# Web Security Basics

In the early days of the Internet, the vast majority of websites were static. This means the content served to a user was largely fixed and identical for everyone, regardless of who was viewing it or when. The primary flow of information was in a single direction, exclusively from the server to the browser. User interaction was minimal, mostly limited to clicking links. Authentication and login mechanisms were either absent or rudimentary. Due to this lack of interactivity and personalized content, a compromise of an early website generally did not expose critical user or secret business information.

Today, the Web is dominated by web applications: highly interactive and complex systems that extend far beyond simple content delivery. These applications are the backbone of e-commerce, social media, and business operations. A bidirectional flow of information is now the norm, requiring constant interaction and communication between the user (client) and the server. This heavy communication extensively on Application Programming Interfaces (APIs) to facilitate the exchange of data and services. Modern web applications are composed of multiple, interconnected components that work together (e.g., Databases, Scripting Code, Mail Servers, Cloud Storage, etc.). Web pages are dynamically generated in real-time. Their content is highly personalized and dependent on factors like the user's authentication status and the specific systems making the request. Crucially, web application typically store, process, and transmit critical information, including personal data, financial records, and proprietary business secrets. This makes robust security and authentication non-negotiable, as a compromise can have significant, high-impact consequences.

## 1.1 The Core Security Challenge: Arbitrary Input

The main security problem facing modern web application sis the handling of arbitrary input. Attackers can deliberately manipulate inputs to a web application, such as URL parameters, POST request bodies, cookies, or HTTP headers, to alter the application's intended behavior. This manipulation can be used for various malicious purposes, including bypassing security controls, fraud, impersonation, data theft and so on.

It is critical to enforce strict security and validation over every piece of data a user provides. The basic defense mechanism must revolve around the fundamental acknowledgment that a web application can receive arbitrary, potentially hostile inputs. This necessitates proactive measures, which include:

- *Handling User Authentication Data*: Ensuring that mechanisms prevent one user from successfully logging in or acting as another user.
- *Handling Possible Malformed Inputs*: Validating and sanitizing all inputs to prevent data that is structurally incorrect or contains malicious code from being processed.
- *Handling Potential Attacker Actions*: Implementing controls to prevent denial-of-service (DoS) style actions, such as slowing down or overwhelming the system through multiple subsequent, rapid operations.

- *Administrator Actions to Log Application Activity*: Maintaining comprehensive logs of application activity to monitor for suspicious behavior and provide an audit trail for forensic analysis.

**Handling User Authentication Data.** Securing a web application begins with reliably determining and controlling who a user is and what they are allowed to do. This involves three interconnected concepts: Authentication, Session Management, and Access Control.

**(1)** Authentication is the process of correctly verifying a user's claimed identity. While the lowest degree of access is anonymous, this is largely avoided or restricted in modern applications that handle sensitive data. Credential-based authentication is the most straightforward and common method, and relies typically on a username and password. However, relying solely on standard username/password combination is often considered weak due to their susceptibility to various attacks (e.g., brute-forcing, dictionary attacks, or credential stuffing). To counter these weaknesses, multifactor authentication (MFA) is widely employed. MFA requires the user to provide two or more distinct verification factors.

**(2)** Once a user is authenticated, Session Management provides a powerful way for the application to handle multiple, subsequent requests from the same user without requiring them to log in again for every action. Applications typically generate a unique, cryptographically secure session token that identifies the user's active session. This token is then transmitted (usually via a cookie or HTTP header) with every authenticated communication between the browser and the server. Because this token is the *de facto* identity of the user for the duration of the session, the security of these tokens is crucial. If an attacker steals a valid session token (a process known as session hijacking), they can impersonate the legitimate user.

**(3)** Finally, Access Control determines which contents, resources, or functions an authenticated user is permitted to utilize or view. By default, access control ensures that unauthorized users cannot access specific contents meant only for logged-in or administrative users. Access control is also used to limit the actions of authenticated users based on their role. For example, in a banking application, money transfers might be limited to users who have completed extra verification steps, or an administrative panel may only be accessible by users with the "Administrator" role.

**Handling Malformed Inputs.** A fundamental step in securing a web application is the rigorous evaluation of all data submitted by the user. Arbitrary input can be deliberately malformed or crafted to trick the application into parsing unexpected data or executing hostile commands. To mitigate this, applications employ several strategies. Many forms put limitations on what the user can insert (e.g., character limits, allowed formats), while others may employ hidden form values: data sent with the form that the user is not intended to see or deliberately manipulate. However, these checks are easily bypassed by savvy attackers who can manipulate the raw HTTP request, meaning server-side validation is always mandatory. The primary defensive techniques for handling malformed inputs are:

- *Reject Known Bad*: This technique involves blacklisting specific input patterns, keywords, or characters known to be associated with attacks. The application scans the input for malicious signatures, such as SQL keywords ( `SELECT` , `UNION` , `DROP` ) or script tags ( `<script>` ) associated with SQL Injection or Cross-Site Scripting (XSS) attacks. Blacklisting is easily bypassed by attackers who user obfuscation. For example, blocking `SELECT` can be circumvented by using `SeLeCt` or encoding techniques. Attackers may use a NULL-byte attack (injecting `%00` ) in the input string. In some older or poorly written applications, the server-side logic might stop processing the string at the first null byte, causing the security filter to not analyze the rest of the string. However, the subsequent application logic (like a database query) might still execute the full, malicious payload.

- *Accept Known Good and Sanitization*: This approach is significantly more robust than blacklisting because it defines and accepts only a safe subset of possible inputs. Accept Known Good employs a whitelist of explicitly accepted characters, formats, or values. If an input contains anything outside this approved list, it is rejected. This is often the most effective way to filter input, especially for fields with known constraints (like a phone number or a specific set of drop-down options). Sanitization provides automatic fixes to input that is potentially malicious but cannot be automatically be rejected. Instead of rejecting the whole input, sanitization modifies the dangerous parts: for instance, replacing `<` , `>` , and `&` with their harmless HTML entities ( `&lt;` , `&gt;` , `&amp;` ) before the data is stored or displayed.

- *Safe Data Handling and Semantic Checks*: These techniques focus on processing data securely, regardless of the input source. Safe Data Handling means utilizing formally correct coding procedures to prevent the application's underlying code from misinterpreting user input as code. The best example is using parameterized queries (or prepared statements) for all database interactions. This technique ensures user input is always treated purely as data and can never be executed as part of an SQL command, virtually eliminating most forms of SQL Injection. While an input may be syntactically correct and not contain malicious characters, it can still be used in malicious ways based on the context of the transaction. A semantic check ensures that the data makes logical sense in the context of the application's business logic. For example, a user attempting to purchase an item might pass a price parameter that is technically a valid number, but a semantic check would reject it if the price doesn't match the actual, server-side price of the product.

It is tempting to believe that simple data sanitization at the point of entry is sufficient to secure a web application. This approach assumes that one only needs to worry about the raw input data and not the internal web application logic, leading to the idea that all checks can be performed as a single, monolithic, a priori phase. However, relying on a single input validation step is insufficient. Web applications perform multiple data processing steps across various internal components. Data that is safe at the entry point might become hostile after an intermediate processing step. The variety and complexity of potential attacks are too high to be fixed in one phase. Furthermore, the sanitization techniques effective against one attack (e.g., removing angle brackets for XSS) might not be compatible or effective against another (e.g., preventing a logic flaw). The correct approach is Boundary Validation, which means making checks in multiple stages throughout the application's lifecycle. The necessary checks depend heavily on which components are being run by the web app. For instance, input passed to a database requires parameterized queries, while input being rendered back to a user's browser requires HTML entity encoding. Checks are performed by considering the input that is passed at each state: at the external boundary and at every internal boundary.

**Handling Attackers.** When a web application is directly targeted, defensive strategies shift from passive prevention (like input validation) to active defense, monitoring, and reaction. The core objective is to reduce the attacker's motivation and opportunity by implementing robust error handling, detailed logging, and a swift incident response plan.

- *Handling Errors*: When an unexpected event occurs (such as trying to visit a non-existent page or a server-side exception), the application must avoid leaking sensitive information. The primary goal of secure error handling is to mask internal server details. Generic error messages (e.g., "An unexpected error occurred") should be shown to the user, while detailed information (e.g., stack traces, database query errors, server paths) must be logged internally but never displayed publicly. Leaking such details can provide attackers with clues about the application's architecture and potential vulnerabilities.
- *Maintaining Audit Logs*: Audit logs are indispensable tools for monitoring system activity, conducting forensic analysis, and detecting suspicious behavior. Logs should comprehensively monitor critical user activities, including: successful and failed login attempts, payment transactions and other business-critical actions, and blocked access attempts and security violations. Every log record must contain essential information to identify and trace the event, such as timestamp, source IP address, and the session ID or user account.
- *Alerting Administrators and Reacting to Attacks*: Automatic alerts are essential for enabling prompt administrative reactions to ongoing attacks. Alerting systems are typically triggered by anomalies, such as:
  - Usage Anomalies: Sudden spikes in failed login attempts or unusually high request volumes.
  - Business Anomalies: Transactions outside typical parameters (e.g., an unreasonably large money transfer).
  - Attack Signatures: Requests containing known malicious strings or attempts to manipulate hidden form data.

  Upon detection, automated or manual reactions can be deployed: intentionally slowing down requests from a suspicious source to hinder automated attacks, temporarily or permanently banning specific source IP addresses or user accounts. Automatically logging out users after a certain time or number of failed attempts.
- *Securing Administrative Functionalities*: The administrative components of a web application represent a high-value target and a severe vulnerability if compromised. Administrative Functionalities are often assumed to be more secure than public-facing parts, but vulnerabilities here can allow an attacker to compromise the entire application (e.g., gain full database access or execute remote code). Administrators possess powerful privileges, such as the ability to influence other users' sessions, view sensitive information, or grant powerful access rights. If an attacker gains access to the administrator panel, they acquire these same, potentially devastating, capabilities.

## 1.2 Lab - 1

There are three tasks in this Lab. The tasks concern web app spidering and the basics of requests/response manipulation. The tasks have been designed to make you acquire practical skills on the course topics.

### Task 1 - Spidering

Perform a web spidering of the following web application: `http://zero.webappsecurity.com/`. To log in, you can use the following credentials: user - *username*, password - *password*. Report all the useful information that you can find, such as

- Pages found automatically.
- Pages not found automatically (if any - specify why it is difficult to find them automatically).

**Solution.** The primary security flaw in this application is the insecure handling of user authentication and sensitive data transmission. Burp is initially unable to map the site structure until a successful login is achieved. Upon performing the login, Burp automatically builds the site map, but the fundamental security vulnerability becomes immediately apparent. The HTTPS request used for authentication transmits the `session_id` , `username` , and `password` in plaintext. This means that if an attacker were to intercept the GET request, they would gain full, unauthorized access to the private user session. Further exploration of the links discovered by Burp reveals pages containing extremely sensitive financial information, including full credit card numbers and records of bill payments, which are completely accessible to the compromised session. Finally, by analyzing the POST request made when attempting to log in with different credentials, it is clear that the correct, working credentials are still being stored as a persistent Cookie, introducing yet another layer of risk by making the account vulnerable to session fixation or theft.

### Task 2 - Basic Web Hacking

Complete the levels of the Natas wargame until Natas 5 (included) by using Burp to manipulate requests when needed. By solving each level, you obtain a password to proceed to the next one. The wargame starts here: `http://natas0.natas.labs.overthewire.org/`, with user - *natas0* and pass - *natas0*. To access the next level, just go to the corresponding URL. The goal of the task is to briefly describe the solution for each level.

**Solution.** These solutions require various web application penetration testing techniques to uncover hidden data and bypass access controls.

**(1)** *Level 0* ( `0nzCigAq7t2iALyvU9xcHlYN4MlkIwlq` ): The password is easily retrieved by simply inspecting the webpage's source code, where it is left in an HTML comment within the response.

**(2)** *Level 1* ( `TguMNxKo1DSa1tujBLuZJnDUlCcUAPlI` ): Although the webpage has right-clicking disabled, the password can still be found by inspecting the raw HTTP response using a proxy like Burp. The password remains visible in an HTML comment.

**(3)** *Level 2* ( `3gqisGdR0pjm6tpkDKdIWO2hSvchLeYH` ): The page displays as empty, and Burp confirms it only shows a single-pixel image sourced from `files/pixels.png` . This path suggests a directory. Navigating to the parent directory, `files/` , reveals a directory listing that contains the `users.txt` file, which stores the plaintext password for the next level.

**(4)** *Level 3* ( `QryZXc2e0zahULdHrtHxzyYkj59kUxLQ` ): This page is also empty and offers no clues through direct inspection. The solution lies in checking the `robots.txt` file. This file, intended for web crawlers, contains a hint: `/s3cr3t/` . Following this link reveals the existence of a `users.txt` file, where the plaintext password is stored.

**(5)** *Level 4* ( `0n35PkggAPm2zbEpOU802c0x0Msn1ToK` ): The application states that access requires the request to originate from `http://natas5.natas.labs.overthewire.org/` . This is a Referer header check. The user must intercept the HTTP request, modify the `Referer` header to the required URL, and then refresh the page (which is necessary to avoid re-sending authentication details) to display the password.

**(6)** *Level 5* ( `0RoJwHdSKWFTYR5WuiAewauSuNaBXned` ): The page denies access based on a "not logged in" status. By intercepting the HTTP request, we can see a Cookie named `loggedin` with a value of `0` . Simply changing the cookie's value from `0` to `1` and resending the request successfully bypasses the access control, revealing the password.

### Task 3 - Managing Requests/Responses

Your final goal is finding a secret flag (password) by analyzing the following web application: `https://jupiter.challenges.picoctf.org/problem/28921/`. The flag has the format `picoCTF...` .

**Solution.** Upon initial inspection of the page, there is no information hidden within the HTML source code itself. However, the message displayed on the page clearly indicates the required action: the server is performing a check on the client's identity. The solution is straightforward: the client must change the `User-Agent` header in the HTTP request to the required value, `picobrowser` . Once this change is made and the request is sent, the server grants access and the solution is displayed: `p1c0_s3cr3t_ag3nt_84f9c865` .