blog.sqreen.com

OWASP Top 10 Cheat Sheet - Sqreen Blog

About the Author

13-16 minutes

In recent times, hacks seem to be increasingly prevalent, not to mention severe. What's more, it doesn't matter whether you're a small player or a big name corporation such as LinkedIn or Yahoo! If you develop web-based applications, there's the strong possibility that your application is vulnerable to attack.

Not sure why someone might attack *your* application? Attacks can have wide-ranging motivations; from something as simple as getting a product or service for free, to corporate espionage and industrial-scale blackmail.

Attackers can be seeking to acquire greater security access, steal some, or all, of your users' access credentials or financial details. They may be looking for compromising information, or to steal trade secrets.

Regardless of motivation, what's important is that your application may be vulnerable. So, in this post, I want to help you be better prepared.

To do that, I'll first step you through the anatomy of an application vulnerability, so that you know what *can* go wrong. Then, I'm going to step you through several ways in which you can reduce your application's vulnerability. In doing so, we'll also reduce your application's level of vulnerability.

What Is An Application Vulnerability?

An application vulnerability is a weakness that can be exploited to compromise an application. These attacks target the confidentiality, integrity, or availability (known as the "CIA triad") of an application, its developers, and users.

There are a large number of web application weaknesses. But, the best source to turn to is the OWASP Top 10.

1. Injection

The first vulnerability relates to trusting user input. An injection happens when an attacker sends invalid data to the application with an intent to make the application do something that it’s ideally not supposed to do. One example of this is SQL Injection attacks against your database. Some web applications use SQL database and use user input in SQL queries. In such cases, if the user input is not handled carefully and is directly used in the SQL query, it could lead to a SQL Injection attack.

For example, if the SQL query is display details for a user based on username give as input:

select * from userdetails where username='';

Here, <user_name_input> will be replaced by what the user types in the form field and submit. If an attacker gives the input as ‘ **or 1=1 —**, then the SQL query would look like this:

select * from userdetails where username='' or 1=1 --;

And this query would display all details of all the users. There are many ways SQL injection can be used. If successful, a SQL Injection can lead to discovery and leakage of information, the ability to change sensitive data, or to side-step authentication measures.

Data can come from form submissions through your web-based forms and API requests. It can also come from internal users. Regardless, if the data come from anywhere outside of your application, it can **never** be trusted. That's the first rule of web application security: "***Never trust user input!***"

There are several ways to mitigate this kind of attack. Survive The Deep End: PHP Security recommends the following:

- Always perform data filtering and validation
- Escape data
- Use parameterised queries
- Enforce the Principle of Least Privilege

2. Broken Authentication and Session Management

The second vulnerability covers the ability of users to manipulate, or workaround authentication mechanisms and sessions within an application. For example, is a user able to log in at one privilege level, yet elevate their privileges above and beyond what they should have?

OWASP lists several ways in which this attack can happen, including:

- User authentication credentials aren't protected when stored using hashing or encryption.
- Attackers can guess or overwrite credentials through weak account management functions.
- URL exposes session IDS.
- Lack of authentication when accessing privileged pages or functions.
- Session IDs are vulnerable to session fixation attacks.
- There's no timeout for Session IDs.
- Passwords, session IDs, and other credentials are sent over unencrypted connections

- Poor logout management

  To mitigate this kind of attack:

- Implement strong password policies and storage mechanisms
- Ensure that sessions, regardless of language:
- Timeout
- Are not able to be fixated
- Are rotated
- Use secure connections
- Implement secure logout management.
- Use encrypted communication (SSL/TLS).

3. Cross-Site Scripting (XSS)

This vulnerability occurs at the browser-level when user data is rendered without being escaped or validated. Attackers could injection malicious scripts to retrieve sensitive information or manipulation the way the application looks.

For example, if the application takes Username as input and greets the user with their name. If you give “**Bob**” as input in the form, output on the page could say something like this:

Hello Bob! Welcome to this application!

But if you inject something malicious instead of a valid username, let’s say, “**alert(document.cookie)**“, then when this input is rendered for output, it would display the cookie value.

This type of attack often results in:

- Hijacked sessions
- Defaced websites
- Users redirected to unintended destinations

To prevent this type of attack, again, remember the golden rule:

> **Never** trust data from outside your application

With that in mind, Sitepoint recommends three key ways to prevent XSS attacks:

1. Data validation
2. Data sanitization
3. Output escaping

4. XML External Entities (XXE)

You can find this vulnerability in applications that are poorly and insecurely processing XML data. Many applications use XML to transfer data to and from browser and servers. There are different features and specifications of XML and some of these and dangerous. Attackers can manipulate with the XML data to perform malicious tasks.

Risks of XXE vulnerability:

- Access to files on the server and confidential information.
- Access to back-end systems that can’t be accessed from outside the local/restricted network.
- Denial of Service.
- Perform Server-side Request Forgery.

Some straightforward ways of preventing XXE attacks are:

- Patching and updating all XML libraries and processors.
- Disabling XML external entity and DTD.

- Allow-listing and deny-listing of inputs, keywords, functions, etc.

5. Security Misconfiguration

This vulnerability occurs when configurations have not been security hardened. As the application keeps getting complex, there’s a lot of things to take care. And if you don’t give enough attention to detail, you might end up with security misconfigurations. This vulnerability depends on the way things are built and maintained.

This can include:

- Using outdated or insecure server and application configurations.
- Not protecting files and directories from being served by a web server
- Not removing default or guest accounts in a database
- Leaving unnecessary operating system ports open
- Using outdated software libraries

To protect against this kind of attack, for every part of your application, be that a server, language runtime, or operating system, ensure that it is suitably hardened, based on recommended best practices. If it is an external service, refer to their documentation and other material to ensure that they provide a hardened service and stay up to date.

6. Sensitive Data Exposure

This vulnerability relates to any access to information by anyone who shouldn't have access to it such as credentials, credit card details, etc. This includes data rendered in a browser or API response, data in logs and regular backups, and data sent between the client and the server.

OWASP has five excellent recommendations for protecting against this kind of attack:

1. Make sure you encrypt all sensitive data at rest and in transit
2. Don't store sensitive data unnecessarily
3. Ensure strong standard algorithms and strong keys are used, and proper key management is in place

4. Ensure passwords are stored with an algorithm specifically designed for password protection, such as *bcrypt*, *PBKDF2*, or *scrypt*
5. Disable autocomplete on forms collecting sensitive data and disable caching for pages that contain sensitive data

7. Broken Access Control

You can find this vulnerability in web applications that have user-specific control. An application can have 3 types of users in general:

- Anonymous users (users without logging in).
- Normal users (logged in but have minimum privileges and functions)
- Admin users

Each of these types of users has access to different data, features and functions. Broken access control weakness exists when a flaw in applications' logic would allow one type of user to get access to some other type of user.

One simple example of broken access control is when applications use different URLs for different features and don't authenticate users. Let's say you are using an application that requires you to log in to use it. If you log in as a normal user, the application would redirect you to ***www.example.com/dashboard.php***. And if you log in as an admin user, the application would redirect you to ***www.example.com/admin.php***. Let's say you log in as a normal user and the application redirects you to ***dashboard.php****.* But then you manually change the URL to ***www.example.com/admin.php.*** If the application doesn't do certain checks, you, as a normal user would have access to admin privileges.

Hence, broken access control could lead to risks from getting access to all of the applications' information to bringing down the whole application down. You can prevent broken access control risks starting with:

- Implementing lease privileges.
- Use multi-factor authentication for high valued accounts.
- Use Access control lists and role-based

authentication.

- Remove unnecessary users and services.
- Implement rate limits

8. Insecure Deserialization

This vulnerability exists where the data is being serialized and deserialized. Serialization means breaking data into a form for different purposes for transfer or storage. And deserialization is putting the broken data back together. Insecure deserialization occurs when the application puts back data together without checking for maliciousness or change in data.

Let’s take an example and see how this can be dangerous. Suppose you have an application that has user data along with a flag that indicates whether a user is an admin or not. When the application serializes data, it could be stored under different fields, for example, **name=Bob, email=bob@abc.com, admin=false**.

Now, the application would put this data together when it needs. If an attacker changes the data, then it could lead to security incidents. An attacker can create his own account and change the admin flag to **true,** to get admin access. If the application deserialized this data insecurely, then an attacker would get access to all the admin functions. And this is as scary as you can imagine. Attackers use insecure deserialization mostly for remote code execution.

You could start by implementing the following to prevent insecure deserialization:

- Integrity checks using hashes or signatures.
- Restricting or monitoring network.
- Avoiding serialization and deserialization where not necessary.

9. Using Components With Known Vulnerabilities

This vulnerability, as the name says, is where one or more components upon which an application depends, has a known vulnerability. This can include a third-party software library, a version of the web server or even in the operating system itself.

To protect against this kind of vulnerability:

- Always ensure that the components which you use are the most recent available
- Implement a security scanner which looks for known issues in your application and its related components
- Regularly install and update patches.

10. Insufficient Logging & Monitoring

Web application security is not a one-step process and doesn't end after you implement measures. You have frequently, or even better continuously monitor your application. Insufficient logging and monitoring will stop you from identify incidents earlier before they do any real harm. You can use attack indicators and signature along with logging and monitoring to identify possible security attacks and prevent them from succeeding. This will also help you in knowing your application's security and improving it.

You can use attack indicators and signature along with logging and monitoring to identify possible security attacks and prevent them from succeeding.

In Conclusion

That concludes our discussion on the OWASP top 10 vulnerabilities and how CTOs can protect their applications against each of them. If this is your first time considering application vulnerability, it's just the beginning of your journey.

However, you now have a good understanding of what makes an application vulnerable, as well as how to create ones which are more secure.

Moreover, like any other skill, security takes time to master. What's more, given that the internet's protocols were never designed with security in mind, security needs to be as fundamental to your applications as testing. However, by doing so, the possibility of your applications suffering a security breach is greatly reduced.

Shameless plug: Check out Sqreen **if you don't have time to protect your application against each individual OWASP top 10 vulnerability. Get real-time security monitoring and protection for your apps.**

Matthew Setter is an independent software developer and technical writer. He specializes in creating test-driven applications and writing about modern software practices, including continuous development, testing, and security.