Department of Computer Science
University of California, Los Angeles

Computer Science 144: Web Applications

Spring 2025
Prof. Ryan Rosario

Lecture 16: May 28, 2025

Outline

CI/CD on Google App Engine

First, your node project must have a `build` and `start` script. This can be as simple as `npm run build` and `npm start`.

You will need to enable the following GCP APIs in your Cloud project:

1. Cloud Build APIs
2. App Engine Admin API
3. Cloud Resource Manager API

CI/CD on Google App Engine (contd.)

Next, create a `.cloudbuild.yaml` file in the root of your project.

```yaml
steps:
  - name: 'gcr.io/cloud-builders/npm'
    args: ['install']   # npm install

  - name: 'gcr.io/cloud-builders/npm'
    args: ['run', 'build']  # If using a build step: npm run build

  - name: 'gcr.io/cloud-builders/gcloud'
    args: ['app', 'deploy', '--quiet']  # gcloud app deploy

timeout: '900s'
```

CI/CD on Google App Engine (contd.)

Next, create a trigger in Cloud Build.

In this interface, you can connect to your Github repo.

CI/CD on Google Kubernetes Engine

The process is similar for GKE. Create the following `.cloudbuild.yaml` file in the root of your project.

```yaml
steps:
  - name: 'gcr.io/cloud-builders/docker'
    args: ['build', '-t', '<REGION>-docker.pkg.dev/<PROJECT_ID>/<REPO>/<IMAGE>:<TAG>', '.']

  - name: 'gcr.io/cloud-builders/docker'
    args: ['push', '<REGION>-docker.pkg.dev/<PROJECT_ID>/<REPO>/<IMAGE>:<TAG>']

  - name: 'gcr.io/cloud-builders/kubectl'
    env:
      - 'CLOUDSDK_COMPUTE_REGION=<REGION>'
      - 'CLOUDSDK_CONTAINER_CLUSTER=<CLUSTER_NAME>'
    args: ['apply', '-f', 'k8s/']

images:
  - '<REGION>-docker.pkg.dev/<PROJECT_ID>/<REPO>/<IMAGE>:<TAB>'
```

You'll need to grant Cloud Build permission to access GKE

HTTPS and TLS

The original Internet was a bit like the Wild Wild West. It was a network where everybody talked to each other. Data packets could be intercepted and read by anyone with rudimentary software.

Important transactions require:

1. Confidentiality
2. Message/data integrity
3. Authentication
4. Authorization

Cryptography to the Rescue

We can use encryption to prevent eavesdropping by making data unreadable.

Two keys are involved – a public key and a private key (Asymmetric cipher) to establish trush and open a session.

Once established, the sesson key is used for symmetric encryption of all messages.

Visiting an HTTPS Site: ClientHello

When you visit the HTTPS site, the browser:

1. sends over a list of supported ciphers to the server
2. the highest TLS version it supports
3. a random number (nonce) to the server (client random)

Visiting an HTTPS Site: ServerHello

The server responds with:

1. the highest TLS version they both support.
2. the cipher suite that the server and client support
3. a random number (nonce) to the client (server random)
4. the certificate with other information such as public key, domain, validity period, and a signature.
5. optional key exchange information
6. a ServerHelloDone message

Visiting an HTTPS Site: Verification and Key Exchange

The client verifies that the certificate is valid using the public key of the CA that issued it, checks that the time period is valid and that the domain name matches.

If so, the client generates a **pre-master secret** (a random number) and encrypts it with the server's public key. Only the server can decrypt it, using its private key.

A session key is then derived from the pre-master secret.

Visiting an HTTPS Site: Finished Handshake

The client sends a `Finished` message encrypted with the session key.

The server decrypts it and sends its own `Finished` message encrypted with the session key.

At this point, both client and server have established a secure channel using the session key (symmetric). All subsequent messages are encrypted using this key.

TLS 1.3 simplifies this process further.

Transport Layer Security (TLS)

TLS provides confidentiality, integrity, and authentication on the web.

TLS only protectes data in transit, not at rest.

HTTPS is just HTTP over TLS (previously SSL).

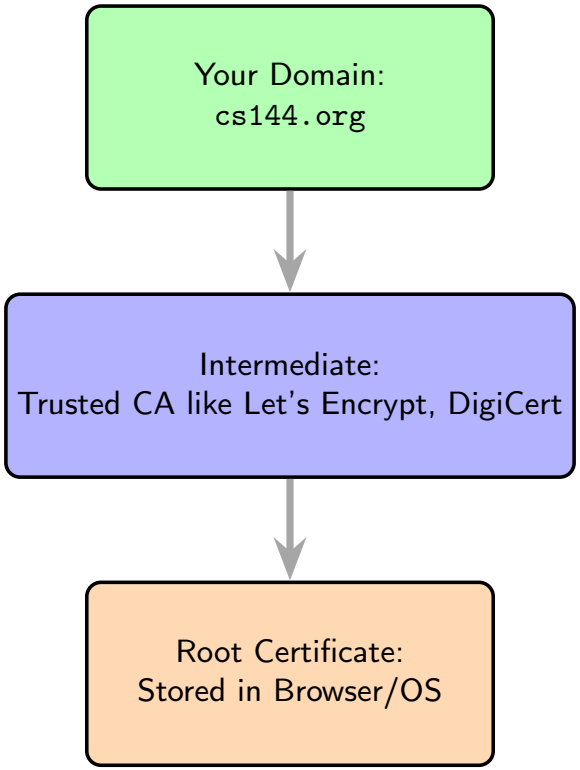
Certificates

A certificate verifies that the public key you receive when you visit an HTTPS enabled website actually belongs to the entity that hosts the site, and not some nefarious actor.

The certificate is "signed" by an organization whose job it is to make sure that public/private key pairs are only distributed by the site that is offering them to the user.

Certificates (contd.)

The chain of trust:



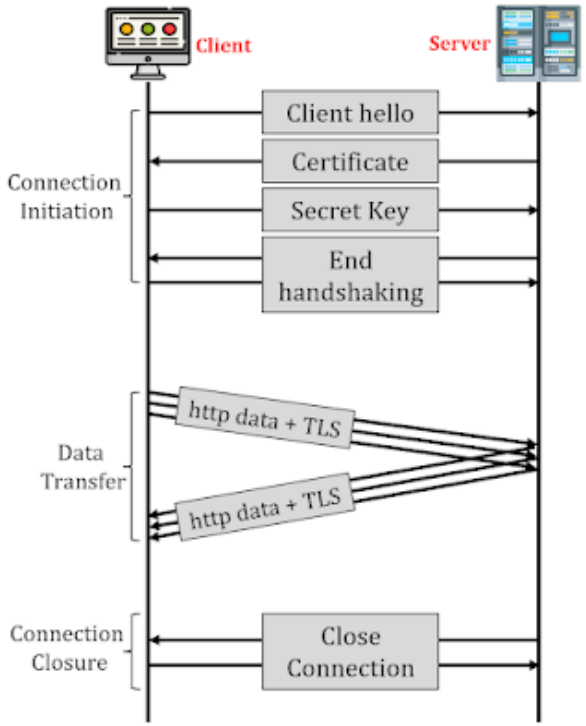
Certificates (contd.)

A certificate can be obtained by a Certificate Authority (CA) like VeriSign.

Or, you can use a self-signed certificate where the owner and issuer are the same. Many domain name providers and web hosts provide free certificates. Let's Encrypt is a popular option.

Let's Encrypt automates issuance with ACME (Automatic Certificate Management Environment) protocol

Certificates (contd.)



Source: Chirag Bhalodia

This is why when you create a password for a site you are visiting, the password can be sent in plain text – it's actually encrypted!

Modern TLS (1.3) prefers ephemeral Diffie-Hellman (ECDHE)

HTTP Strict Transport Security: HSTS

"From now on, always connect to this site over HTTPS and never over HTTP."

It prevents downgrade attacks, ensures automatic HTTPS upgrade, prevents users from visiting HTTP version of site.

Server HTTP response header:

```
Strict-Transport-Security: max-age=31536000;
includeSubDomains; preload
```

HTTP Strict Transport Security: HSTS (contd.)

You will often see a **mixed content warning** when an HTTPS site tries to load resources over HTTP.

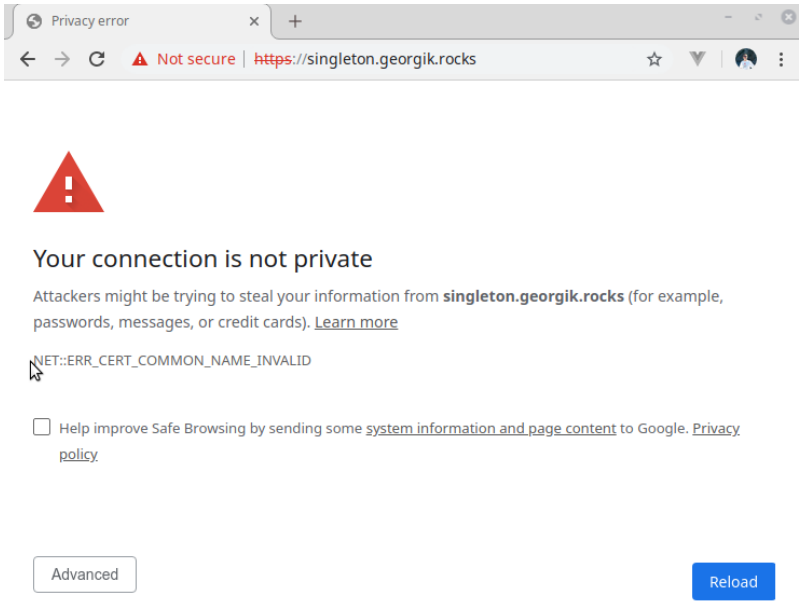
This violates confidentiality and integrity of HTTPS and could allow MITM attackers to tamper with content.

- Scripts, iframes and stylesheets are blocked (Active Mixed Content)
- Images and video are allows with a warning (Passive Mixed Content)

Everything should be loaded over HTTPS.

HTTPS is Ubiquitous

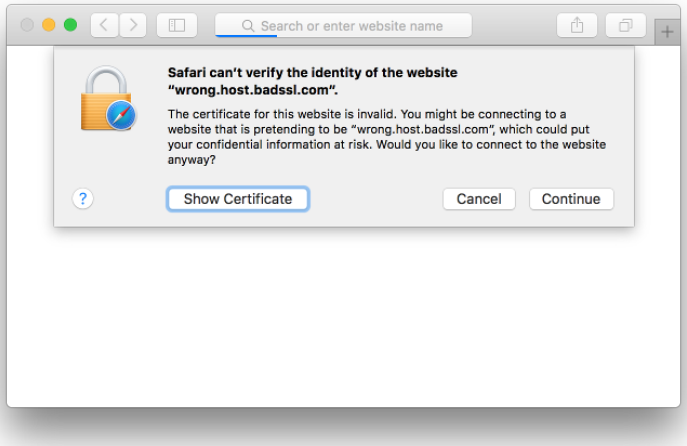
Approximately 90% of all web traffic uses HTTPS enabled by default. You may occasionally see this message (in Chrome):



Proceed with caution. Make sure to read why the certificate is invalid.

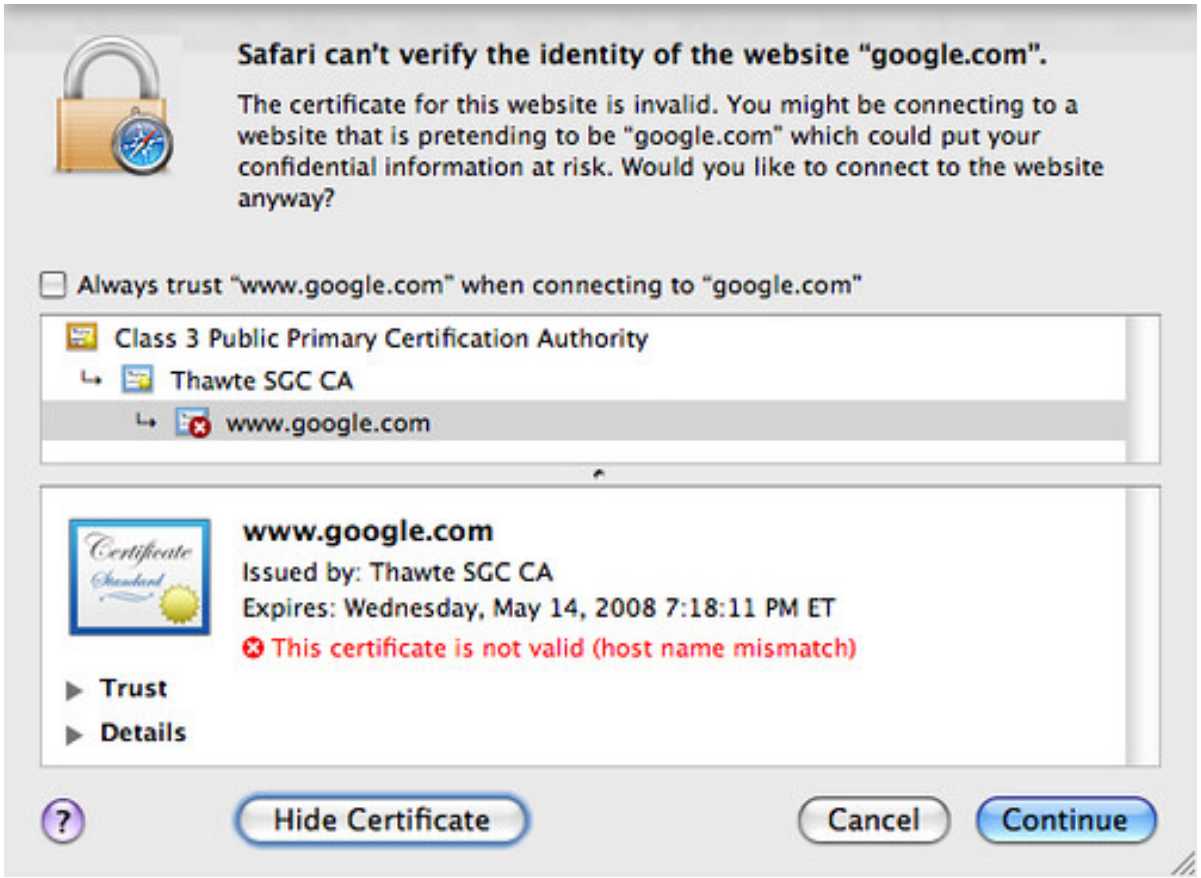
HTTPS is Ubiquitous (contd.)

In Safari:



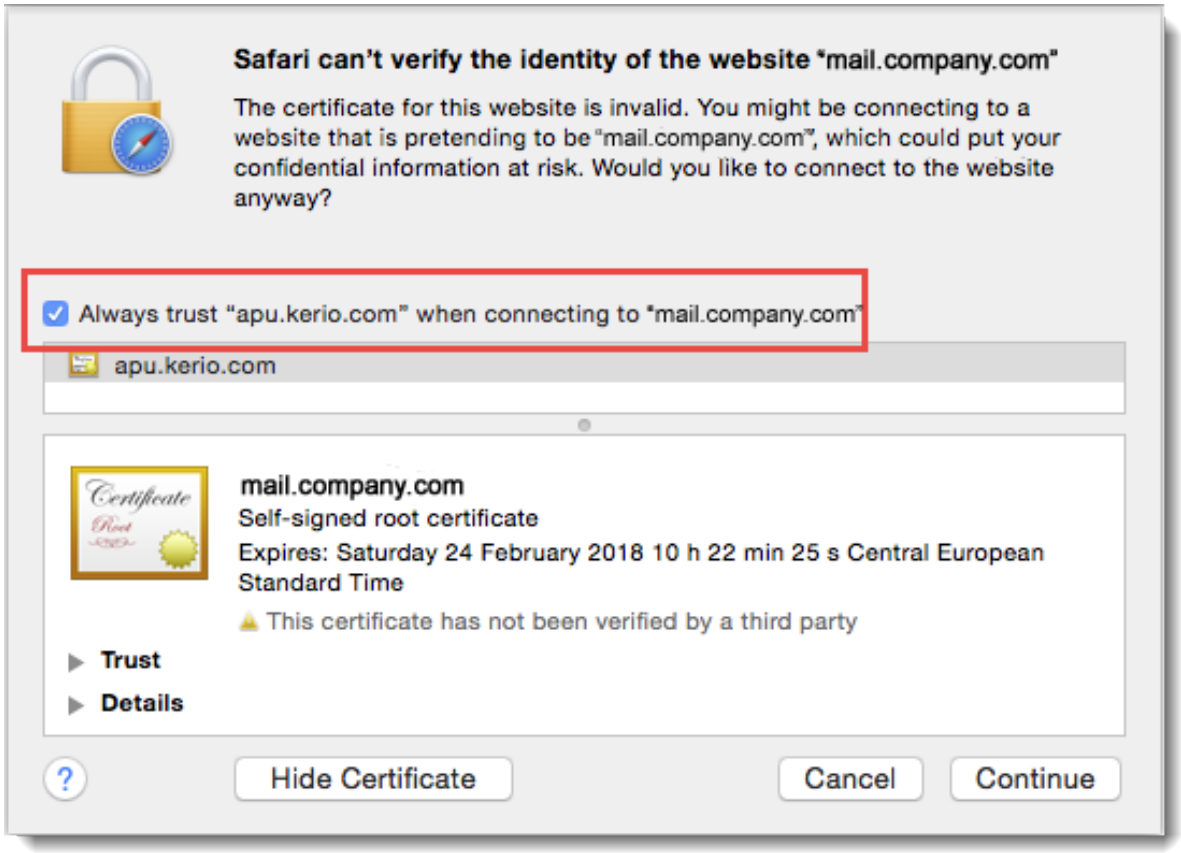
Proceed with caution. Make sure to read why the certificate is invalid.

HTTPS is Ubiquitous (contd.)



Proceed with caution. Make sure to read why the certificate is invalid.

HTTPS is Ubiquitous (contd.)



Proceed with caution. Make sure to read why the certificate is invalid.

Using Let's Encrypt

Using a Let's Encrypt certificate is pretty easy. See the directions for Ubuntu (GCE) here.

1. Install and run certbot
2. Modify NGINX virtual host configuration to use the certificate and keys
3. Allow HTTPS traffic through the firewall
4. Restart NGINX

Using Let's Encrypt (contd.)

```
server {
    listen 443 ssl;
    server_name yourdomain.com;
    ssl_certificate /etc/letsencrypt/live/yourdomain.com/fullchain.pem;
    ssl_certificate_key /etc/letsencrypt/live/yourdomain.com/privkey.pem;
}
```

The catch is that you **must** have a domain name.

Using Let's Encrypt (contd.)

For your final project, if you have a domain name, you can use it, or use a subdomain.

If you do not (you are not expected to), we can provide you with one on `cs144.org`.

Using Let's Encrypt (contd.)

In Express, it looks like this:

```
import fs from 'fs';
import https from 'https';
import express from 'express';
const app = express();

const options = {
  key: fs.readFileSync('/path/to/privkey.pem'),
  cert: fs.readFileSync('/path/to/fullchain.pem')
};

https.createServer(options, app).listen(443, () => {
  console.log('Secure server listening on port 443');
});
```

HTTPS on GKE

When using GKE and other GCP services, you can use a Google-managed certificate instead. The catch is that we must:

1. Request a static IP address for the Ingress.
2. Point a domain name to the static IP address.
3. Create a `ManagedCertificate` YAML file
4. Create an `Ingress` YAML file that references the certificate. Ingress is the point at which data entres the cluster. In Lecture 15, we used the Service for this.
5. Create a `FrontendConfig` YAML file to redirect HTTP to HTTPS.

HTTPS on GKE (contd.)

Example `Deployment` YAML file:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: chat-app
spec:
  selector:
    matchLabels:
      app: chat-app
  replicas: 1
  template:
    metadata:
      labels:
        app: chat-app
    spec:
      containers:
      - name: chat-app-container
        image: us-docker.pkg.dev/google-samples/containers/gke/chat-app:1.0
        ports:
        - containerPort: 1919
```

HTTPS on GKE (contd.)

The `Service` YAML file would look like the following. Note the `type` is different:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: chat-app-service # Renamed for consistency with Deployment
spec:
  selector:
    app: chat-app
  ports:
    - protocol: TCP
      port: 80 # Service port
      targetPort: 1919 # Container port
  type: ClusterIP # Essential for Ingress backends
```

HTTPS on GKE (contd.)

Then apply using:

```
kubectl apply -f deployment.yaml
kubectl apply -f service.yaml
```

HTTPS on GKE (contd.)

To request a static IP address:

1. Reserve a static IP address using `gcloud compute addresses create static-ip-name -global`
2. Fetch the address with `gcloud compute addresses describe static-ip-name -global`
3. Create an DNS A record pointing your domain to the static IP address.

HTTPS on GKE (contd.)

Example `Certificate` YAML file:

```yaml
apiVersion: networking.gke.io/v1
kind: ManagedCertificate
metadata:
  name: chat-app-certificate
spec:
  domains:
    - your.domain.com # Replace with your actual domain
```

HTTPS on GKE (contd.)

Then apply using: `kubectl apply -f certificate.yaml`

Wait for it to be `Active`:

`kubectl get managedcertificate chat-app-certificate -o yaml` (up to 2 hours)

Create an `Ingress` into your cluster. This was originally handled by the Service but it does not operate at the correct level to terminate HTTPS.

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: chat-app-ingress
  annotations:
    # Use a static IP (optional but recommended)
    kubernetes.io/ingress.global-static-ip-name: "static-ip-name" # Replace with your static IP name
    # Link your Google-managed certificate
    networking.gke.io/managed-certificates: "chat-app-certificate"
    # Automatic HTTP to HTTPS redirect
    networking.gke.io/v1beta1.FrontendConfig: "default-redirect-config" # We'll create this next
spec:
  rules:
    - host: your.domain.com # Replace with your actual domain
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: chat-app-service # Your ClusterIP Service name
                port:
                  number: 80 # The port your Service listens on
```

HTTPS on GKE

Finally, we need to force HTTPS by redirecting HTTP traffic to HTTPS. This is done using a `FrontendConfig` resource:

```
apiVersion: networking.gke.io/v1beta1
kind: FrontendConfig
metadata:
  name: default-redirect-config
spec:
  redirectToHttps:
    enabled: true
    responseCode: MOVED_PERMANENTLY_DEFAULT
```

HTTPS on GKE (contd.)

Then apply:

```
kubectl apply -f ingress.yaml
kubectl apply -f frontendconfig.yaml
```

HTTPS on Google App Engine

Google App Engine *automatically* provides HTTPS for your app using the GCP selected domain name (`*.appspot.com`)

How convenient!

Never DIY

There are some things in your web app that you should never DIY.

1. Authentication libraries
2. Password hashing
3. Token generation
4. Legal compliance (GDPR/CCPA)

Common Vulnerabilities

There are several vulnerabilities that come up with web applications. We will discuss the following:

1. Injection (briefly, as we covered in CS 143)
2. Client state manipulation
3. Cross-site scripting (XSS)
4. Cross-site request forgery (CSRF/XSRF)

SQL Injection

It is possible for a bad actor to steal, update or destroy data by inserting nefarious input into a SQL query when using strings. Suppose we have:

```
"SELECT uid, name FROM bruinbase WHERE uid = " + user_input + ";"
```

1. `0 OR 1=1`
2. `0 AND 1=0; INSERT INTO bruinbase VALUES ('424242424', 'Rejected from UCLA')`
3. `0 AND 1=0; DROP TABLE bruinbase`

What happens?

Command Injection

The same can happen with Unix commands running on a server hosting a web application.

```
system("cp secretFile.txt $userInput")
```

What if `$userInput` is a publicly accessible directory?

Preventing Injection Attacks

We can prevent SQL injection attacks by using *prepared queries* that escape user input:

```
cur.execute("""
  SELECT uid, last, first, mi, gpa, major FROM bruinbase
  WHERE major=%(major_input)s;
  """, {
    'major_input': request_args['major']
  }
)
```

Preventing Injection Attacks (contd.)

We can prevent command injection by refraining from using `eval()` (e.g. JavaScript, Python, Ruby, PHP, Perl, Lua) or `exec()` or `system()`.

Typically, these functions take raw user input and process them as commands without sanitizing it.

Famous SQL Injections

The Mueller investigation found that Russian operatives from GRU were able to steal information on millions of voters in Illinois by executing a SQL injection on the website for the Illinois State Board of Elections.

One Source: `https://chicago.suntimes.com/news/mueller-report-special-counsel-russia-hacking-illinois-state-board-elections/`

Famous SQL Injections (contd.)

Other famous SQL injection attacks:

1. 2021: 70GB of data was stolen from the right wing social network *Gab* using a SQL injection. The vulnerability was introduced by the CTO.
2. 2015: Personal data from 159,959 customers of TalkTalk was stolen via their login portal
3. 2012: Half a million login credentials, stored in plain text, from Yahoo! were stolen.
4. 2011: Sony Pictures Entertainment was hacked and 77 million accounts were stolen.
5. 2009: 130 million accounts were stolen from Heartland Payment Systems, a credit card processor.
6. 2005: A teenager broke into the site of a Taiwanese IS magazine to steal customers' information.

Client State Manipulation

This is a rather dumb one, but we will discuss it to be complete. Suppose we have the following HTML from an e-shop:

```
<form ...>
  <input type="hidden" name="price" value="5.00">
  <input type="submit">
  ...
</form>
```

What is the Problem Here? Cookies suffer the same fate.

Protecting against Client State Manipulation

Protecting against this vulnerability is mostly a no-brainer:

- Maintain most tangible state on the server (e.g. prices of items).
- Use session IDs to tie the state of the server with the client.
- Can also use signed state (e.g. HMAC). Include a signature with data and check the signature for tampering when returned to the server.

Cross-Site Scripting (XSS)

```
<body>
Welcome to {{ user_name }}'s Site!
</body>
```

What if the input is

```
Ryan Rosario<script src="do_something_bad.js"></script>
```

Now we have loaded a JavaScript file that does something bad, when the page loads.

Click here to read more about XSS.

Protecting against XSS

To protect against XSS, we could just prevent the user from using *any* HTML tags in their input, but for use cases like HTML mail, this will not work.

For standard use cases, it's best to escape <, >, $, and ".

It is very difficult to protect 100% against XSS attacks.

We can maintain an allow-list of HTML keywords rather than a deny-list. It is also important to use user input validation.

Cross-Site Request Forgery (XSRF/CRSF)

A reminder about cookies:

1. Arbitrary key-value pairs.
2. Returned to the server on every HTTP request.
3. Can be used to track a user's session.
4. Adheres to same origin policy. Cookies only sent back to their originators.

Can a malicious web site still see the cookies issued by a trustworthy site? Can an attacker trick the user and browser into doing something bad?

Cross-Site Request Forgery (XSRF/CRSF) (contd.)

Cross-site Request Forgery (CSRF) is a technique that enables attackers to impersonate a legitimate, trusted user.

Cross-Site Request Forgery (XSRF/CRSF) (contd.)

Scenario 1: Suppose we have a bank and we wish to make a transfer of $5,000 to a family member. We log in as usual and are offered a cookie. **We stay logged in.**

We could use a GET request to make the transfer (though it violates HTTP standard):

```
http://www.acmebank.com/transfer.php?to=5551212&amount=5000
```

Cross-Site Request Forgery (XSRF/CRSF) (contd.)

Then... a hacker sends us the following message.

```
To: Victim
Subject: A gift of flowers for you!

Hello victim,
We know your birthday is coming up and have a
special gift for you. Just click here to receive it!
```

The Click Here link takes us to

```
http://www.acmebank.com/transfer.php?to=1212121&amount=5000.
```

Cross-Site Request Forgery (XSRF/CRSF) (contd.)

Important: ACME Bank is happy to process the transaction because we have a cookie from ACME Bank that is being sent to ACME Bank.

Cross-Site Request Forgery (XSRF/CRSF) (contd.)

Scenario 2: Just as bad, the link sends us to `evil.com` containing the following HTML form:

```
<body onload="document.forms[0].submit()">
<form action="http://www.acmebank.com">
        <input type="hidden" name="amount"  value="1000">
        <input type="hidden" name="to" value="1212121">
</form>
</body>
```

Note that POST is not any more secure.

Cross-Site Request Forgery (XSRF/CRSF) (contd.)

Important: Again, ACME Bank is happy to process the transaction because we have a cookie from ACME Bank that is being sent to ACME Bank. It does not matter that the form lives on `evil.com`.

Cross-Site Request Forgery (XSRF/CRSF) (contd.)

Scenario 3: Or even worse, the transfer can happen automatically via a pixel.

```
<img src="http://acmebank.com/fundtransfer?acct=224224&amount=500000"
  width="0" height="0" border="0">
```

Protecting Against CSRF/XSRF

To protect against CSRF, we could use a version of "sending a password" on every request to the good site – a password that the evil site doesn't know.

Good site sends an *action token* (hash of a session ID) in a hidden form field.

When the user performs some action on the legit page, the token is sent to the server. The server checks it and should authorize the request.

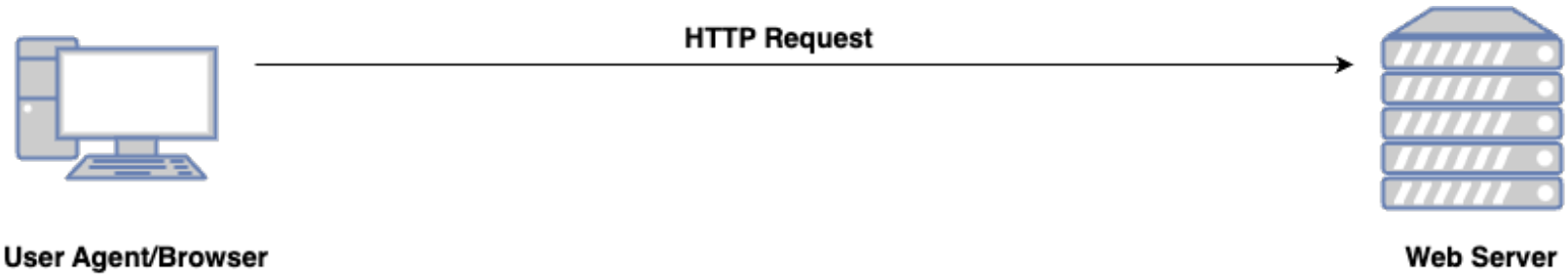
A nefarious web site will be missing this token, or it will likely be incorrect. The server will reject the request if the token doesn't match.

Protecting Against CSRF/XSRF (contd.)

User Agent/Browser

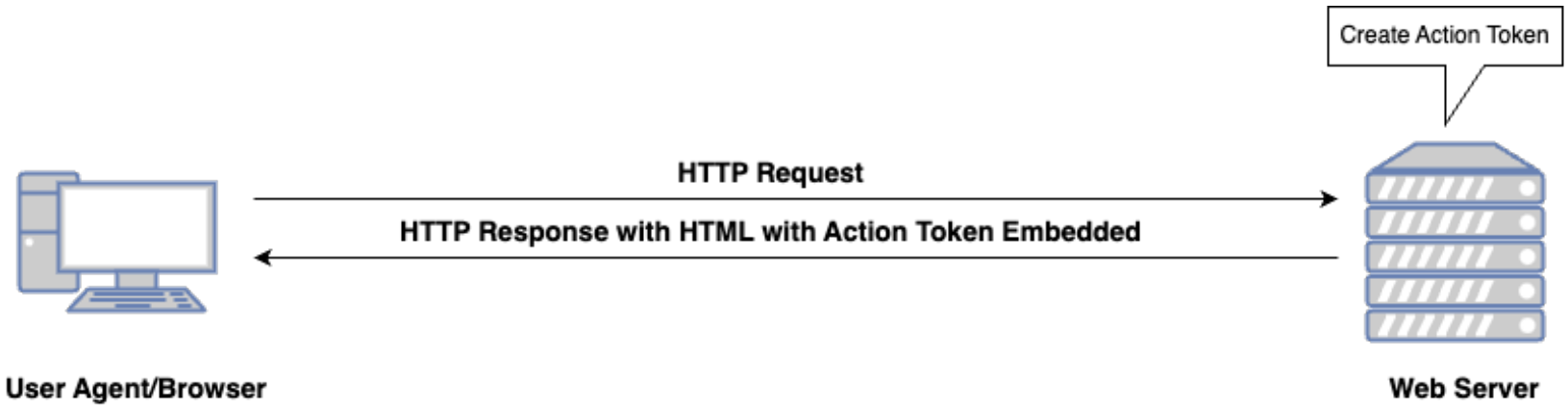
Web Server

Protecting Against CSRF/XSRF (contd.)

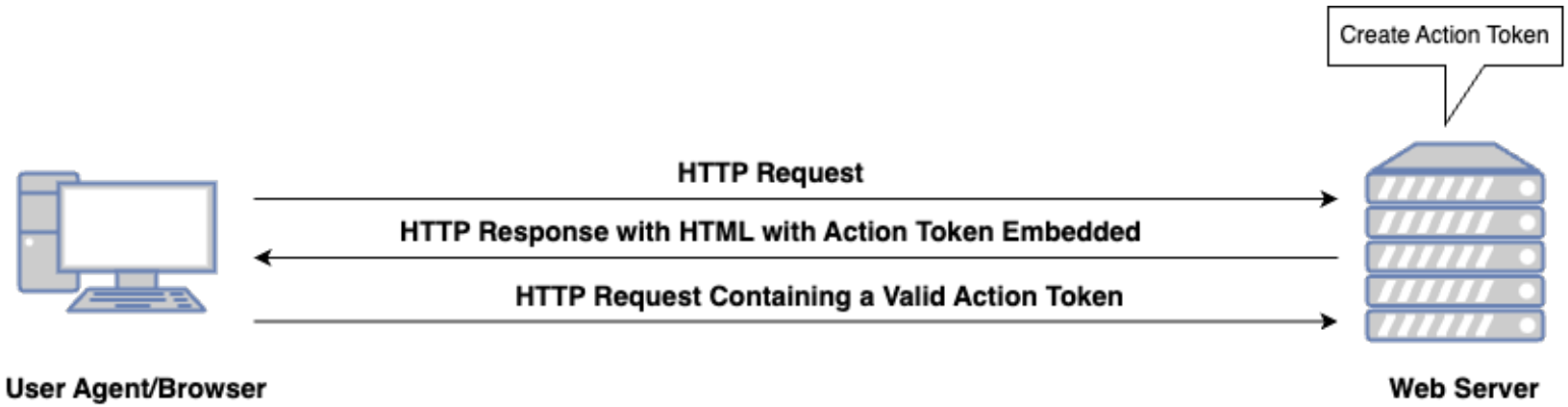


Protecting Against CSRF/XSRF (contd.)

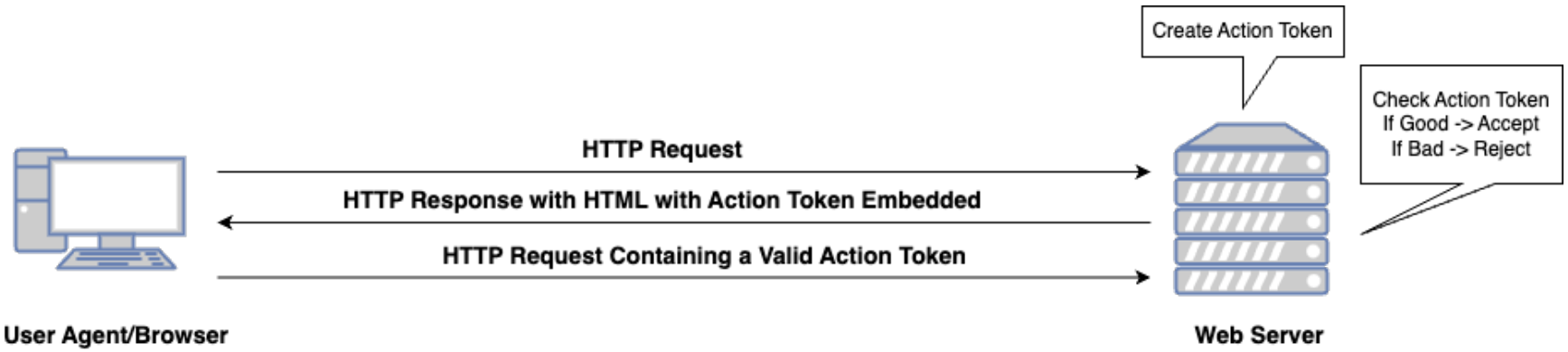
Protecting Against CSRF/XSRF (contd.)



Protecting Against CSRF/XSRF (contd.)



Protecting Against CSRF/XSRF (contd.)



Protecting Against CSRF/XSRF (contd.)

We still have a problem if the attacker has access to the legitimate stored web page or if the attacker can guess the token. But we will leave it at that.

Best Practices

Some general best practices to avoid these vulnerabilities:

1. Be very careful with your permissions.
2. Create users that are used only to serve content. These users include for databases, Unix, etc.
3. Encrypt user data that is lying around on the disk
4. Never store user passwords in cleartext. Always salt (bcrypt/scrypt) and/or pepper.
5. Specify what the user *can* do and keep it minimal, instead of specifying a long list of things they *cannot* do.
6. Use HTTPS when possible.
7. **Never, ever, trust user input!**

Security vs. Privacy

There is a subtle but crucial difference between security and privacy.

- Security is about protecting data from unauthorized access.
- Privacy is about controlling how your personal information is collected, used, and shared.
- Security measures can help protect privacy, but they are not the same thing.

Privacy

Privacy violations come in many different forms. The most common complaints regarding privacy violations these days include:

1. Using personal information to enrich a corporation (ads in our case).
2. Using personal information to train AI models.

Ads

Ads work as follows:

- Advertisers pay a fee to show ads to users. The fee can be based on impressions (CPM), clicks (CPC), or conversions (CPA).
- Advertisers provide ad creatives (images, videos, text) and targeting criteria (demographics, interests).
- Ad platforms (like Google, Facebook) use algorithms to match ads with users based on their profiles and behavior.
- Users see ads in their feeds, search results, or websites they visit.
- Targeting ads to niche populations, or too broadly, is expensive.

Third Party Cookies

Many domains know things about you. This is how Google, Facebook and Amazon are able to target ads to you on other domains.

How do they do it?

Third Party Cookies (contd.)

They use the concept of a **Third Party Cookie**.

Basically, there is some entity out there, call it A, that sets cookies on your user agent, and your user agent is consistently sending these cookies back to this entity A while you're on domains other than the A's.

Huh?

Third Party Cookies (contd.)

Suppose entity A (e.g. Facebook, Google) wants to track users across its partner's sites. The partner site B is usually a site that serves ads on behalf of A.

To participate in the partnership, B embeds a tiny image called a **pixel** on its web pages. The pixel is managed and hosted by A. Note that JavaScript and `iframes` can be used as well.

When a user requests B's web pages, several HTTP requests are executed for HTML, CSS, images, etc... **and** for this tiny image from A as well.

Many sites embed the pixel from A, and thus at some point the user accepts a cookie from A. On every request for the pixel, the cookie is sent back to A via HTTP request.

Third Party Cookies (contd.)

It's **third party** because the cookie is being set and passed without the user explicitly knowing it. The user believes it is only interacting directly with B, and not A.

A knows all of the pages that the user accessed on B, across all sites that partner with A.

This means that A knows what types of sites you visit, what you search for, and some other "crumbs" about your activity.

The Future of Third Party Cookies

Google has been trying to phase out Third Party Cookies due to privacy concerns, but the deprecation has been deprecated.

As of January 2024, Google has disabled third party cookies in Chrome to 1% of the user base.

Disabling third party cookies means that advertisers cannot track users across the Web as easily. Note that "first party" cookies issued by a website directly to a user are still valid.

Beyond Cookies

There are ways to identify users without cookies:

1. Browser fingerprinting
2. Canvas fingerprinting
3. Audio fingerprinting
4. Font fingerprinting

GDPR

GDPR is an EU regulation that protects the privacy and personal data of individuals. Personal data is anything that can identify a person, even pseudonymously. The principles are:

1. Lawfulness, fairness, transparency
2. Limitation of purpose
3. Minimize data collection
4. Accuracy
5. Retention liitation
6. Integrity and confidentiality
7. Accountability and documentation

Users must be informed and provide consent.

CCPA

CCPA is a California law that has been enforced since 2020. It applies to very large corporations that make 50%+ of their revenue from selling (sharing) personal data.

1. Right to know
2. Right to delete
3. Right to opt-out
4. Right to non-discrimination

Golden Rule

Privacy is perhaps the biggest factor that affects the integrity of your web app and your business.

- Do not collect more data than you need.
- Do not share data with third parties unless absolutely necessary.
- Be transparent about how you use and share data.
- Provide users with control over their data.