Common Web Vulnerabilities

If we were performing a penetration test on an internally developed web application or did not find any public exploits for a public web application, we may manually identify several vulnerabilities. We may also uncover vulnerabilities caused by misconfigurations, even in publicly available web applications, since these types of vulnerabilities are not caused by the public version of the web application but by a misconfiguration made by the developers. The below examples are some of the most common vulnerability types for web applications, part of OWASP Top 10 vulnerabilities for web applications.

Broken Authentication/Access Control

`Broken authentication` and `Broken Access Control` are among the most common and most dangerous vulnerabilities for web applications.

`Broken Authentication` refers to vulnerabilities that allow attackers to bypass authentication functions. For example, this may allow an attacker to login without having a valid set of credentials or allow a normal user to become an administrator without having the privileges to do so.

`Broken Access Control` refers to vulnerabilities that allow attackers to access pages and features they should not have access to. For example, a normal user gaining access to the admin panel.

For example, `College Management System 1.2` has a simple Auth Bypass vulnerability that allows us to login without having an account, by inputting the following for the email field: `' or 0=0 #`, and using any password with it.

Malicious File Upload

Another common way to gain control over web applications is through uploading malicious scripts. If the web application has a file upload feature and does not properly validate the uploaded files, we may upload a malicious script (i.e., a `PHP` script), which will allow us to execute commands on the remote server.

Even though this is a basic vulnerability, many developers are not aware of these threats, so they do not properly check and validate uploaded files. Furthermore, some developers do perform checks and attempt to validate uploaded files, but these checks can often be bypassed, which would still allow us to upload malicious scripts.

For example, the WordPress Plugin `Responsive Thumbnail Slider 1.0` can be exploited to upload any arbitrary file, including malicious scripts, by uploading a file with a double extension (i.e. `shell.php.jpg`). There's even a Metasploit Module that allows us to exploit this vulnerability easily.

Command Injection

Many web applications execute local Operating System commands to perform certain processes. For example, a web application may install a plugin of our choosing by executing an OS command that downloads that plugin, using the plugin name provided. If not properly filtered and sanitized, attackers may be able to inject another command to be executed alongside the originally intended command (i.e., as the plugin name), which allows them to directly execute commands on the back end server and gain control over it. This type of vulnerability is called command injection.

This vulnerability is widespread, as developers may not properly sanitize user input or use weak tests to do so, allowing attackers to bypass any checks or filtering put in place and execute their commands.

For example, the WordPress Plugin `Plainview Activity Monitor 20161228` has a vulnerability that allows attackers to inject their command in the `ip` value, by simply adding `| COMMAND...` after the `ip` value.

SQL Injection (SQLi)

Another very common vulnerability in web applications is a `SQL Injection` vulnerability. Similarly to a Command Injection vulnerability, this vulnerability may occur when the web application executes a SQL query, including a value taken from user-supplied input.

For example, in the `database` section, we saw an example of how a web application would use user-input to search within a certain table, with the following line of code:

Code: php

```
$query = "select * from users where name like '%$searchInput%'";
```

If the user input is not properly filtered and validated (as is the case with `Command Injections`), we may execute another SQL query alongside this query, which may eventually allow us to take control over the database and its hosting server.

For example, the same previous `College Management System 1.2` suffers from a SQL injection vulnerability, in which we can execute another `SQL` query that always returns `true`, meaning we successfully authenticated, which allows us to log in to the application. We can use the same vulnerability to retrieve data from the database or even gain control over the hosting server.

We will see these vulnerabilities again and again in our learning journey and real-world assessments. It is important to become familiar with each of these as even a basic understanding of each will give us a leg up in any information security realm. Later modules will cover each of these vulnerabilities in-depth.

Questions

Answer the question(s) below to complete this Section and earn cubes!

+ 1 To which of the above categories does public vulnerability 'CVE-2014-6271' belongs to?

Submit your answer here... Submit Hint

Previous Next

Go to Questions

Table of Contents

Introduction to Web Applications
- Introduction
- Web Application Layout
- Front End vs. Back End

Front End Components
- HTML
- Cascading Style Sheets (CSS)
- JavaScript

Front End Vulnerabilities
- Sensitive Data Exposure
- HTML Injection
- Cross-Site Scripting (XSS)
- Cross-Site Request Forgery (CSRF)

Back End Components
- Back End Servers
- Web Servers
- Databases
- Development Frameworks & APIs

Back End Vulnerabilities
- Common Web Vulnerabilities
- Public Vulnerabilities

Next Steps
- Next Steps

My Workstation

OFFLINE

Start Instance

∞ / 1 spawns left