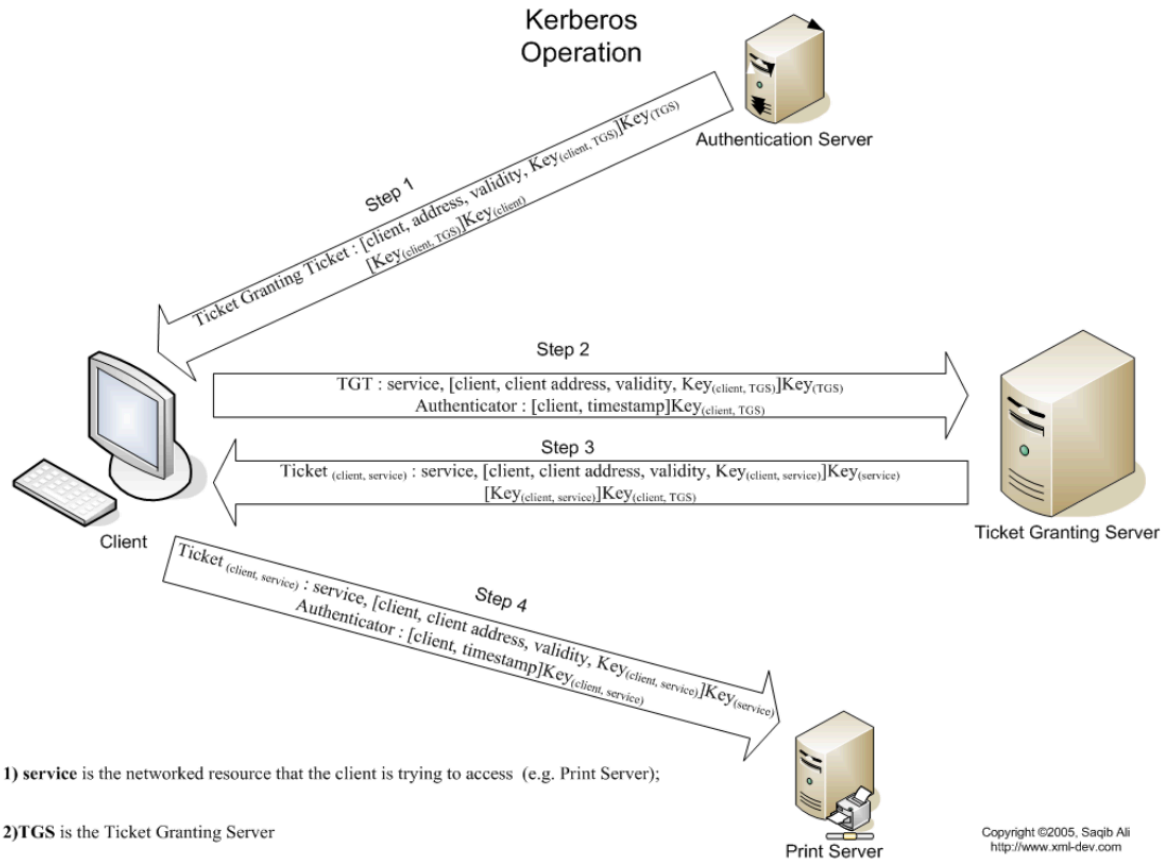
Web Service Infra

Authentication who is accessing the service?

- every service responsible for devising own means of securely ide
  Authorization what is the user allowed to do?

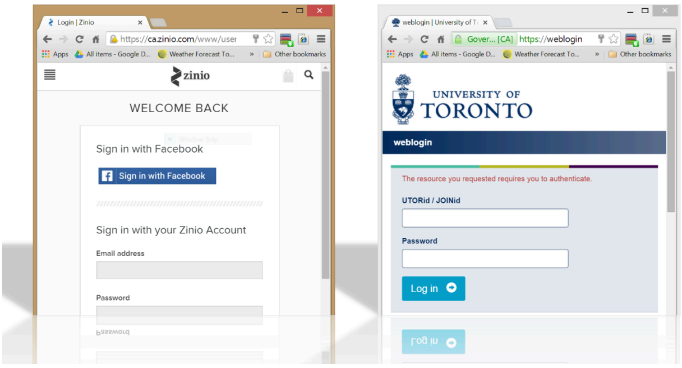
Kerberos System

- **authentication & authorization** service that controls access to network resources
  - authentication server
    - knows all user passwords
    - provides ticket-granting ticket to user, certifying user authorized to use ticket granting service
  - ticket granting server - user authorization
    - provides service tickets to user, indicate user authorized to use some service
    - knows secret keys of all services



Federated Identity

- manage authentication service = complex
- Federated Identity service - maintains uname & pwd, handles takss of maintaining authentication service
- Multiple services rely on that one Federated Identity provider to help authenticate users that interact with the services.



Typical login flow with Federated Identity:

1. The user authenticates (logs in) to the identity service provider.
2. The service provider sends the user an unforgeable "token" that contains the user's identity.
3. User presents the token to the service they want to access, which accepts it in lieu of authenticating the user.
4. The service now has a certified identity of the user.

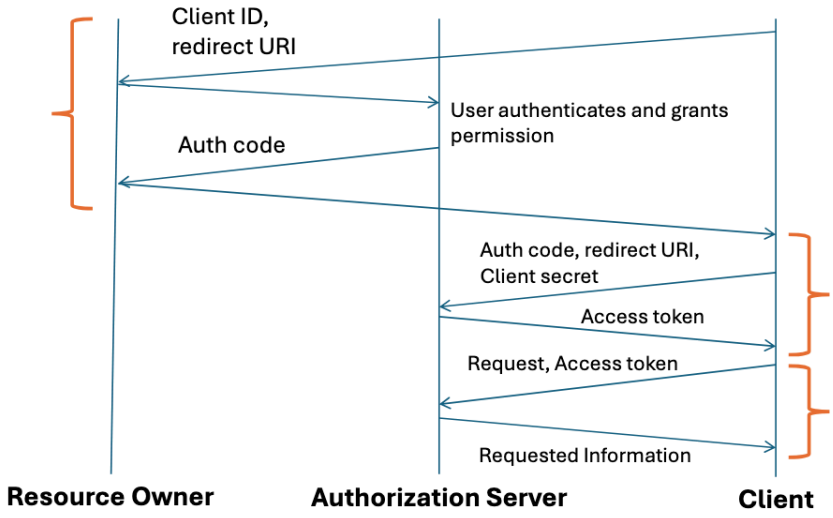
Example: OAuth

???

OAuth supports several authorization protocols. One of the commonly used ones is called **Authorization Code Grant.**

There are 3 parties in any authorization/authentication event

1. The **Resource Owner** (*i.e.*, the User)
2. The **Authorization Server** (*i.e.*, Facebook)
3. The **Client** (*i.e.*, Zinio)



Web Auth: Secure Password

Password downfall

- easy passwords
- re-use passwords
- authentication is not mutual (user sending passwd to correct site?)

Storing Passwords

- systems should never store passwords as PT
  - hashed using one-way hash; only hash is stored
- login procedure
  - input passwd, system computes hash, compare with stored hash

Password Salts

- if one P & $H(P)$ pair is compromised, all users with same password is vulnerable

- **salt** - random value (different for each user) added to the password before hashing
  - **User 1:** H("**aa**" + "password") = `8d43705d3febf70f815754adfc1d2b569b16aae289f3677a0cb866e6f65a52b1`
  - **User 2:** H("**bb**" + "password") = `8ebd81d07993a0af13e7ffcc1cc8c6caef091ae6a379f42ad66ad82ce9bb5843`

  The salt value is stored in the password file, usually in plaintext
  - The is exactly the same idea as an Initialization Vector
  - Do salts make it harder to break a single password?

bcrypt

- hashing function written for **secure pw hashing**
  - strong hashing; built-in salt value, very slow

```
$2a$12$R9h/cIPz0gi.URNNX3kh2OPST9/PgBkqquzi.Ss7KIUgO2t0jWMUW
\__/\/ \____________________/\_____________________________/
Alg Cost      Salt                        Hash
```

Where:

- `$2a$` : The hash algorithm identifier (bcrypt)
- `12` : Input cost (2^{12} i.e. 4096 rounds)
- `R9h/cIPz0gi.URNNX3kh2O` : A base-64 encoding of the input salt
- `PST9/PgBkqquzi.Ss7KIUgO2t0jWMUW` : A base-64 encoding of the first 23 bytes of the computed 24 byte hash

One-Time Passwords (TOTP/HOTP)

- static password can be broken given enough time; OTP changes every time it is used
- requires
  - pre-shared secret
  - 2way communication
  - OTP can be implemented using challenge-response authentication

Example:
Server encrypts 'n' using shared key, sends E(n) to client
Client decrypts 'n', adds 1, and send E(n+1) to server
Client is authenticated if server is able to decrypt the message and get 'n+1'

- *What's this?*

Multi-Factor Authentication

- **authentication factor**
  - something user knows(password), has, is, can do
- multi-factor → using multiple authentication factors

Smart Cards

- good option for security token
  - contains secure microcontroller that is hardened against tampering
  - contains keys & performs cryptographic operations
- have the card sign a randomly generated string (used for authentication)

Biometrics

- face detection, fingerprints, retina

Turing Test

- use a CAPTCHA (Completely Automated Public Turing Test to tell Computers and Humans Apart)

Establish a Session

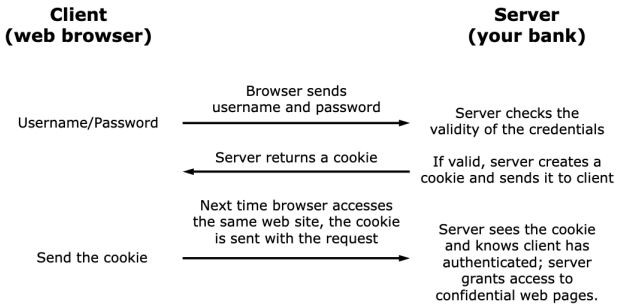
- HTTP - identity of user needs to be re-transmitted from the browser to the server on every request (did not support long-lived sessions)

Basic Web Authentication

- every time you access a page, credentials (remembered by web page) is transmitted to web server → giving additional opportunities to compromise them
- Browsers retain password information indefinitely (until the application is closed)

Cookie-Based Authentication

- upon successfully authentication, server generates a session token to browser, which saves it as a **cookie**
- next time browser visits same server, send the same cookie back to server, which uses cookie as authentication token
- cookie also used for tracking HTTP requests belonging to the same user



Advantages of Cookies

- cookies not passed in URL (not recorded in browsing history)
- cookies do not reveal the password (adversary learn the value of cookie, but do not knwo the user's username and password)
- cookies have **expirty time** after which server will not accept
- cookies make HTTP stateful
  - "Stateful" means that a system or protocol maintains information about the ongoing interaction between the client and server across multiple requests.

Cautions of Cookies

- should not be easy to guess a valid cookie

- Cookies should not be used for authentication without SSL, or else the cookies can be easily stolen
  - A web server can specify a policy with the cookie (*e.g.*, browser should send the cookie over SSL only)
  - Cookies for authentication should be specified “SSL only”
- Cookies should not be made to last indefinitely (*i.e.*, without an expiry time); that would create a long window of vulnerability for an attacker to steal a browser’s cookie file

Vulnerabilities and Defences

- browser-based attacks
  - drive-by-downloads, XSS, phishing, persistent popups, browser history theft
- sites have web browsers execute small JS programs
  - inside HTML tags

```
<html>
<head>
<script type="text/javascript">
function myfunction() {

    alert("HELLO");

}
</script>
</head>
```

Malicious JavaScript

- JS code has access to cookies, browsing history available in web browser
- what if malicious site can read the cookies of your bank site

Fictitious Javascript Attack (XSS)

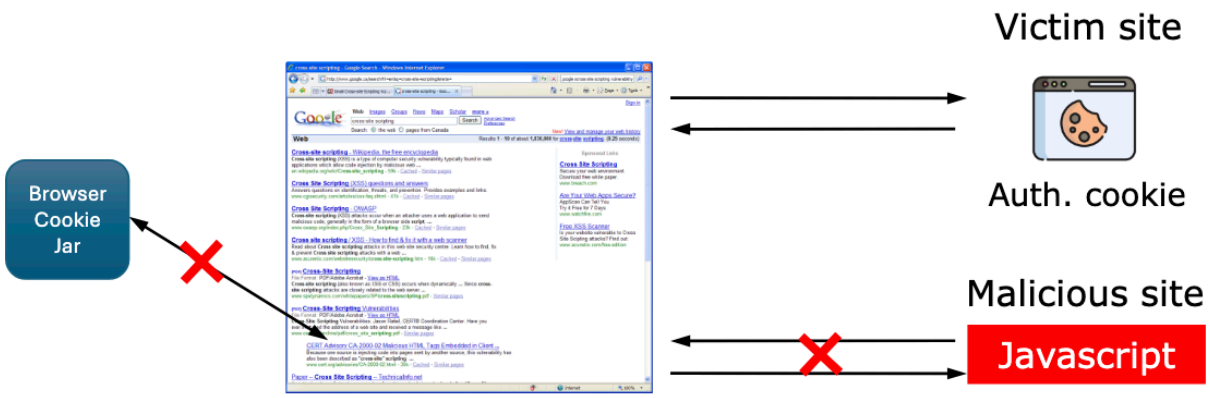
- browser logs onto victim bank sites and gets authentiction cookie
- ayyacker tricks user visit malicious site, sends script to user's browser
- user browser executes script, gets auth cookie and sends it to malicious site (identity forgerty)

Stealing Data

- JS can read cookie by accessing the variable `document.cookie`
- Sending Data
  - script can then incorporate the cookie value into an HTTP request and send request back to attacker's website
  - put it into the URL of a GET request
  - put it into the form data of a POST request

SOP (Same Origin Policy)

- SOP dictates that scripts from one origin (website) cannot access/set the properties of a document from another origin
- applies to cookies, documents and its properties, making requests to other web sites, running scripts from documents etc
- 2 URL have same origin if
  - they use same protocol (http/https)
  - same hostname
  - same port number
- browser isolates different sites into different identities
- Note that two domains under the control of the same owner are not considered the same origin i.e. www.amazon.com and www.amazon.co.uk are not considered the same origin



- HTTP GET requests - For accessing pages, account information, or triggering actions through URL parameters
- HTTP POST requests - For submitting forms, making transactions, or other state-changing operations

XSS (Cross-Site Scripting)

- vulnerability allowing malicious user to inject script code into web page
  - Type 1/ Reflected
    - attacker crafts URL, user clicks on URL for successful attack
    - website not modified
    - SOP not violated; script originates from same site as cookies (server executes request)
    - JS can perform arbitrary actions on HTML page returned by victim site
      - web server sanitize; value in name is name not script
      - **poor input validation**
  - Type 2/Persistent
    - attacker posts arbitrary script on vulnerable site
    - user visits site
    - website is modified
    - input validation → post free content but not script

The fundamental vulnerability is that the server is directly including user input in the HTML output without proper sanitization or encoding. This is why input validation, output encoding, and Content Security Policy are critical safeguards against XSS attacks.

Defenses

- input filtering

- convert all special characters before sending it to a user
- use whitelisting instead of blacklisting → allow a set of safe characers
- HTTP_only cookies
  - web browser will not let any JS read the cookie even from same site (website can tag certain cookies as being inaccessible to JS)

SQL Injection

- injecting code into db query commands

The first step in evaluating your application for a potential vulnerability is usually performing tests on the input, to see if SQL Injection is possible.

Username: jskule' and 1=1 --
Password: **********

```
query = 'SELECT * FROM users WHERE ' +\
            'username="' + username + '" AND ' + \
            'password="' + hashedPassword
```

SELECT * from users WHERE username='jskule' and 1=1 -- AND password='1$3f28ab3f9....'

This can also be done with URLs (GET/POST):

https://www.ece568.ca/API/login?username=jskule%27%20and%201=1%20--&password=...

```
query = 'SELECT * FROM users WHERE ' +\
            'username="' + username + '" AND ' + \
            'password="' + hashedPassword
```

SELECT * from users WHERE username='jskule' and 1=1 -- AND password='1$3f28ab3f9....'

HTTP Response Splitting

CSRF (Request Forgery)