Web Security I

Web Security Models
Browser Security
Web Technologies and Protocols
Web Security Model
Web Applications

What are the dangers?

Browser

Web Server

Network
Threat Models

The main vector of attack is via the content of a website.

Web Server

requests

Web attacker

Denial Of Service (DOS) attacks, or malware

network attacker

Browser

responses

malware, compromised client

2/15/24
Network Attacks

- **Standard Flow**
- **Block (DoS)**
- **Wiretapping (sniffing)**
- **Attacker in the Middle (passive)**
- **Attacker in the Middle (active)**
- **Creation (spoofing)**
Web Attacker Capabilities

• Attacker controls malicious website
  – Website might look professional, legitimate, etc.
  – Attacker can get users to visit website (how?)

• Good website is compromised by attacker
  – Attacker inserts malicious content into website
  – Attacker steals sensitive data from website
  – … Attacker does not have direct access to user's machine
Potential Damage

• An attacker gets you to visit a malicious website
  – Can they perform actions on other websites impersonating you?
  – Can they run evil code on your OS?
• Ideally, none of these exploits are possible ...
Attack Vectors

• **Web browser (focus of this lecture)**
  – Renders web content (HTML pages, scripts)
  – Responsible for confining web content
  – **Note:** Browser implementations dictate what websites can do

• **Web applications**
  – Server code (PHP, Ruby, Python, …)
  – Client-side code (JavaScript)
  – Many potential bugs (which you’ll explore in Project 2 😊)
Browser Security: Sandbox

• **Goal**: protect local computer from web attacker
  – Safely execute code on a website
  – ... without the code accessing your files, tampering with your network, accessing other sites

• High stakes ($40K bounty for Google Chrome; www.google.com/about/appsecurity/chrome-rewards/)

• We won't address attacks that break the sandbox

• But they **happen** check the **CVE list**
  – https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=sandbox
Domains, HTML and HTTP
URL and FQDN

- **URL Uniform Resource Locator**
  - https://cs.brown.edu/about/contacts.html
  - a protocol (e.g. https), a FQDN (e.g. cs.brown.edu)
  - a path and file name (e.g. /about/contacts.html)

- **FQDN (Fully Qualified Domain Name)**
  - [Host name].[Domain].[TLD].[Root]
  - Two or more labels, separated by dots (e.g., cs.brown.edu)

- **Root name server**
  - It is a “.” at the end of the FQDN

- **Top-level domain (TLD)**
  - Generic (gTLD), .com, .org, .net, ...
  - Country-code (ccTLD), .ca, .it, ...

2/15/24
HTML

• Hypertext markup language (HTML)
  – Allows linking to other pages (href)
  – Supports embedding of images, scripts, other pages (script, iframe)
  – User input accepted in forms

```html
<html>
  <head>
    <title>Google</title>
  </head>
  <body>
    <p>Welcome to my page.</p>
    <script>alert("Hello world");</script>
    <iframe src="http://example.com">
    </iframe>
  </body>
</html>
```
HTTP (Hypertext Transport Protocol)

• Communication protocol between client and server

```
GET /search?q=cs166&num=02 HTTP/1.1
Host: www.google.com
```

```
HTTP/1.1 200 OK
Server: Apache/2.2.3 (CentOS) ... 
Content-Type: text/html
<html>
 <head>
   <title>Google</title>
 </head>
 <body>...</body>
</html>
```
What’s in a request (or response)?

**REQUEST**

- URL (domain, path)

```
GET /search?q=cs166&num=02 HTTP/1.1
Host: www.google.com
```

**RESPONSE**

- Metadata
  - HTTP/1.1 200 OK
  - Server: Apache/2.2.3 (CentOS) ...
  - Content-Type: text/html

```
<html>
<head>
  <title>Google</title>
</head>
<body>...
</body>
</html>
```

- Resource
Variables

- Key-value pairs obtained from user input into forms and submitted to server
- Submit variables in HTTP via GET or PUT
- GET request: variables within HTTP URL, e.g.,
  
  ```
  http://www.google.com/search?q=cs166&num=02
  ```

- POST request: variables within HTTP body, e.g.,

  ```
  POST / HTTP/1.1
  Host: example.com
  Content-Type: application/x-www-form-urlencoded
  Content-Length: 18

  month=5&year=2024
  ```
Semantics: GET vs. POST

**GET**
- Request target resource
- Read-only method
- Submitted variables may specify target resource and/or its format

**POST**
- Request processing of target resource
- Read/write/create method
- Submitted variables may specify how resource is processed (e.g., content of resource to be created, updated, or executed)
## GET vs. POST

<table>
<thead>
<tr>
<th></th>
<th>GET</th>
<th>POST</th>
</tr>
</thead>
<tbody>
<tr>
<td>Browser history</td>
<td>✓</td>
<td>X</td>
</tr>
<tr>
<td>Browser bookmarking</td>
<td>✓</td>
<td>X</td>
</tr>
<tr>
<td>Browser caching</td>
<td>✓</td>
<td>X</td>
</tr>
<tr>
<td>Server logs</td>
<td>✓</td>
<td>X</td>
</tr>
<tr>
<td>Reloading page</td>
<td>immediate</td>
<td>warning</td>
</tr>
<tr>
<td>Variable values</td>
<td>Restricted</td>
<td>arbitrary</td>
</tr>
</tbody>
</table>
Moving from Browser Security to Web Application Security: Client-Side Controls
Client-Side Controls

- Web security problems arise because clients can submit arbitrary input.

- What about using client-side controls to check the input?

- Which kind of controls?
Client-Side Controls

- A standard application may rely on client-side controls to restrict user input in two general ways:
  - Transmitting data via the client component using a mechanism that should prevent the user from modifying that data
  - Implementing measures on the client side
- In this model the Server does not trust the Client
Bypassing Web Client-Side Controls

• In general a security flaw because it is easy to bypass
• The user:
  – has a full control over the client and the data it submits
  – Can bypass any controls that are client-side and not replicated on the server
• Why these controls are still useful?
  – E.g. for load balancing or usability
  – Often we can suppose that the vast majority of users are honest
Transmitting Data Via the Client

• A common developer **bad habit** is passing data to the client in a **form** that the end user cannot **directly** see or modify

• Why is it so common?
  – It removes or reduces the amount of data to store server side per-session
  – In a multi-server application it removes the need to synchronize the session data among different servers
  – The use of third-party components on the server may be difficult or impossible to integrate

• Transmitting data via the client is often the **easy solution** but unfortunately is **not secure**.
Common Mechanisms

• HTML Hidden fields
  – A field flagged hidden is not displayed on-screen

• HTTP Cookies
  – Not displayed on-screen, and the user cannot modify directly

• Referer Header
  – An optional field in the http request that it indicates the URL of the page from which the current request originated

• If you use the proper tool you can tamper the data on the client-side
Web client tool

• Web inspection tool:
  – Firefox or Chrome web developer:
    • powerful tools that allow you to edit HTML, CSS and view the coding behind any website: CSS, HTML, DOM and JavaScript

• Web Proxy:
  – Burp, OWASP ZAP, etc.
    • Allow to modify GET or POST requests
HTTP Proxy

• An intercepting Proxy:
  – *inspect* and *modify* traffic between your browser and the target application
  – Burp Intruder, OWASP ZAP, etc.
Demos

• Owasp Webgoat
  https://github.com/WebGoat/WebGoat
    – parameter injection
    – Bypass html field restrictions
    – Exploit hidden fields
    – Bypass client side java script validation
BREAK!
Web Intro
Most of our trust on web security relies on information stored in the Browser:

- A Browser should be updated since Bugs in the browser implementation can lead to various attacks
  

- Add-ons too are dangerous
  - Hacking Team flash exploits - goo.gl/syVwiD
  - github.com/greatsuspending/thegreatsuspending/issues/1263

- Executing a browser with low privileges helps
Browser Security: Same-Origin Policy

• Very simple idea: “Content from different origins should be isolated”
  – Website origin defined over tuple (protocol, domain, port)

• Very difficult to execute in practice...
  – Messy number of cases to worry about...
    • HTML elements?
    • Navigating Links?
    • Browser cookies?
    • JavaScript capabilities?
    • iframes?
    • etc.
  – Browsers didn’t always get this correct...
Browser Security: SOP

• **Goal**: Protect and isolate web content from other web content
  – Content from different origins should be isolated, e.g., mal.com should not interact with bank.com in unexpected ways
  – What about `cs.brown.edu` vs `brown.edu` or `mail.google.com` vs `drive.google.com`?
  – Lots of subtleties
SOP Example:  
http://store.company.com/dir/page.html

<table>
<thead>
<tr>
<th>URL</th>
<th>Outcome</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://store.company.com/dir2/other.html">http://store.company.com/dir2/other.html</a></td>
<td>Same origin</td>
<td>Only the path differs</td>
</tr>
<tr>
<td><a href="http://store.company.com/dir/inner/another.html">http://store.company.com/dir/inner/another.html</a></td>
<td>Same origin</td>
<td>Only the path differs</td>
</tr>
<tr>
<td><a href="https://store.company.com/page.html">https://store.company.com/page.html</a></td>
<td>Failure</td>
<td>Different protocol</td>
</tr>
<tr>
<td><a href="http://store.company.com:81/dir/page.html">http://store.company.com:81/dir/page.html</a></td>
<td>Failure</td>
<td>Different port (http:// is port 80 by default)</td>
</tr>
<tr>
<td><a href="http://news.company.com/dir/page.html">http://news.company.com/dir/page.html</a></td>
<td>Failure</td>
<td>Different host</td>
</tr>
</tbody>
</table>

Cookies
Cookies

- HTTP is a stateless protocol; cookies used to emulate state
- Servers can store cookies (name-value pairs) into browser
  - Used for user preferences (e.g., language and page layout), user tracking, authentication
  - Expiration date can be set
  - May contain sensitive information (e.g., for user authentication)
- Browser sends back cookies to server on the next connection

```plaintext
POST /login.php HTTP/1.1
Set-Cookie: Name: sessionid
            Value: 19daj3kdop8gx
            Domain: cs.brown.edu
            Expires: Wed, 21 Feb 2024 ...
```
Cookie Scope

• Each cookie has a scope
  – Base domain, which is a given host (e.g., brown.edu)
  – Plus, optionally, all its subdomains (cs.brown.edu, math.brown.edu, www.cs.brown.edu, etc.)

• For ease of notation, we denote with . the inclusion of subdomains (e.g., .brown.edu)
  – This isn’t the real notation—it’s actually specified in HTTP with the "Domain:" attribute of a cookie
Same Origin Policy: Cookie Reads

Websites can only read cookies within their scope

- Example: browser has cookies with scope brown.edu .brown.edu, .math.brown.edu cs.brown.edu .cs.brown.edu, blog.cs.brown.edu
- Browser accesses cs.brown.edu
- Browser sends cookies with scope .brown.edu cs.brown.edu .cs.brown.edu
Same Origin Policy: Cookie Writes

A website can set cookies for (1) its base domain; or (2) a super domain (except TLDs) and its subdomains

- Browser accesses `cs.brown.edu`
- `cs.brown.edu` can set cookies for `brown.edu`
- `cs.brown.edu`

- But not for `google.com`
- `com`
- `math.brown.edu`
- `brown.edu`
- `...`
Clicker Question #1

If the browser accesses cs.brown.edu, the server can set cookies with which of the following scopes?

A. .brown.edu
B. only math.brown.edu
C. only help.cs.brown.edu
D. All of the above
E. None of the above
If the browser accesses cs.brown.edu, the server can set cookies with which of the following scopes?

A. .brown.edu
B. only math.brown.edu
C. only help.cs.brown.edu

... The scope is cs.brown.edu by default
The server can optionally set cookies with scope .cs.brown.edu and .brown.edu, but nothing else
Session Management

• **Session**
  – Keep track of client over a series of requests
  – Server assigns clients a unique, unguessable ID
  – Clients send back ID to verify themselves

• **Session**
  – Necessary in sites with authentication (e.g., banking)
  – Useful in most other sites (e.g., remembering preferences)

• Various methods to implement them (mainly cookies), but also could be in HTTP variables
Session Management: goal

- **Goal**
  - Users should not have to authenticate for every single request
- **Problem**
  - HTTP is stateless
- **Solution**
  - User logs in once
  - Server generate session ID and gives it to browser
    - Temporary token that identifies and authenticates user
  - Browser returns session ID to server in subsequent requests
Specifications for a Session ID

• Created by server upon successful user authentication
  — Generated as long random string
  — Associated with scope (set of domains) and expiration
  — Sent to browser

• Kept as secret shared by browser and server

• Transmitted by browser at each subsequent request to server
  — Must use secure channel between browser and server

• Session ID becomes invalid after expiration
  — User asked to authenticate again
Implementation of Session ID

- **Cookie**
  - Transmitted in HTTP headers
  - `Set-Cookie: SID=c5Wuk7...`
  - `Cookie: SID=c5Wuk7...

- **GET variable**
  - Added to URLs in links
  - `https://www.example.com?SID=c5Wuk7...

- **POST variable**
  - Navigation via POST requests with hidden variable
  - `<input type="hidden" name="SID" value="c5Wuk7...">`
Session ID in Cookie

POST /login HTTP/1.1
Host: www.example.com
Username: cs166ta
Password: Ilove166

HTTP/1.1 200 OK
Set-Cookie: SID=c5Wuk7...

GET /profile.html HTTP/1.1
Host: www.example.com
Cookie: SID=c5Wuk7...
Session ID in Cookie

• Advantages
  – Cookies automatically returned by browser
  – Cookie attributes provide support for expiration, restriction to secure transmission (HTTPS), and blocking JavaScript access (httpOnly)

• Disadvantages
  – Cookies are shared among all browser tabs
    • (not other browsers or incognito)
  – Cookies are returned by browser even when request to server is made from element (e.g., image or form) within page from other server
  – This may cause browser to send cookies in context not intended by user
Session ID in GET Variable

Browser

POST /login HTTP/1.1
Host: www.example.com
Username: cs166ta
Password: Ilove166

Server

HTTP/1.1 200 OK
<html>
...
<a href="/profile.html?SID=c5Wuk7...">
...

GET /profile.html?SID=c5Wuk7... HTTP/1.1
Host: www.example.com
Session ID in GET Variable

• Advantages
  – Session ID transmitted to server only when intended by user

• Disadvantages
  – Session ID inadvertently transmitted when user shares URL
  – Session ID transmitted to third-party site within referrer
  – Session ID exposed by bookmarking and logging
  – Server needs to dynamically generate pages to customize site navigation links and POST actions for each user
  – Transmission of session ID needs to be restricted to HTTPS on every link and POST action
Session ID in POST Variable

```
POST /login HTTP/1.1
Host: www.example.com
Username: cs166ta
Password: Ilove166

HTTP/1.1 200 OK
...
<form method="POST"
action=".../profile"
name="SID"
value="c5Wuk7...

POST /profile HTTP/1.1
Host: www.example.com
SID=c5Wuk7...
```
Session ID in POST Variable

• Advantages
  — Session ID transmitted to server only when intended by user
  — Session ID not present in URL, hence not logged, bookmarked, or transmitted within referrer

• Disadvantages
  — Navigation must be made via POST requests
  — Server needs to dynamically generate pages to customize forms for each user
  — Transmission of session ID needs to be restricted to HTTPS on every link and POST action
Clicker Question 2

In the cookie implementation of session tokens, how is the token transmitted to/from the server?

A. Included as a parameter in the URL
B. As a hidden variable in the initial POST request
C. As an additional field when the user authenticates
D. In the HTTP header (both request and response)
In the cookie implementation of session tokens, how is the token transmitted to/from the server?

A. Included as a parameter in the URL
B. As a hidden variable in the initial POST request
C. As an additional field when the user authenticates
D. In the HTTP header (both request and response)
1. Remove cookies or tampering parameters, and it erases authentication
   — Server makes us log in again
2. Close session you do not remove server cookie
3. Logout and session cookie removed on server
4. Cookie stealing for authentication
5. Remember me checkbox on the login
   — Cookie does not expire in the browser but also on the server
6. If we disable cookies, cannot sign in to most websites
7. Burp analysis for the entropy of session cookies

Note: In particular for last demos, Browsers can have different policies
OWASP Top Ten (2013-17)

A1: Injection
A2: Broken Authentication and Session Management
A3: Cross-Site Scripting (XSS)
A4: Broken Access Control
A5: Security Misconfiguration
A6: Sensitive Data Exposure
A7: Insufficient Attack Protection
A8: Cross Site Request Forgery (CSRF)
A9: Using Components with Known Vulnerabilities
A10: Unprotected API
Owasp 2017 - 2021

www.owasp.org/index.php/Top_10
What We Have Learned

• Web Security Models
• Same-Origin Policy
• Basics of HTTP protocol
• GET and POST methods for HTTP variables
• Client-Side Controls
• Cookies and session management
• Same origin policy (SOP)