CSCC09F
Programming on the Web

Securing Web Apps

Security Properties,

Vulnerabilities, Defenses

Secure Web Apps

- When you visit an online banking site, to access services like bill-payments, what gives you the confidence to interact with such a site without worrying that your account will be drained by some scammer?
- What kinds of things do you look for on the browser window?
- When you read your GMail, is your session data private or could someone else be "listening in"?
- When you develop a Web app, how do you ensure that it can't be misused to steal user credentials, to corrupt app data, to deface the app's views?

Secure Web Apps

- Suppose you want to add security protection to a Web app (e.g. for user authentication to Eatz).
- What kinds of things might app-users be concerned about?
  - Confidentiality of their personal information such as username, password, e-mail address
  - Authenticity of the server (e.g. not a phishing site)
  - Integrity of their requests as received by your server, that is, the request is guaranteed not to be tampered or modified in transit (and likewise for server responses)
  - Privacy of their browsing data and behavior

Securing Web Apps

- Easy to overlook security issues when developing Web apps – first goal is usable functionality
- But, an insecure app can:
  - Expose user credentials and data to eavesdroppers
    - common credentials problem
  - Enable attackers to forge user requests, e.g. to modify app data, to perform requests not intended by user
  - Allow attackers to inject malicious code into user pages and even the MongoDB
- An app that doesn't protect user data and its own functional integrity will not be successful !

Security Issues for Web Apps

What are the biggest security issues for Web apps?

1. Unauthenticated servers (e.g. phishing sites) that trick users, eavesdroppers that snoop on user data, network attacks that can tamper with (modify) user data
2. SQL Injection
   - Can expose or corrupt app data/base, enable unauthenticated access to app
3. Cross-Site Request Forgery (CSRF)
   - Enables an attacker to make requests on behalf of user and have app server accept them as legitimate
4. Cross-Site Scripting (XSS)
   - Attacker injects JavaScript that can steal user credentials, perform unauthorized user transactions, deface app views

Phishing, Eavesdropping, Tampering

- Is Eatz vulnerable to phishing, eavesdropping, tampering?
  - YES
- Eatz runs on the http protocol, which provides no native protection against phishing sites, eavesdropping or data tampering
- HTTP does not ensure that when you connect to the Eatz app site it is genuine, and not a forgery e.g. for phishing
- An attacker running a tool like Wireshark could read all your request data including usernames and passwords
- A more sophisticated attacker could conduct a “man-in-the-middle” (MITM) attack to both eavesdrop and modify any of your HTTP messages/data

SQL Injection

- Is Eatz with MongoDB vulnerable to SQL injection?
  - note that Node.js can be used with SQL DB's also
- In a literal sense no, since there's no SQL, but could an attacker achieve an injection-type attack without SQL?
- A comparable attack action would be the evaluation of arbitrary JavaScript code as part of a MongoDB query
- A couple obscure approaches to accomplish this are referenced by OWASP, but at the moment, MongoDB seems to be generally resistant to injection-type attacks

Cross-Site Request Forgery (CSRF)

- Is Eatz vulnerable to CSRF?
- How could we test it?
- Use of client-side framework a bit of a challenge compared with traditional static client-side forms – templates full of model references, form with no submit button, etc
- Idea: copy EditView.html, convert it to a standard form with method and action fields, submit button, host it on mathlab Apache, submit target is Eatz Node.js port
- If it works, and user is already logged in on a different tab, then should be able to add new junk dishes!
- More useful to attacker … if Eatz had a dish rating/review system could submit lots of fake ratings/reviews to boost/harm reputation

Cross-Site Scripting (XSS)

- Is Eatz vulnerable to XSS?
- How could we test it?
- This one's pretty easy to test – need to pick an input field that isn't validated (or that won't filter out JavaScript) – choose the URL field
- If we can run any simple JavaScript here, then could arbitrarily extend it for more sophisticated attacks
- As test, let's try stealing the user's cookie values

```
<script>alert(document.cookie)</script>
```

- Oh ya!
- If we click the save button, this morphs from a reflected to a persistent XSS attack – stored in DB for future use!

Securing Web Apps: Strategy

- Multi-pronged approach to close holes in Eatz app:

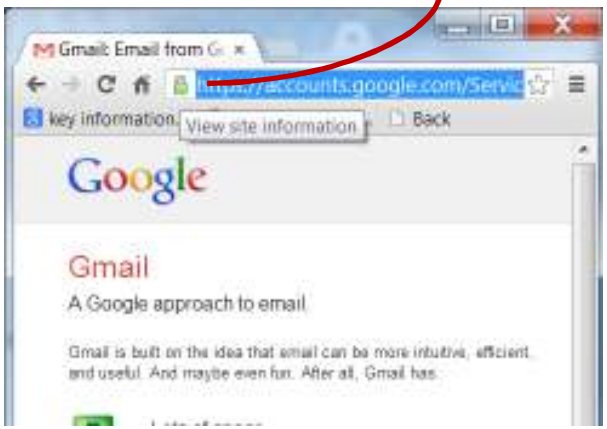
- Switch to secure HTTPS connections to prevent:
  - side-jacking: authentication cookie copied by attacker
  - mixed-content attacks which are vulnerable to tampering
  - SSL-stripping to downgrade connections for eavesdropping
- Sanitize user-input data to block malicious content
  - block XSS attacks
- "Multi-factor" authentication mechanism
  - block CSRF attacks

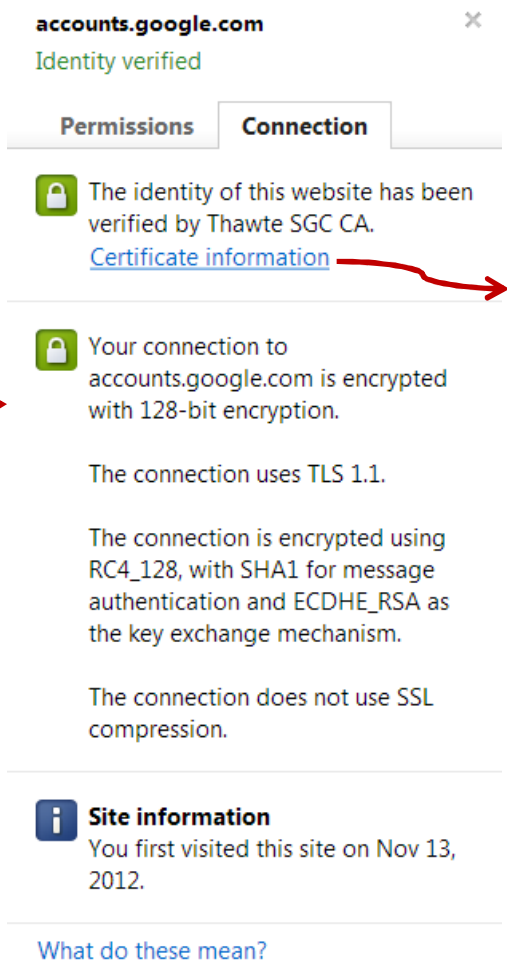
Secure Web Apps: HTTPS

- The HTTPS protocol is based on the SSL/TLS protocol, an Internet standard designed to guarantee several key properties that users expect from secure online services:
  - confidentiality – all data sent on an HTTPS connection is encrypted with a strong-encryption algorithm
  - authentication – the server with whom you communicate must provide a certificate to verify its identity and encryption key
  - integrity – the protocol uses strong message-integrity checks to ensure that messages cannot be altered in transit
- How do you know if you are using an HTTPS connection, and if it is legitimate?

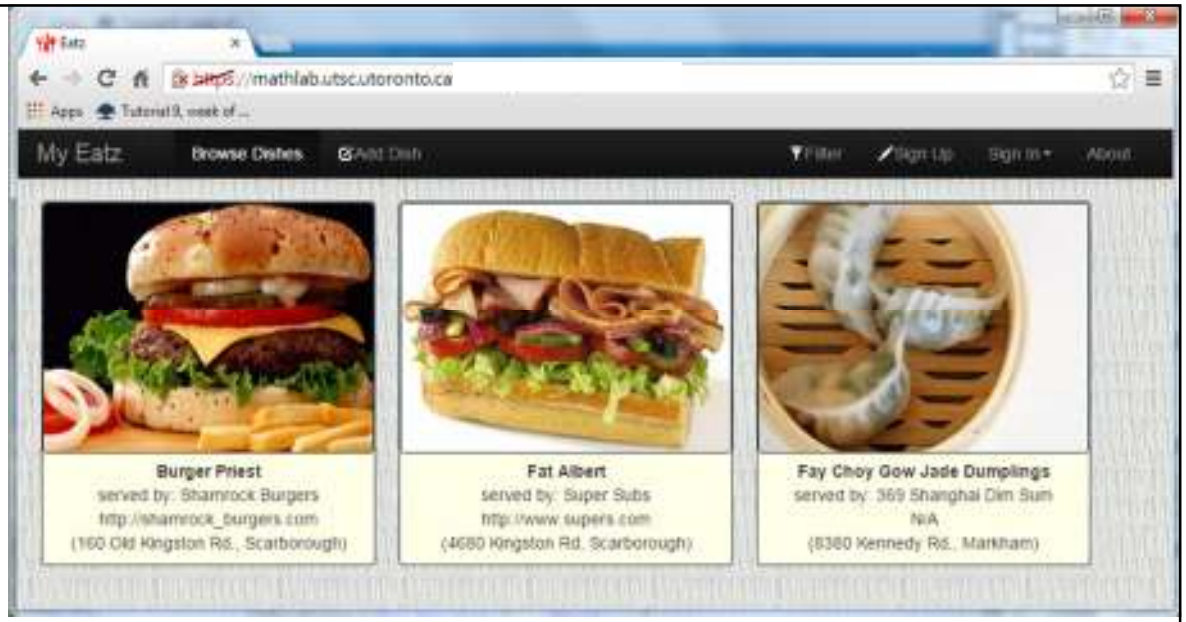
HTTPS

- How do you know if you are using an HTTPS connection, and if it is legit?
- Click on the padlock

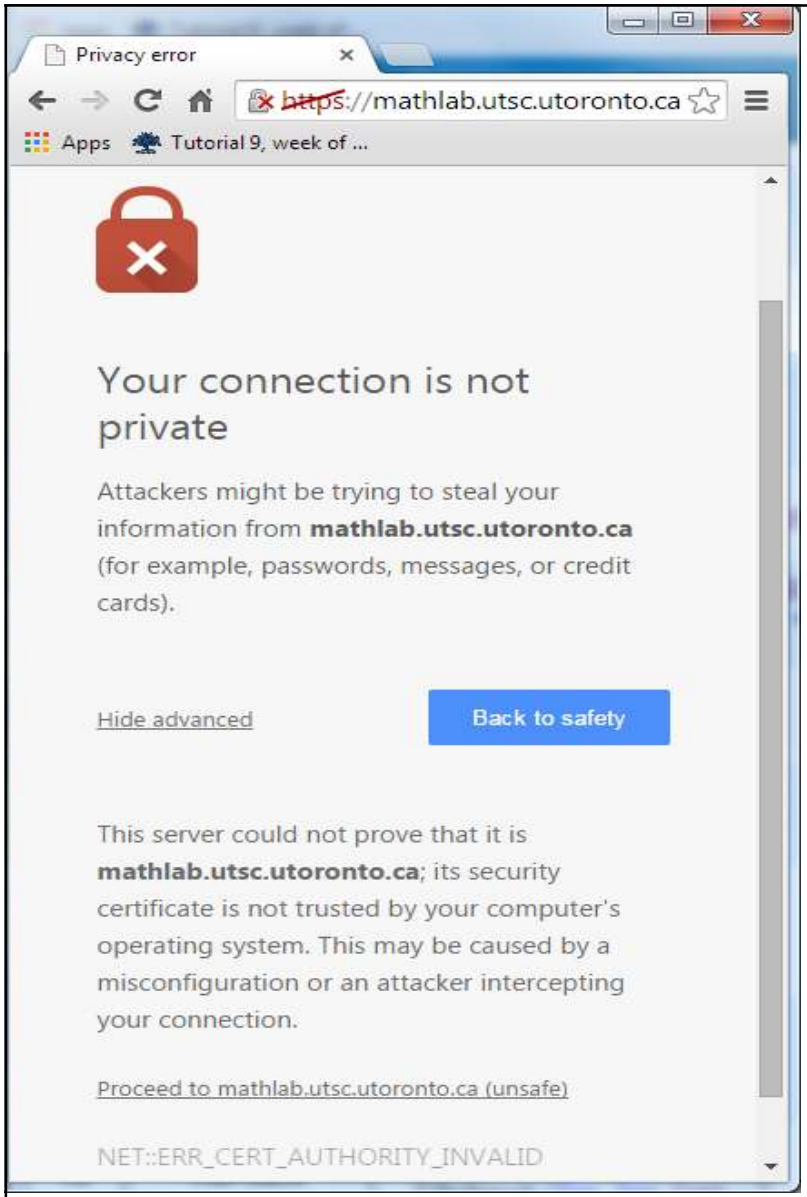




HTTPS



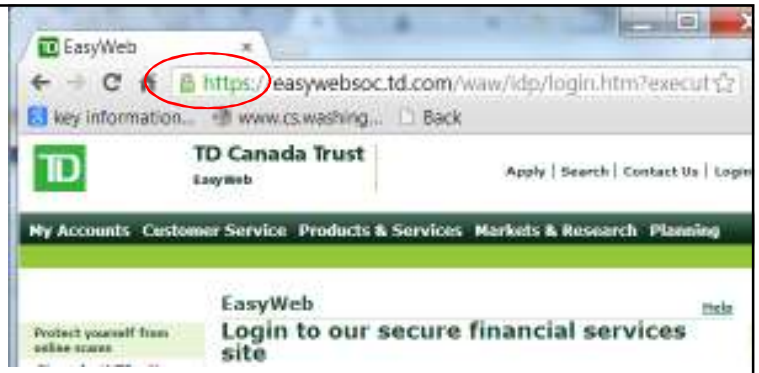
- Can Node.js serve HTTPS?
- Yes! Requires:
  - creation of a public-private key-pair for your server
  - authentication of this key-pair within a signed certificate
  - choosing a port number on which to serve the HTTPS protocol
    - must be separate port from HTTP port number in use
    - Node.js can serve both HTTP and HTTPS requests (in parallel), as can most Web servers
- More details provided in tutorial



HTTPS

- We use “self-signed” key-certificates, which makes Chrome nervous
- A real app would use a properly-signed certificate

HTTPS Technologies



- HTTPS is built on top of a protocol called SSL/TLS (secure sockets layer/transport layer security)
- When you connect to an HTTPS server:
  - The server sends its public key to the browser
  - Browser uses that public key to confidentially send a random number that is used as a symmetric session key seed
  - Browser and server independently compute same symmetric session key
  - Symmetric session key is used to encrypt all subsequent messages sent between client and server
  - Additionally, all HTTP messages between client and server are protected against tampering by a secure hash algorithm

HTTPS Mixed-mode Content

- Beware of loading "mixed-mode" app content, that is, loading the primary page (e.g. index.html) over https, but then loading non-https resources such as scripts
- Mixed content opens a serious security hole – attacker can replace http scripts with malicious code
- Relative URL's from https page are safe, but must change all explicit **`http://`** URLs to **`https://`**
- Some browsers try to protect you against mixed-mode: e
- **`[blocked] The page at 'https://mathlab.utsc.utoronto.ca:nnnnn/public/#dishes' was loaded over HTTPS, but ran insecure content from 'http://mathlab.utsc.utoronto.ca:nnnnn/public/js/utils.js': this content should also be loaded over HTTPS`**

HTTPS Authentication Protection

- Although parts of your app may have no apparent need for HTTPS encryption, e.g. browsing, HTTPS also provides authentication of the app-server which protects users against phishing sites, that look like your app in order to trick users into disclosing private information
- If user connections start out using http for browsing and then switch to https for authentication, your app is exposed to “SSL-stripping”, a technique for harvesting sensitive user data by preventing the connection from upgrading to https
- If all your app connections are https (no http support), you prevent SSL-stripping attacks

Mitigating CSRF

- How could we mitigate CSRF attacks?
- Problem is that browsers automatically send cookies with any request, regardless of where that request originated

- Solution 1: Add **`Referer`** HTTP header, that identifies the domain from which the request page originated
  - If you load a page from your node.js server, the Referer request header would be something like:

    `http://mathlab.utsc.utoronto.ca:port_#/public/`
  - Whereas if you make a request from an attack-page loaded from bad.com, the Referer header would be:

    `http://bad.com/EditView.html`

Mitigating CSRF

- How could we mitigate CSRF attacks?

- Solution 2: add unforgable/untamperable nonce token to all forms returned to the browser by your app server
  - CSRF attacker piggybacks on browser secure-cookie behavior (does not have to know cookie value)
  - Attacker usually does not have access to actual cookie value – does this provide extra capability?
  - How to prevent cookie value access?
  - 2 mechanisms: mark cookie as **`httponly`** and **`secure`**

Mitigating CSRF

- Solution 2: add unforgable/untamperable nonce token to all forms delivered by your app server
- To keep attacker from gaining access to cookie value:
  - 2 mechanisms: mark cookie as **`httponly`** and **`secure`**
  - **`httponly`** ensures that cookie can't be read by JavaScript; example XSS attack shown above would fail: **`<script>alert(document.cookie)</script>`**
  - **`secure`** ensures that cookies are sent only over secure https connections, so attacker can't read them using network sniffing tools

Mitigating CSRF

- How could we mitigate CSRF attacks?
- Solution 3: add unforgable/untamperable nonce-token in HTTP header on all requests to your app server
  - Server checks that nonce, generated e.g. by Express CSRF middleware, is present in HTTP request header
  - If nonce is not provided, request is rejected
  - Depends on secure HTTPS connection, otherwise attacker can eavesdrop to obtain secure nonce-token

Mitigating XSS

- How could we mitigate XSS attacks?
- In general this is quite hard, since there are many ways to hide the presence of code (not just a matter of blocking HTML tags)
- In general, better to take a whitelist approach wherein you list what is allowed, rather than a blacklist approach that lists what is banned
  - if your list is incomplete (likely), err on the side of caution

Mitigating XSS

- One simple step we can take is to use the **`_.escape()`** method provided by the client-side Underscore library
- **`_.escape()`** will transform an input such as **`<script>alert(1);</script>`** into **`<script>alert(1)</script>`** which will not be executed by the browser.
- Another sanitizing feature provided by the Underscore template system is escaping HTML in templates. Replacing **`<%= ... %>`** with **`<%- ... %>`** will cause any HTML tags in model values to be escaped as with **`_.escape()`**
- More comprehensive solutions could make use of both client and server-side input validation oriented toward XSS