Web Security

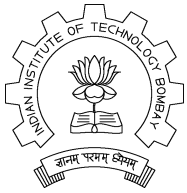
Dual Degree Project
1st Stage Report

Submitted by

Lapsy Garg
Roll No: 03D05017

under the guidance of

Prof. Bernard L. Menezes



Department of Computer Science and Engineering,
Indian Institute of Technology, Bombay
Mumbai
July 11, 2007

Contents

Abstract

The web applications like google, gmail, ebay, amazon, yahoo, msn etc. have become a part of our life. These applications are used by users for communicating, shopping, research etc. With such important transactions taking place on these web applications, the concern for user security has also grown. Also as these applications will grow, so will be the opportunities to attack them. With network firewalls protecting the servers, it's no more easy for the attackers to attack servers. Hence, user workstation is becoming the new playground for attackers. In this report we will study two application level attacks cross site scripting and SQL injections.

1 Introduction

We live in an age where we are witnessing a transition of applications from our operating system to the browsers. In the past few years we have seen companies coming up with new web application. Browsers are slowly and steadily becoming our work platforms. Web applications run on a web server and they provide services over a network. Web applications require no kind of installation on the operating system of the user. Also, the presence of web browser as a layer between operating system and the web application makes the web applications independent of the operating systems. Since, these web applications provide service over a network they can be accessed from anywhere and anytime provided you have a computer and an internet connection. The easy deployment procedure, operating system independence and wide scale accessibility are the reasons why web applications have become so popular.

Web applications are developed using technologies like HTML, PHP, JavaScript, CSS, XML, AJAX etc. These web applications generally rely on a three-tier architecture [15]:

- The client is the web browser executed on end-user's system.
- The web application itself is a Web Page, often incorporating a large number of different techniques.
- The data provider is often a database containing relevant information.

All three tiers have their own vulnerabilities and the attacker just needs to exploit one to compromise one part of the whole web application.

In this stage I studied two of the most widespread attacks on web applications:

- **Cross Site Scripting (XSS)**
  XSS is an attack that forces a web site to echo attacker-supplied executable code, which then loads into the user's browser.

- **SQL Injections**
  SQL injection is a technique for exploiting web applications that uses user supplied data to build SQL queries without filtering potentially harmful characters.

The report is divided further divided into 3 sections. The 2nd section of this report will deal with cross site scripting attacks, the 3rd section will deal with SQL injections and the last section will present the conclusion and future work.

2 Cross Site Scripting

2.1 Introduction

Designing a secure web application is inherently a difficult task. The application needs to present a public face to the users and also have to interact with the user accepting and returning data. A number of methods to attack these web applications has been devised e.g. SQL injections, buffer overflow etc. Cross Site Scripting, also known as XSS in the internet community, is a form of attack which has received a great deal of attention because of the ease of finding XSS vulnerabilities and the ease with which they can be exploited.

Cross Site Scripting attacks are those in which attackers inject malicious code into the web application. XSS attacks occur when the web server fails to validate the data sent by the user and a dynamic web page is generated containing this data. The XSS attacks do not affect the architecture of the web server. The XSS code, written typically in HTML/JavaScript, gets executed on the user browser and not on the web server. XSS attacks enable attackers to steal browser cookies, which can be used by them to gain control of the user account. Later, in this section we will look at a few real life examples of the methods by which XSS can be used for browser cookie theft.

XSS outbreaks [12]:

- Are likely to originate on popular websites with community-driven features such as social networking, blogs, user reviews, message boards, chat rooms, web mail, and wikis.
- Can occur at any time because the vulnerability (Cross-Site Scripting) required for propagation exists in over 80% of all websites.
- Are capable of propagating faster and cleaner than even the most notorious worms such as Code Red, Slammer and Blaster.

- Maintain operating system independence (Windows, Linux, Macintosh OS X, etc.) since execution occurs in the web browser.
- Circumvent network congestion by propagating in a web server-to-web browser (client-server) model rather than a typical blind peer-to-peer model.
- Do not rely on web browser or operating system vulnerabilities.
- May propagate by utilizing third-party providers of Web page widgets (advertising banners, weather and poll blocks, JavaScript RSS feeds, traffic counters, etc.)
- Will be a challenge to spot because the network behavior of infected browsers remains relatively unchanged and the JavaScript exploit code is hard to distinguish from normal web page markup.
- Are easier to stop than traditional Internet viruses because denying access to the infectious website will quarantine the spread.

2.2 Types of XSS Attacks

There are two ways for a user to become infected with XSS attacks, Persistent and Non-Persistent [1]. In Non-Persistent attack a user is tricked into clicking a specially crafted link which leads to the execution of a malicious XSS code. In Persistent attack a user gets attacked by XSS only by visiting a web page with malicious XSS code. We will now look into these attacks with the help of some real life examples.

2.2.1 Non-Persistent

Consider that an attacker wants to attack a website http://victim/. The attacker would first of all identify the XSS vulnerability in this website and then he would craft a link which if visited by a user would lead to the execution of the XSS code.

I will explain this attack using the XSS vulnerability present in the Q & A section of the website http://www.rediff.com/ i.e. http://qna.rediff.com/. The data entered into the search box of this website which has been highlighted in the figure 1 is not properly filtered by the web server. Also the search query inserted in the search box gets printed after a search has been run. This XSS vulnerability can be utilized by the attacker to insert malicious code in the search box. I inserted the code "<SCRIPT>alert('hey')</SCRIPT>" in the search box figure 1 which resulted in an alert printing "hey" after the search query was made in figure 2. The same result can be obtained if someone clicks on the link

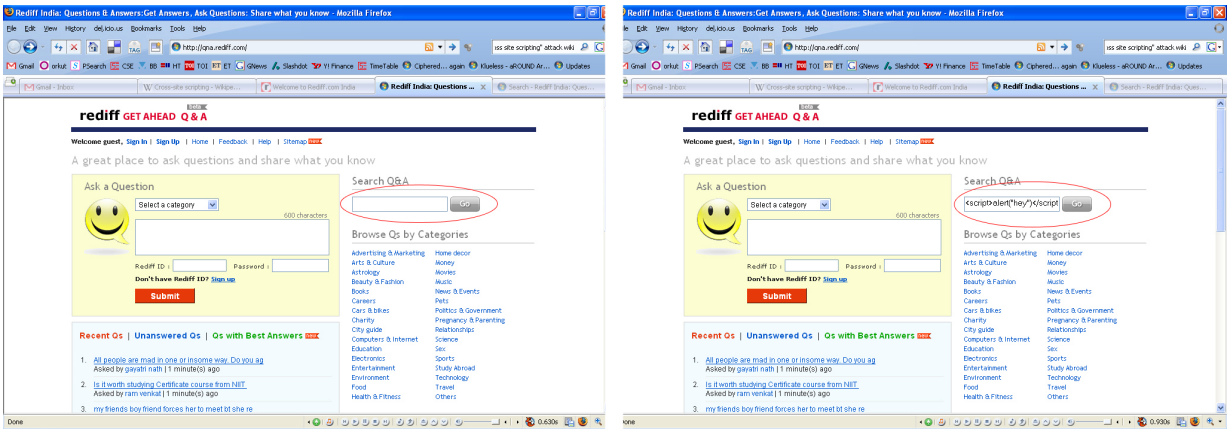


Figure 1: *Rediff Q & A Search*

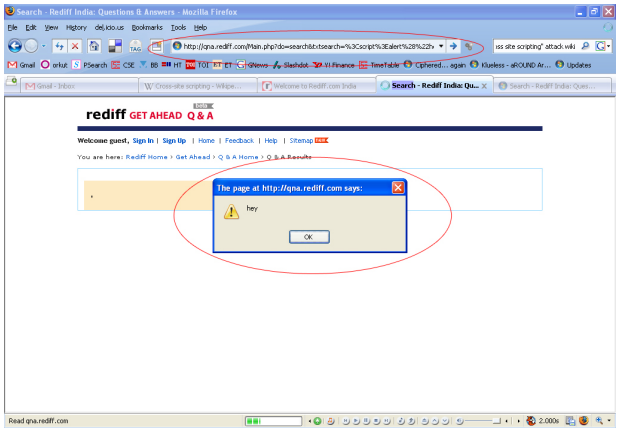


Figure 2: *Rediff Q & A XSS Attack*

http://qna.rediff.com/Main.php?do=search&
txtsearch=%3Cscript%3Ealert%28%22hey%22%29%3Cscript%3E.

After identifying this vulnerability the attacker can insert much more complicated JavaScript code in the link for various purposes such as cookie theft. The cookie theft can be done by simply inserting the following code:

<SCRIPT>document.location=http://attackerSite/?cookie=document.cookie</SCRIPT>

Once the attacker is finished devising the code he can publish this link to trick other users into clicking the same. What makes this attack so effective is the fact that most of the users are going to click the link without any suspicion and the attacker can easily device a code to hide the attack, so that even after the malicious code has executed the user will not have any idea about the attack.

2.2.2 Persistent

Persistent attacks occur mostly at community driven web application which involve users visiting the web pages of other users. Persistent attacks do not require attackers to publish any specially crafted URLs. These attacks happen only by visiting webpage and can be easily hidden from the victims. In this sense persistent attacks are much more vicious then non-persistent attacks. SAMY worm is one of the most famous persistent attacks that happened on MySpace.com. I will discuss this worm in detail later in this report.

Let us have a look at this kind of attack on Ebay website http://www.ebay.in/. As shown in figure 3, the web page for selling an item contains a section where a user can input HTML code. Even a reputed company like ebay has not taken care of stripping of JavaScript code from this HTML code. Therefore, an attacker can input any kind of JavaScript code in this section. Now, let us look at a simple example of how this vulnerability can be exploited by an attacker. If we go to the web page of any item, which is being sold on ebay, there is button called "Watch This Item" on the web page as highlighted in figure 3. Clicking on this button would add the respective item into our watch list. If we add the following code to the description part of our item, it would result in the addition of our item to the user's watch list, whenever he visits the webpage of the same.

<SCRIPT>document.forms["watch_thisItem"].submit();</SCRIPT>

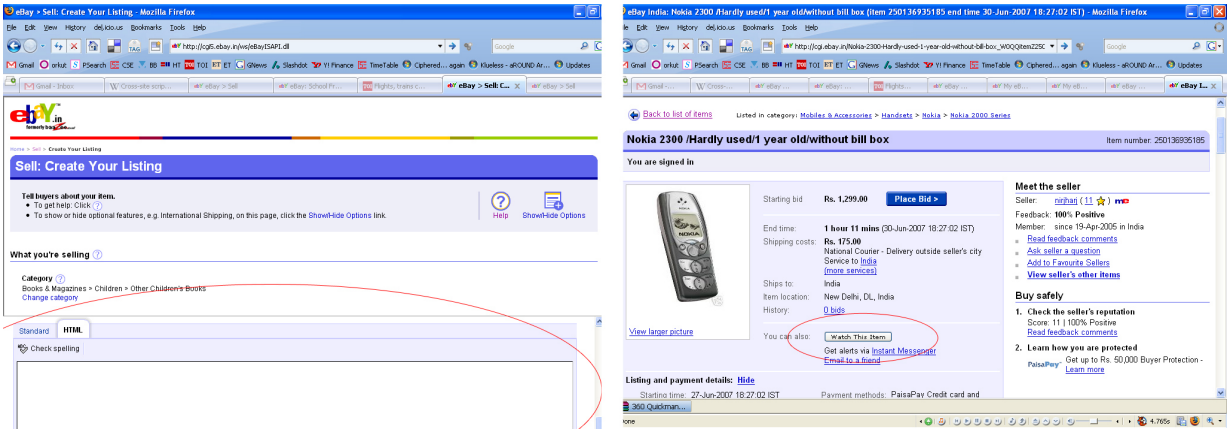


Figure 3: *eBay Webpage*

This is potentially not a very harmful attack but a little more complex code can be used to place automatic bids on our item, display a false feedback rating etc. This example clearly

demonstrates how lethal these kind of attacks can be and why e-commerce websites should specially take care when designing their web applications.

2.3 Tools of XSS

Now, I will concentrate on the tools used by the attackers to attack web applications with XSS attacks. XSS attacks are typically done with HTML/JavaScript code. There are three ways in which HTML/JavaScript codes can be used [12]:

- Embedded HTML Tags
- JavaScript and DOM Objects
- XMLHttpRequest

2.3.1 Embedded HTML Tags

In this kind of attack attackers embed malicious JavaScript code in the HTML tags. In few cases this kind of attack help attackers overcome JavaScript filters. For example, if there is a web application which filters out <SCRIPT> tag and the data inserted in this tag, then the attacker can use tags such as <img> or <a> to insert malicious JavaScript code. Following are the few examples which demonstrate how these tags can be used for the execution of malicious JavaScript code.

- <A href="javascript:alert('malicious code')">Malicious Link</A>
- <A href="http://link" onmouseover="alert('malicious code')">Malicious Link</A>
- <IMG src="http://www.attackerWebsite">. This link would result in the auto upload of attackerWebsite on the victim browser. This attack no more works with Internet Explorer 7.0 and Firefox 2.0.
- Attacker can also insert a code of a malicious form in the victim website and using JavaScript can automatically submit the form.

2.3.2 JavaScript DOM objects

Now, we will have a look at how attackers can utilize JavaScript DOM objects for attack on a web application. But before that, we will briefly have a look at the security model of the JavaScript.

[2] Every browser executes JavaScript from a random website with the premise that it is hostile. The primary JavaScript security policy is the same origin policy. According to this policy

JavaScript loaded from one web application prohibits it from accessing other web applications. i.e. JavaScript loaded from the website http://www.abc.com/ cannot gain access to any objects loaded from the website http://www.xyz.com/. Without this policy JavaScript from a hostile webpage would be free to manipulate the document structure of any other website, which might result in potentially fatal results like stealing of user's login cookies, credit card number etc.

When a script attempts to access methods from a different documents opened using "window.open()", the browser does the same origin policy check and if this condition is violated, an exception is thrown. Two documents are said to have the same origin if they were loaded from the same domain. Therefore, JavaScript loaded from webpage http://www.abc.com/src/mail.php can gain access to the objects of the webpage http://www.abc.com/. Also, webpage with the url http://www.xyz.abc.com/ cannot access the objects of the webpage http://www.abc.com/ because they are considered to be from different domain. But in the later case, this policy can be easily overcome by setting javascript.domain property to "abc.com" in the former url.ie. javascript.domain="abc.com".

Before, I go on to explain the attacks that can happen using JavaScript, let us have a brief look at the Document Object Model [11]. An HTML page is made up of many objects such as forms, images, links etc. To help authors have a better control over these objects every browser defines a document object model (DOM). DOM can be defined as a prototype or plan for the organization of objects on a page. DOM primarily focuses on HTML and the content nested inside it. It also provides author some control over the environment that contains the document: the window. Web developers can express their control over the document object model using the languages like JavaScript. So, basically the malicious JavaScript code inserted by an attacker provides him a full control over the document object model of the page.

Now let us have a look at few examples which demonstrate how JavaScript DOM objects can be used for XSS attack.

- My earlier example of ebay demonstrates how an attacker can use DOM objects to his advantage. Using JavaScript, attacker is able to access the form with name "watch_thisItem" and automate the process of submission of this form without the user's permission.
- Similarly, JavaScript can be used by an attacker to open pop up windows for say, displaying some advertisement etc. i.e.

<SCRIPT>window.open("http://attackerURL/");</SCRIPT>

2.3.3 XMLHttpRequest

XMLHttpRequest is used to send asynchronous HTTP requests to the server. The request processing takes place in the background and results of the request are not displayed immediately. Rather, the result of the request gets returned to the object of XMLHttpRequest which was used to send the request. This utility of JavaScript is used for faster display and processing of web pages. At the same time, this utility of JavaScript can also be put to use for the most fatal attacks on web application. Since the processing takes place in the background, it can be used for executing the most complex codes without user noticing a thing. The SAMY worm which we will be discussing later in this section utilized this functionality to best of its advantage. An example of XMLHttpRequest is as following:

```
var req = new XMLHttpRequest();
req.open('GET','http://victimURL','true');
req.onreadystatechange=function()
{
        if(req.readyState==4)
        {
          document.write(req.responseText);
        }
};
req.send(null);
```

This can be used for sending a GET HTTP request to the victimURL.

```
var post_data="display_name=YouHaveBeenHacked&visible=true";
var req= new XMLHttpRequest();
req.open('POST','http://victimURL','true');
req.onreadystatechange=function()
{
    if(req.readyState==4)
    {
        document.write(req.responseText);
    }
};
req.send(post_data);
```

This code will result in a POST HTTP request.

2.4 Defense against XSS

As the attacks mature, so does the defense. Over years research community has not only unveiled new ways of attack but also the methods of defense against these new attacks. The biggest reason why XSS even exist is because some web application developers do not care or forget to properly sanitize the data. With web application becoming large and large such mistakes will occur and it's the same reason why these mistakes will become more and more fatal for the web applications. The research community has come up with a few good methods to overcome these mistakes and now I will discuss some of these methods.

2.4.1 Secure Web Applications Project (SWAP)

The Secure Web Application Project [17] [18] is a research initiative between the Laboratory for Communication Engineering and the Computer Library of the University of Cambridge. One of the most difficult task in the development of a web application is to abstract out the security code.

- The web applications might be written in variety of non-interoperating languages. In this case there is no easy way to abstract out the security code behind a clean API.
- Due to the growth in open-source software development and a number of mergers and acquisitions the web application today often contain 3rd party code and its not always feasible to modify the 3rd party code.

This project provides web application developers with the tools to protect their application from application level attacks such as XSS and SQL Injection attacks. We will look into SQL injection in detail in the next section.

System Architecture

This system consists of the following components:

- **Security Policy Description Language (SPDL)** SPDL is used to specify a set of validation and transformation rules on HTTP requests and responses.
- **Policy Compiler** It automatically translates SPDL into code.
- **Security Gateway** It is placed between Web Server and Client machines.

Security Policy Description Language

Security Policy Description Language is used by web application developers for code validation and transformation rules. Validation rules are used to place restriction on cookies, URL parameters and forms. Some examples of the validation rules are as follows:

- Maximum and minimum length of a parameter
- Type of a parameter
- A constraint on the presence of a parameter i.e. is the parameter always required or not.
- Method by which a parameter should be passed i.e. GET, POST or both.

The validation rules are specified using XML. Following is an example of validation rules:

<policy>
<URL prefix="http://example">
<parameter name="p1" maxlength="4" type="int" required="Y">
</parameter>
<parameter name="p2" method="POST"maxlength="3" type="string">
</parameter>
</URL>
</policy>

Similarly transformation rules can be defined for each parameter. So, if the user wants to apply a transformation t1 followed by t2 on parameter p then he can define the following rule:

<URL prefix="...">
<parameter name="p" ...>
<transformation> t1 | t2 </transformation>
</parameter>
</URL>

Transforations can be of the following type:

- **EscapeSingleQuotes:** Replace all single quotes with their HTML character encoding.
- **EscapeDoubleQuotes:** Replace all double quotes with their HTML character encoding.
- **HTMLEncode:** Replace data with its HTML encoding.
- **PartialHTMLEncode:** HTML encode the data but leave a small number of allowed tags .e.g <b>, <u>, <i> etc.

Policy Compiler

Policy Compiler takes the SPDL specification and compiles it for execution on security gateway. Compilation is performed in two passes. In the first pass, parameters and their types are

enumerated and in the second pass, validation and transformation rules are compiled.

The Security Gateway

The security gateway first extracts the URL prameters from the HTTP request. After seperating all the parameters in cookies and URL, the security gateway checks all the parameters and errors are generated if (i) any of the parameter doesn't match the SPDL specification (ii) Type mismatch (iii)the cookies present don't match the one specified in SPDL (iv) any parameter is missing. If parameters and cookies pass this test, the security gateway tests if the message authentication code is valid or not. Next security gateway applies the transformation specified in SPDL and then it checks the parameters and cookies with the validation code. If parameters and cookies pass all this test then the HTTP response is returned, else it returns an error message.

Discussion

It's a good approach as far as abstracting security code to one place is concerned. This approach would also not require web application developers to make any changes in their existing code. Moreover, considering the number of web application and their size which exists today, this approach can be very useful to protect these web applications without making any changes in the existing code. But this approach again faces the problem of scalability. As the application grow, so do the number of URL parameters and forms. Defining rules for each and every parameter in this case would be a very difficult task for the application developer. This task becomes even more difficult when application developers are using third party code, e.g. a web application using phpBB for a forum, because it would require him to study the entire third party code. In this approach the whole responsibility of defeating XSS falls on the shoulders of the web application, due to which the security check overheads become quite high for the application. In the next sub-section, we will look at another approach which distributes the responsibility of defeating XSS between browser and the web application.

2.4.2 BEEP (Browser Enforced Embedded Policies)

BEEP [14] is a technique to prevent script injections based on the following observations:

- Browsers perform perfect script detection. If a browser doesn't parse some content as script, then it will not be executed. Therefore, browser is the ideal place to filter the scripts.
- The web application developer knows exactly what script should be executed for the application to function properly. Hence, web application should supply the filtering policy to the web browser.

In the implementation of BEEP, the security policy is expressed as a trusted JavaScript function that a web application embeds in the pages it serves. This function is called security hook. BEEP

implements two kind of policies for protection against XSS.

The first policy is whitelist, in which the hook function consists of hash of each legitimate script appearing on the web page. When the browser parses the web page and it detects a script, the script is passed to the hook function which compares the hash of the script with legitimate script hashes. If a match is found, then it's considered a valid script else the script gets rejected.

The second policy is DOM sandbox. In this case the possible malicious user content is placed under a or element which acts as a sandbox. Within the sandbox, if a script is found, it gets rejected.

Discussion

This approach demonstrates how browsers and web applications can work together to defeat XSS. Due to the hook functions, the unapproved scripts will never be parsed by the browser and will therefore, get rejected making it very difficult for the attackers to find XSS vulnerabilities. Also, due to the collaboration between browser and the web application, the overhead of security check on the web application reduces considerably making them much more efficient. The biggest drawback of this approach is that it would require changes in the existing code, if the code already doesn't support BEEP, which is a considerably difficult task considering the size of web applications that exist on internet today. But that's the price which web application developers would be ready to pay considering the importance of security of web application.

Clearly, BEEP is a much better approach then SWAP for defeating XSS attacks. But SWAP is much broader than BEEP in the sense that SWAP can not only be used for defeating XSS but also other kind of application level attacks such as SQL injections, PHP injections etc. Another advantage of SWAP is that it abstracts out the security code from the application which is not the case with BEEP. In BEEP, the security code needs to be the part of the web application.

2.5 SAMY

I will conclude my discussion on XSS worms with an overview of SAMY worm [3]. On October 4, 2005 the first major XSS worm hit the MySpace website. Samy, the author of this worm was on a spree to become popular amongst his friends. So, he designed this worm to exploit XSS vulnerability in MySpace website. MySpace was using some kind of data filtering to prevent such attacks but they were far from perfect. Utilizing one of these vulnerabilities SAMY was able to inject his malicious code into his profile webpage which led to a persistent type of XSS attack.

If any MySpace user visited Samy's profile page, Samy will get added to his/her friend list.

Moreover Samy will get added to the user's hero list and most of all the display name of the user will change to "Samy's my hero". Also the same malicious code will get injected into the user's profile page and the same actions will now happen if any user visited this user's webpage.

Clearly, this worm will spread exponentially and that is what happened. Starting with one user, within 24 hours this worm spread to 1,000,000 users. MySpace was forced to shutdown its operations to stop this worm from spreading any further and to fix the vulnerability and to do the clean up. This worm did not do any potential harm to MySpace. They were forced to shutdown their operations but the problem was fixable. But had it been written by some serious attacker, the implication of such an attack could have had been much more fatal.

This worm clearly demonstrates the potential of cross site scripting attacks and why any web application developers cannot take them lightly. Such vulnerabilities still exist on many social networking websites like http://www.hi5.com, http://www.desimartini.com/, etc. We hope, it would not be a worm like Samy which will make them realize how potentially harmful these vulnerabilities can be.

3 SQL Injections

3.1 Introduction

In the last section, we had a look at the cross site scripting attacks. In this section I will be discussing SQL Injections, which is another kind of application level attack. Web applications with database-driven content are ubiquitous today. In such a scenario security of database becomes one of the most important tasks for the application developers. A mistake in the use of database through the application and a relentless attacker is all it will take to bring down the application. Hence, SQL Injections are a big problem and every web application developer need to have knowledge about them.

SQL injections are the vulnerabilities which occur if a web application uses user supplied data for SQL queries without properly sanitizing it. A database server product has no mechanism to deal with SQL injections. The root cause of SQL injection exists not in the database layer but in the application itself. In the following sub-section we will have a look at some of the examples of SQL injections.

3.2 SQL Injection Attacks

SQL injection attacks, as described earlier occur because of the use of user supplied data in SQL queries without sanitizing the data properly. One of the reasons why SQL injection attacks are difficult to detect is because they result in perfectly valid SQL commands, which gets executed because some web developer didn't filter the data properly and the attacker having found this vulnerability utilized it to reformat the structure of SQL commands being used by the web application. Table 1 explains broadly categories in which an attacker can utilize SQL injections to his advantage [16].

Attack Type	Results
Unauthorized Data Access	Allows the attacker to trick the application in order to obtain from the database information that is not supposed to be returned or is not allowed to be seen by this user.
Authentication bypass	Allows the attacker to access the database-driven application and observe data from the database without presenting proper credentials.
Database modification	Allows the attacker to insert, modify, or destroy data content without authorization.
Escape from a database	Allows the attacker to compromise the host running the database or even attack other systems.

Table 1: *Forms of SQL Injection Attack*

Let us now, look at some of the examples of SQL injection attacks.

3.2.1 Parameterized Attack

Most of the web applications use SQL queries for data retrieval. In these SQL queries some of the user supplied data gets inserted. If the user supplied data is not properly sanitized then it would provide an attacker with the opportunity to disclose some of the information which he is not be authorized to access. Such attacks depend on a strategically crafted input into the SQL query template and are called parameterized attacks [16]. Some of the examples of parameterized attacks are as follows.

- **Dangling Parameters** A dangling parameter is a first order replacement in an SQL query. In this case there is generally nothing following the last parameter and hence, can result in

the modification of the query.So, for example, for a query of the form:

SELECT * FROM users WHERE name = '{$name}'

In this case if an attacker passes the value of $name as "' OR '1'='1" it would result in the following query

SELECT * FROM users WHERE name='' OR '1'='1'.

Now the condition email='' OR 1=1 will always evaluate to true and hence, this query would print all the rows of the table "users".

- **Second Order Replacement**

  Second Order replacement involves string replacement anywhere in the query. So, for example if we have the query of the form

  SELECT * from users where username='{$user}' AND passwd='{$passwd}'

  Now if the attacker sends "admin' –" for $user and any string for $passwd say, "abc", it would result in the following query:

  SELECT * from users where username='admin' – ' AND passwd='abc'

  Due to the double hyphens (–) " ' AND passwd='abc' " part of the query will get ignored and hence, the tuple with username='admin' in table users will get returned. This kind of query can give an attacker unauthorized access to other user's account on the web application.

- **Unquoted Numerical Parameters** This is the most dangerous location for exploits in an SQL statement. The absence of quotes will allow the query to be changed and methods like escaping string literals wouldn't work. So, for example

  UPDATE items
  SET price={$price}
  WHERE item_name={$item_name}

  In this example, the malicious user can pass any form of content for $price. So in this case any string literal in $price other then a number will be a part of the SQL query.

3.3 Methods of Defense against SQL Injections

Now, I will discuss some of the precautions and measures that a web application developer can have to protect his web application against SQL injections.

3.3.1 Parameterized Queries

One way to prevent SQL injections is not to use dynamically generated SQL statements. Instead the web application developers should use parameterized SQL queries. Let's look at an example to study this concept in more detail.

In java following code can be used to execute a SQL query

Connection con = (acquire Connection)
Statement stmt = con.createStatement();
ResultSet rset = stmt.executeQuery("SELECT * FROM users WHERE name =' " + userName + "';");

This code results in the dynamic execution of the query by which we mean, the SQL query will be parsed at runtime instead of compile time. The same query can also be run using the following code.

Connection con = (acquire Connection)
PreparedStatement pstmt = con.prepareStatement("SELECT * FROM users WHERE name = ?");
pstmt.setString(1, userName);
ResultSet rset = pstmt.executeQuery();

In this case the SQL statement will get parsed at compile time rather than at execution time. So, an attacker will not be able to change the structure of the query. Also this code puts a type constraint on the user input. "userName" in this case has to be a string and any kind of special characters will get escaped i.e. an input like "' OR 1=1" will be changed into "\' OR 1=1". Hence, no scope for an SQL injection will be left.

But it's not always possible for the application developers to use parameterized queries. For example, if the name of the table in the query is dynamic, then this approach will not work because we will not be able to represent such a query in parameterized form and hence, we will be forced to use dynamic queries. Though such cases are very few but application developers might come across such situations and thus will be forced to use dynamic SQL queries which if not

properly sanitized might open the gates for SQL injections.

3.3.2 SQLRand

This approach to defense against SQL injections [9] involves randomization and de-randomization of the query. The concept behind this approach is the fact that, an attacker will not be able to do any kind of SQL injection if he doesn't know the syntax of SQL queries being used by the server. So, if an attacker tries to use standard SQL syntax for an SQL injection, the system can raise an alert that an SQL injection attack is being attempted.

So, in this approach first of all a random key is used to randomize the SQL queries i.e. all the SQL keywords are encrypted with the help of the random key e.g. SELECT will become SELECT123 if the key is 123. Now, there is proxy server which sits between the database and the application. The query is sent to this proxy server for de-randomization. If, this proxy server while parsing the query finds certain SQL keywords which have not been randomized, it can raise an error, warning the system about an attempt of SQL injection.

For example, a query of the form

SELECT * from users where username='{$username}' AND passwd='{$passwd}'

is randomized to

SELECT123 * from123 users where123 username='{$username}'
AND123 passwd='{$passwd}'

where "123" is the randomization key and not known to the attacker.

In this case if an attacker attempts to pass an input of the form say $username="'OR 1=1 " and $passwd="abc", then his attempt of attack will get thwarted since when the proxy server will be parsing this SQL query, it will find SQL keywords such as OR which have not been randomized.

As far as performance and protection against SQL injections is concerned it's a very good approach. The average overhead for each query came out to be 6.5 ms, which would have almost negligible effect on the performance of a web application. The only shortcoming of this approach is it would require the web application developer to change their existing code but again that is not a very high price to pay considering the importance of database security.

3.3.3 AMNESIA

Now, we will have a look at another approach for protection against SQL Injection attacks. This approach is called AMNESIA [13] which stands for Analysis and Monitoring for NEutralizing SQL-Injection Attacks. This approach consists of four main steps:

- **Identify hotspots:** This requires scanning of the application for identifying points which issue SQL queries.
- **Build SQL query model:** For each hotspot build a SQL query model which can represent all the possible SQL queries at that hotspot. An SQL query model is a non-deterministic finite state automation in which the transition labels consist of SQL tokens, delimiters and placeholders for string values.
- **Instrument Application:** At each hotspot in the application, add calls to the runtime monitor.
- **Runtime Monitoring:** At runtime, check the dynamically generated SQL queries against the respective SQL-query model and if a violation to the model occurs, then report an error.

Figure 4 shows the SQL-query model which will get generated for the query:

SELECT info FROM userTable WHERE login='{$login}' AND pass='{$pass}'

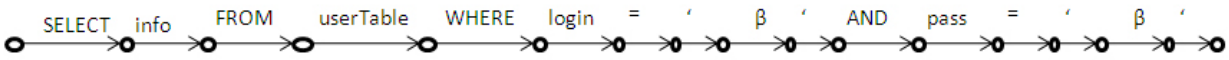


Figure 4: *SQL-Query Model*

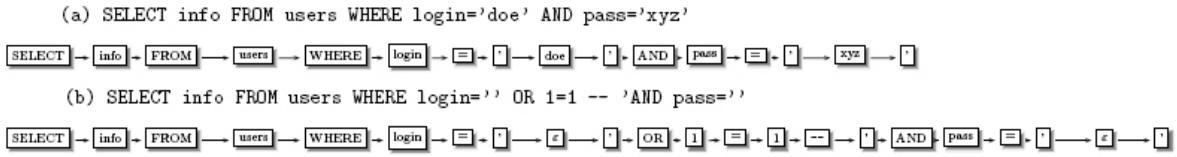


Figure 5: *Example of parsed runtime query*

Now for input $login="doe" and $pass="xyz" we will get the SQL-query model as represented in figure 5(a) and for the input $login="' OR 1=1 – " and $pass="" we will get the

SQL-query model as represented in figure 5(b). Clearly in the second case it leads to a violation of the query model. Thus this approach will raise an alert in this case and attackers attempt at an SQL injection will get thwarted.

4 Future Work

Considering the importance of security against cross site scripting attacks and the large number of web applications which are vulnerable to it, for my 2nd stage of Dual Degree Project, I propose to study cross site scripting attacks in further detail. Also, I will work towards a new solution for cross site scripting, which would automate the detection and prevention of XSS to a much more extent than the existing approaches. This approach will be based on making browser aware of the data which has been supplied by the users. So, any javascript code inside that data should not be parsed. In XSS, I would also like to look at DDoS attacks which can happen using XSS vulnerabilities.

Also, if time permits I would like to look at other form of attacks such as p2p worms and spyware. Spyware is computer software that is installed surreptitiously on a personal computer to intercept or take partial control over the user's interaction with the computer, without the user's informed consent. Spyware is another concern rising in the internet community and present a huge challenge to the research community. There has not been much work done in the area of auto generation of signatures of spyware, which looks like a very interesting field to target.

p2p worms also present a big challenge to the research community. These worms do not need to do any kind of port scanning for finding their target. The target list is radially available through the p2p software. Hence, they can get down to the business of attacking other systems as soon as a system gets infected by them. Due, to this reason p2p worms present a big and very real threat to the internet.

5 Conclusion

In this report, we had a look at two application level attacks which the web applications face today, cross site scripting and SQL injections. One fact, which does come out of this report, is the importance of data sanitization. The biggest reason these problems exist is because sometime web application developers don't sanitize data at all and even if they do, they do not do it properly. We also had a look at some of techniques which research groups around the world have come up with to defeat these attacks. We also looked at short comings and the strengths of these techniques. It's quite unfortunate that these techniques are still in the development stage and has

still not been accepted by the industry.

Also in this report we looked at some real life examples of the attacks which are possible on the web applications like ebay, rediff etc. These attacks could not only be fatal for the web application but also for the users, who are using those web applications. It becomes a responsibility of these web applications to protect their users against such attacks.

References

[1] Threat Classification, http://www.webappsec.org/projects/threat/classes/cross-site_scripting.shtml.

[2] JavaScript Security, http://www.windowsitlibrary.com/Content/1160/22/1.html.

[3] MySpace Worm Explanation, http://namb.la/popular.

[4] The Cross Site Scripting (XSS) FAQ, http://www.cgisecurity.com/articles/xss-faq.shtml.

[5] XSS Cheat Sheet, http://ha.ckers.org/xss.html.

[6] The XSS Blacklist, http://pointblanksecurity.com/xss/.

[7] Rediff, http://www.rediff.com/.

[8] Ebay India, http://www.ebay.in/.

[9] SW Boyd and AD Keromytis. Sqlrand: Preventing sql injection attacks. *In Proceedings of the 2nd Applied Cryptography and Network Security (ACNS) Conference. Yellow Mountain, China. 292–302*, 2004.

[10] Steven Cook. A web developer's guide to cross-site scripting. *SANS Institute*, 2003.

[11] Danny Goodman and Michael Morrison. *JavaScript Bible*. Wiley Publishing, Inc., 2004.

[12] Jeremiah Grossman. Cross site scripting worms and viruses, the impending threat and the best defense. *WhiteHat Security*, 2006.

[13] W. G. Halfond and A. Orso. Amnesia: Analysis and monitoring for neutralizing sql-injection attacks. *In Proc. of the IEEE and ACM Intern. Conf. on Automated Software Engineering (ASE 2005), pages 174Ű183*, Nov, 2005.

[14] T Jim, N Swamy, and M Hicks. Defeating script injection attacks with browser-enforced embedded policies. *To appear in the 16th International World Wide Web Conference*, 2007.

[15] Elias Levy and Ivan Arce. New threats and attacks on the world wide web. *IEEE SECURITY & PRIVACY*, 2006.

[16] Frank S. Rietta. Application layer intrusion detection for sql injection. *Proceedings of the 44th annual Southeast regional conference ACM-SE 44*, 2006.

[17] D. Scott and R. Sharp. Abstracting application-level web security. *Proc. 11th IntŠl World Wide Web Conf., ACM Press, New York*, 2002.

[18] D. Scott and R. Sharp. Developing secure web applications. *IEEE Internet Computing*, 2002.

[19] Kevin Spett. Sql injection, are your web application vulnerable. *Whitepaper, SPI Dynamics, Inc.*, 2005.