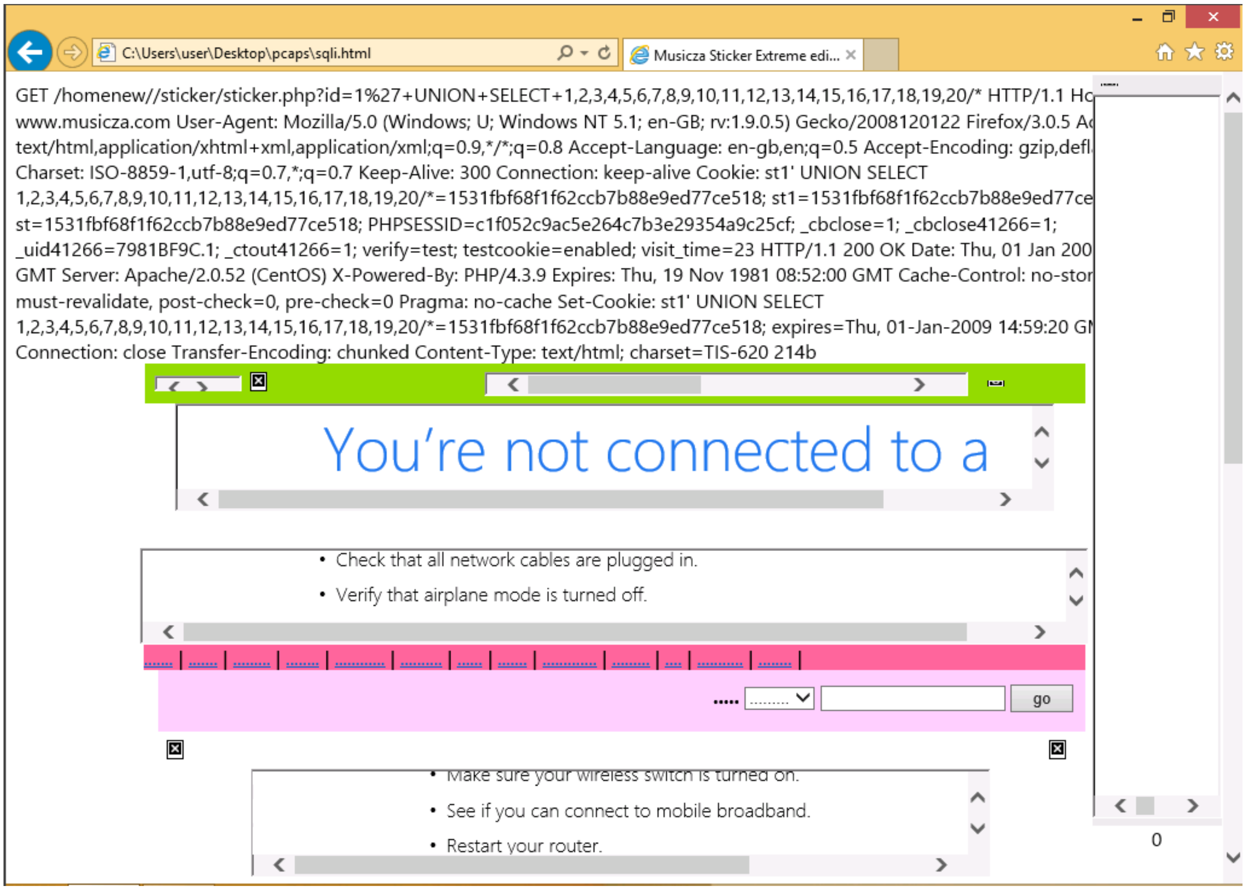


☐ 18. If you allow the blocked content you will see several iframes appear in the browser. If you use the scroll function on them and look inside you'll see some interesting information.

See the screenshot for a clue.

You'll notice the browser has error messages present in each of the iframes telling us we are not connected to a network. That means that each of those iframes tried to call out over the internet to another resource somewhere, but since we are not online that content couldn't be accessed and as a result the browser gave us those errors.

If you go back and look at the HTML and look inside the iframe tags you'll see the "a href" attributes to each of those the various links. Or you could right-click in the browser window and then select **View Source** to see the HTML source code that way.



Switch to Win 8.1

We began by analyzing a network packet capture and seeing evidence of an attempted SQL Injection attack. We then followed the TCP Stream and reviewed the entire request and response cycle. We then saved off the information from the packet capture into a .html file and rendered it in a browser.

We found evidence of the SQL Injection attack possible working and SQL Query syntax reflected back in the client-side code that the server dynamically generated and sent back in response to the GET Request.

Congratulations.