CS 253: Web Security
Cookie and Session Attacks
Assignment 0 is due Friday, October 4 at 5:00pm
Set-Cookie: theme=dark;
Cookie: theme=dark;
Sessions

- **Cookies** are used by the server to implement **sessions**
- **Goal:** Server keeps a set of data related to a user's current "browsing session"
- Examples
  - Logins
  - Shopping carts
  - User tracking
First HTTP request:

POST /login HTTP/1.1
Host: example.com

username=alice&password=password

HTTP response:

HTTP/1.1 200 OK
Set-Cookie: username=alice
Date: Tue, 24 Sep 2019 20:30:00 GMT

<!DOCTYPE html ...

All future HTTP requests:

GET /page.html HTTP/1.1
Host: example.com
Cookie: username=alice;
Ambient authority

- **Access control** - Regulate who can view resources or take actions

- **Ambient authority** - Access control based on a *global and persistent property* of the requester
  - The alternative is explicit authorization *valid only for a specific action*

- There are four types of ambient authority on the web
  - **Cookies** - most common, most versatile method
  - **IP checking** - used at Stanford for library resources
  - **Built-in HTTP authentication** - rarely used
  - **Client certificates** - rarely used
Demo: Sessions
<!doctype html>
<html lang='en'>
<head>
  <meta charset='utf-8' />
  <title>Example Bank</title>
</head>
<body>
<form method='POST' action='/login'>
  Username:
  <input name='username' />
  Password:
  <input name='password' type='password' />
  <input type='submit' value='Login' />
</form>
</body>
</html>
Demo: Insecure Session 1

```javascript
const express = require('express')
const cookieParser = require('cookie-parser')
const { createReadStream } = require('fs')
const bodyParser = require('body-parser')

const app = express()
app.use(cookieParser())
app.use(bodyParser.urlencoded({ extended: false }))

// Routes go here!

app.listen(4000)
```
const USERS = { alice: 'password', bob: 'hunter2' }
const BALANCES = { alice: 500, bob: 100 }

app.get('/', (req, res) => {
  const username = req.cookies.username
  if (username) {
    res.send(`Hi ${username}. Your balance is ${BALANCES[username]}.`)
  } else {
    createReadStream('index.html').pipe(res)
  }
})

app.post('/login', (req, res) => {
  const username = req.body.username
  const password = USERS[username]
  if (password === req.body.password) {
    res.cookie('username', username)
    res.redirect('/')
  } else {
    res.send('fail!')
  }
})
Quick primer: Signature schemes

- Triple of algorithms \((G, S, V)\)
  - \(G() \rightarrow (pk, sk)\) - generator returns public key and secret key
  - \(S(sk, x) \rightarrow t\) - signing returns a tag \(t\) for input \(x\)
  - \(V(pk, x, t) \rightarrow \text{accept}|\text{reject}\) - checks validity of tag \(t\) for given input \(x\)
- Correctness property
  - \(V(pk, x, S(sk, x)) = \text{accept}\) should always be true
- Security property
  - \(V(pk, x, t) = \text{accept}\) should almost never be true when \(x\) and \(t\) are chosen by the attacker
Client

Server

\[ G() \rightarrow (pk, sk) \]
POST /login HTTP/1.1
username=alice&password=password

G() \rightarrow (pk, sk)
POST /login HTTP/1.1
username=alice\&password=password

G() \rightarrow (pk, sk)

Login info ok?
POST /login HTTP/1.1
username=alice&password=password

G() → (pk, sk)

Login info ok?

OK!
POST /login HTTP/1.1
username=alice&password=password

G() → (pk, sk)
Login info ok? OK!
S(sk, 'alice') → t
POST /login HTTP/1.1
username=alice&password=password

HTTP/1.1 200 OK
Set-Cookie: username=alice;
Set-Cookie: tag=t;
POST /login HTTP/1.1
username=alice&password=password

HTTP/1.1 200 OK
Set-Cookie: username=alice;
Set-Cookie: tag=t;

GET / HTTP/1.1
Cookie: username=alice; tag=t

G() → (pk, sk)
Login info ok? OK!
S(sk, 'alice') → t
POST /login HTTP/1.1
username=alice&password=password

HTTP/1.1 200 OK
Set-Cookie: username=alice;
Set-Cookie: tag=t;

GET / HTTP/1.1
Cookie: username=alice; tag=t

G() → (pk, sk)
Login info ok? OK!
S(sk, 'alice') → t
V(pk, 'alice', t) → ok?
POST /login HTTP/1.1
username=alice&password=password

HTTP/1.1 200 OK
Set-Cookie: username=alice;
Set-Cookie: tag=t;

GET / HTTP/1.1
Cookie: username=alice; tag=t

G() \rightarrow (pk, sk)

Login info ok? \rightarrow OK!

S(sk, 'alice') \rightarrow t

V(pk, 'alice', t) \rightarrow ok? \rightarrow OK!
POST /login HTTP/1.1
username=alice&password=password

HTTP/1.1 200 OK
Set-Cookie: username=alice;
Set-Cookie: tag=t;

GET / HTTP/1.1
Cookie: username=alice; tag=t

HTTP/1.1 200 OK
Private webpage for Alice!

G() \rightarrow (pk, sk)

Login info ok? \rightarrow OK!

S(sk, 'alice') \rightarrow t

V(pk, 'alice', t) \rightarrow ok? \rightarrow OK!
POST /login HTTP/1.1
username=alice&password=password

HTTP/1.1 200 OK
Set-Cookie: username=alice;
Set-Cookie: tag=t;

GET / HTTP/1.1
Cookie: username=alice; tag=t

HTTP/1.1 200 OK
Private webpage for Alice!
Demo: Insecure Session 2

```javascript
app.use(cookieParser(COOKIE_SECRET))

app.get('/', (req, res) => {
  const username = req.signedCookies.username
  if (username) {
    res.send(`Hi ${username}. Your balance is ${BALANCES[username]}.`)
  } else {
    createReadStream('index.html').pipe(res)
  }
})

app.post('/login', (req, res) => {
  const username = req.body.username
  const password = USERS[username]
  if (password === req.body.password) {
    res.cookie('username', username, { signed: true })
    res.redirect('/')
  } else {
    res.send('fail!')
  }
})

app.get('/logout', (req, res) => {
  res.clearCookie('username')
  res.redirect('/')
})
```
Demo: Insecure Session 3

```javascript
let nextSessionId = 1
const SESSIONS = {} // sessionId -> username

app.get('/', (req, res) => {
  const sessionId = req.cookies.sessionId
  const username = SESSIONS[sessionId]
  if (username) {
    res.send(`Hi ${username}. Your balance is $${BALANCES[username]}.`)
  } else {
    createReadStream('index.html').pipe(res)
  }
})

app.post('/login', (req, res) => {
  const username = req.body.username
  const password = USERS[username]
  if (password === req.body.password) {
    SESSIONS[nextSessionId] = username
    res.cookie('sessionId', nextSessionId)
    nextSessionId += 1
    res.redirect('/')
  } else {
    res.send('fail!')
  }
})

app.get('/logout', (req, res) => {
  const sessionId = req.cookies.sessionId
  delete SESSIONS[sessionId]
  res.clearCookie('sessionId')
  res.redirect('/')
})
```
const { randomBytes } = require('crypto')

const SESSIONS = {} // sessionId -> username

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

  if (username) {
    res.send(`Hi ${username}. Your balance is ${BALANCES[username]}.`)
  } else {
    createReadStream('index.html').pipe(res)
  }
})

app.post('/login', (req, res) => {
  const username = req.body.username
  const password = USERS[username]
  if (password === req.body.password) {
    const sessionId = randomBytes(16).toString('hex')
    SESSIONS[sessionId] = username
    res.cookie('sessionId', sessionId)
    res.redirect('/')
  } else {
    res.send('fail!')
  }
})

app.get('/logout', (req, res) => {
  const sessionId = req.cookies.sessionId
  delete SESSIONS[sessionId]
  res.clearCookie('sessionId')
  res.redirect('/')
})
History of cookies

- Implemented in 1994 in Netscape and described in 4-page draft
- No spec for 17 years
  - Attempt made in 1997, but made incompatible changes
  - Another attempt in 2000 ("Cookie2"), same problem
  - Around 2011, another effort succeeded (RFC 6265)
- Ad-hoc design has led to *interesting* issues
Cookie attributes

- **Expires** - Specifies expiration date. If no date, then lasts for session
- **Path** - Scope the "Cookie" header to a particular request path prefix
  - e.g. `Path=/docs` will match `/docs` and `/docs/Web/
  - Do not use for security
- **Domain** - Allows the cookie to be scoped to a domain broader than the domain that returned the Set-Cookie header
  - e.g. `login.stanford.edu` could set a cookie for `stanford.edu`
Set-Cookie: theme=dark; Expires=<date>
How long can cookies last?

- Sites can set `Expires` to a very far-future date and the cookie will last until the user clears it.
  - 2007: "The Google Blog announced that Google will be shortening the expiration date of its cookies from the year 2038 to a two-year life cycle." – Search Engine Land

- When `Expires` not specified, lasts for current browser session
  - Caveat: Browsers do session restoring, so can last way longer
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
```
Accessing Cookies from JS

document.cookie = 'name=Feross'
document.cookie = 'favoriteFood=Cookies; Path=/'

document.cookie
// name=Feross; favoriteFood=Cookies;

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

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
GET /HTTP/1.1
Cookie: sessionId=1234
GET /HTTP/1.1
Cookie: sessionId=1234

HTTP/1.1 200 OK
Private webpage!
GET /HTTP/1.1
Cookie: sessionId=1234
GET /HTTP/1.1
Cookie: sessionId=1234
GET /HTTP/1.1
Cookie: sessionId=1234

GET /HTTP/1.1
Cookie: sessionId=1234

HTTP/1.1 200 OK
Private webpage!
Firesheep (2010)
Session hijacking mitigation

- Use **Secure** cookie attribute to prevent cookie from being sent over unencrypted HTTP connections

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

- Even better: Use HTTPS for entire website
Session hijacking via Cross Site Scripting (XSS)

- What if website is vulnerable to XSS?
  - Attacker can insert their code into the webpage
  - At this point, they can easily exfiltrate the user's cookie

```javascript
```

- More on XSS next week!
Protect cookies from XSS

- Use `HttpOnly` cookie attribute to prevent cookie from being read from JavaScript

```
Set-Cookie: key=value; Secure; HttpOnly
```
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 `<iframe>` with the path of the cookie
  - Then, read `iframe.contentDocument.cookie`
- This is allowed by Same Origin Policy (more on this next time!)
- Therefore, only use **Path** as a performance optimization
Demo: CS 106A attack
Demo: CS 106A attack

On CS 106A site:

```javascript
document.cookie = 'sessionId=1234; Path=/class/cs106a/
```

On CS 253 site:

```javascript
const iframe = document.createElement('iframe')
iframe.src = 'https://web.stanford.edu/class/cs106a/
document.body.appendChild(iframe)
console.log(iframe.contentDocument.cookie)
```
Make cookie Path secure?

- No solution! Always unsafe to rely on Path
- Cookies can only be accessed by equal or more-specific domains, so use a subdomain

- `cs106a.stanford.edu` vs. `cs253.stanford.edu`
  - Mutually exclusive

- `cs253.stanford.edu` vs. `stanford.edu`
  - Former can access latter’s cookies. Reverse not true.

- `acm.stanford.edu` vs. `login.stanford.edu`
  - Mutually exclusive

- `hello.login.stanford.edu` vs. `login.stanford.edu`
  - Former can access latter’s cookies. Reverse not true.
What to set cookie Path to?

- Just set it to **Path=*/ and don't ever rely on it

**Set-Cookie**: key=value; Secure; HttpOnly; Path=/

- Why is this better than just omitting **Path**?
Problem with ambient authority

- Unclear which site initiated a request
- Consider this HTML embedded in attacker.com:

```html
<img src='https://bank.example.com/withdraw?from=bob&to=mallory&amount=1000'>
```

- Browser helpfully includes bank.example.com cookies in all requests to bank.example.com, even though the request originated from attacker.com
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
Demo: Cross-Site Request Forgery
Demo: Cross-Site Request Forgery

server.js:

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

app.get('/', (req, res) => {
    const sessionId = req.cookies.sessionId
    const username = SESSIONS[sessionId]
    if (username) {
        res.send(`Hi ${username}. Your balance is $${BALANCES[username]}.<form method='POST' action='http://localhost:4000/transfer'>
        Send amount: <input name='amount' />
        To user: <input name='to' />
        <input type='submit' value='Send' />
    </form>`)
    } else {
        createReadStream('index.html').pipe(res)
    }
})
```

```javascript
app.post('/transfer', (req, res) => {
    const sessionId = req.cookies.sessionId
    const username = SESSIONS[sessionId]
    if (!username) {
        res.send('fail!')
        return
    }
    const amount = Number(req.body.amount)
    const to = req.body.to
    BALANCES[username] -= amount
    BALANCES[to] += amount
    res.redirect('/')
})
```
Demo: Cross-Site Request Forgery

attacker.html:

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

attacker-frame.html:

```html
<form method='POST' action='http://localhost:4000/transfer'>
  <input name='amount' value='100' />
  <input name='to' value='alice' />
  <input type='submit' value='Send' />
</form>
<script>
  document.forms[0].submit()
</script>
```
Mitigate Cross-Site Request Forgery

- Idea: Can we remove "ambient authority" when a request originates from another site?
**SameSite cookies**

- 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
Proposal to make cookies `SameSite=Lax` by default

- "First, cookies should be treated as "SameSite=Lax" by default. Second, cookies that explicitly assert "SameSite=None" in order to enable cross-site delivery should also be marked as "Secure"."

- Who would want to opt into `SameSite=None` cookies?

How long should cookies last?

- Use a reasonable expiration date for your cookies
  - 30-90 days
  - You can set the cookie with each response to restart the 30 day counter, so an active user won't ever be logged out, despite the short timeout

Set-Cookie: key=value; Secure; HttpOnly; Path=/;
SameSite=Lax; Expires=Fri, 1 Nov 2019 00:00:00 GMT
res.cookie('sessionId', sessionId, {
    secure: true,
    httpOnly: true,
    sameSite: 'lax',
    maxAge: 30 * 24 * 60 * 60 * 1000 // 30 days
})

res.clearCookie('sessionId', {
    secure: true,
    httpOnly: true,
    sameSite: 'lax'
})
Final Thoughts

- Cookies are used to implement sessions
- Never trust data from the client!
- Ambient authority is useful but opens us up to additional risks
- If you remember one thing: set your cookies like this:

```
Set-Cookie: key=value; Secure; HttpOnly; Path=/;
SameSite=Lax; Expires=Fri, 1 Nov 2019 00:00:00 GMT
```