# Web Development

COMP 431 / COMP 531

## Authorization

Scott E Pollack, PhD

March 29, 2016

# Part IIb – Back End Development

No Class
Thursday 3/31

- COMP 531 Frontend Review
  - Due Tuesday 4/5
- Homework Assignment 7 (Integrated Web App)
  - Due Tuesday 4/12
- COMP 531 Paper and Presentations 4/21
  - Due Thursday 4/21 before class
- Homework Assignment 8 (Final Full Web App)
  - Due Thursday 4/28

**Part IIb**

Authorization
Security
OAuth/2
Scalability
Service APIs
Integrating

# COMP 531 Paper and Presentation

- Topic

  Web Development or Design
  - New technology
  - Technology comparison
  - Site design analysis
  - Enterprise in the Web
  - E-commerce
  - User experience
  - User interfaces
  - ARIA, ReflectJS, Security, BigData

- Paper
  - 1000 to 2000 words
  - Proof read
  - Review and revise
  - Spelling and grammar
  - Think of it as a *blog post* that your future boss will read

- Presentation
  - No more than 5 minute talk
  - slides, web sites, demos, props, etc...

post your idea on Piazza – there will be no duplicate topics

# Cookies

in plain sight!

```
POST /login
{ username and password }
```

Server returns a magic cookie

Browser "eats" the cookie and returns it with all subsequent requests

```
PUT /logout
```

Server returns "emptied" cookie for browser to eat

# What's it look like in Node?

```
npm install cookie-parser -save
var cookieParser = require('cookie-parser')
app.use(cookieParser())
```

```
exports.setup = function(app) {
    app.post('/login', login)
    app.put('/logout', isLoggedIn, logout)
}

var cookieKey = 'sid'

function isLoggedIn(req, res, next) {
    var sid = req.cookies[cookieKey]

    if (!sid) {
        return res.sendStatus(401)
    }

    var username = sessionUser[sid]
    if (username) {
        req.username = username
        next()
    } else {
        res.sendStatus(401)
    }
```

middleware!

Unauthorized

```
function login(req, res) {
    var username = req.body.username;
    var password = req.body.password;
    if (!username || !password) {
        res.sendStatus(400)
        return
    }
    var userObj = getUser(username)
    if (!userObj || userObj.password !== password) {
        res.sendStatus(401)
        return
    }

    // cookie lasts for 1 hour
    res.cookie(cookieKey, generateCode(userObj),
        {maxAge: 3600*1000, httpOnly: true })

    var msg = { username: username, result: 'success
    res.send(msg)
}
```

Bad Request

**Gasp!**

Unauthorized

# In plain sight?

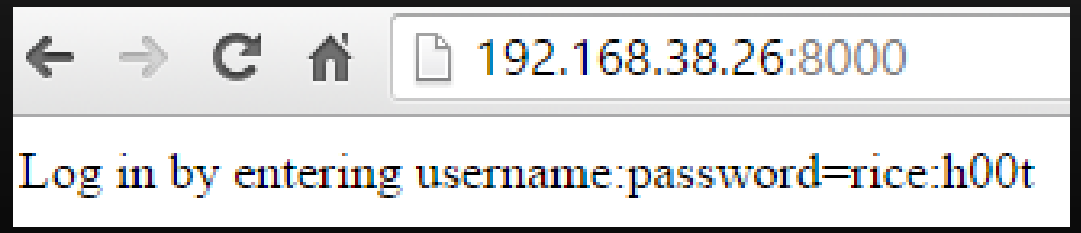
| No. | Time | Source | Destination | Protocol | Length | Info |
|---|---|---|---|---|---|---|
| 1 | 0.000000000 | 127.0.0.1 | 127.0.0.1 | TCP | 74 | 55009→5555 [SYN] Seq=0 Win=43690 Len=0 M |
| 2 | 0.000014000 | 127.0.0.1 | 127.0.0.1 | TCP | 74 | 5555→55009 [SYN, ACK] Seq=0 Ack=1 Win=43 |
| 3 | 0.000028000 | 127.0.0.1 | 127.0.0.1 | TCP | 66 | 55009→5555 [ACK] Seq=1 Ack=1 Win=43776 L |
| 4 | 0.000081000 | 127.0.0.1 | 127.0.0.1 | HTTP | 257 | POST /login HTTP/1.1 (application/json) |
| 5 | 0.000087000 | 127.0.0.1 | 127.0.0.1 | TCP | 66 | 5555→55009 [ACK] Seq=1 Ack=192 Win=44800 |
| 6 | 0.003395000 | 127.0.0.1 | 127.0.0.1 | HTTP | 791 | HTTP/1.1 200 OK (application/json) |

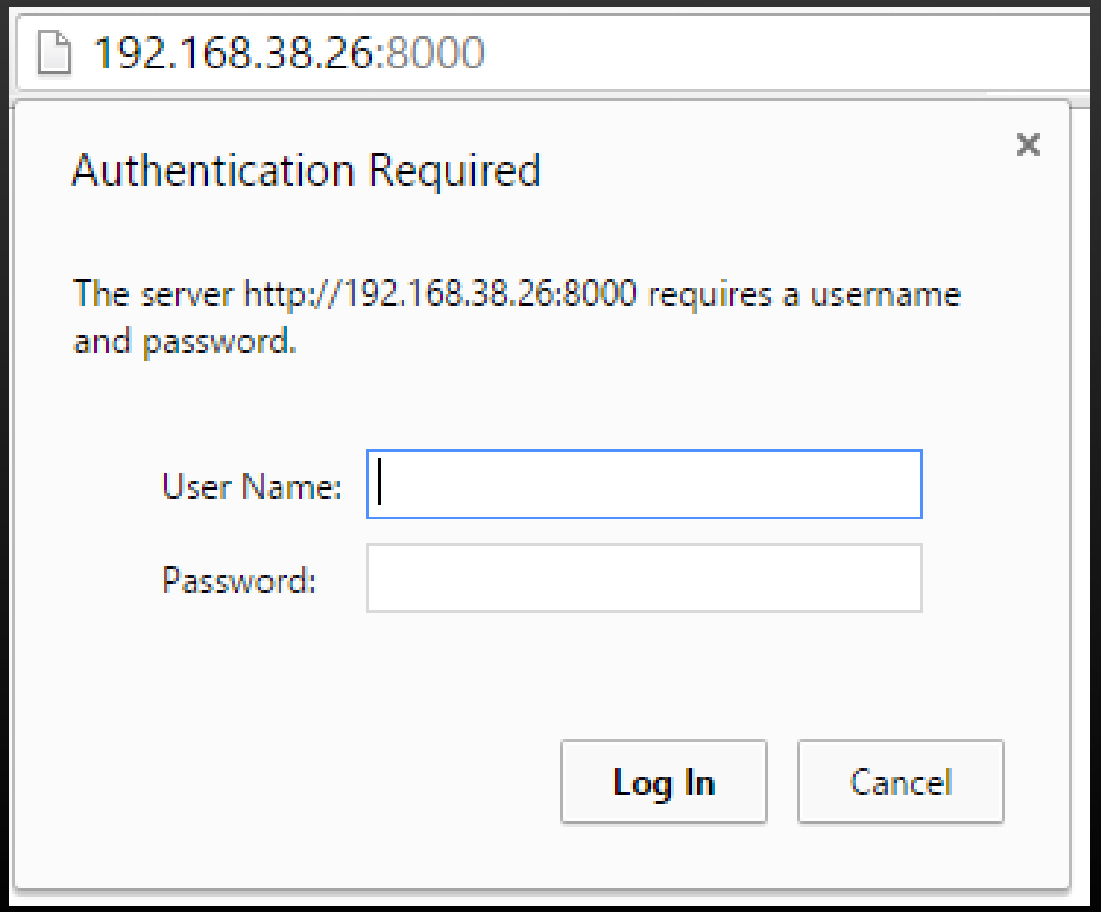
```
▸ Content-Length: 55\r\n
  \r\n
  [Full request URI: http://localhost:5555/login]
  [HTTP request 1/1]
  [Response in frame: 6]
▾ JavaScript Object Notation: application/json
  ▾ Object
    ▾ Member Key: "username"
      String value: sepltest
    ▾ Member Key: "password"
      String value: native-web-tester
```

Luckily **NO**!
We're using *HTTPS* so we have *TLS (SSL)* encrypting our transfers

# HTTP AUTH

- User makes request without Authorization
- Server responds 401 and sets WWW-Authenticate with a "challenge"
- User attempts challenge by filling in username and password
- Server then accepts or rechallenges



```
app.get('/', index)
app.get('/logout', logout)

function index(req, res) {
    var a = req.headers.authorization
    if (!a || !isAuthorized(a)) {
        res.header('WWW-Authenticate', 'Basic')
        res.status(401).send("Log in by entering username:password=rice:h00t
    } else {
        res.send('authorized')
    }
}
```

A Basic challenge

**Authorization:** `Basic cmljZTpoMDB0`

```
app.get('/', index)
app.get('/logout', logout)

function index(req, res) {
    var a = req.headers.authorization
    if (!a || !isAuthorized(a)) {
        res.header('WWW-Authenticate', 'Basic')
        res.status(401).send("Log in by entering username:password=rice:h00t
    } else {
        res.send('authorized')
    }
}

function logout(req, res) {
    var a = req.headers.authoriza
    if (a && isAuthorized(a)) {
        res.header('WWW-Authentic
        res.status(401).send("Log
    } else {
        res.send("Logged Out")
    }
}
```

A Basic challenge

```
function isAuthorized(auth) {
    var as = auth.split(' ')
    if ('Basic' == as[0]) {
        var userpass = atob(as[1])
        console.log('basic auth', userpass)
        return ('rice:h00t' == userpass)
    } else {
        console.err('non basic auth', as)
    }
    return false
}
```

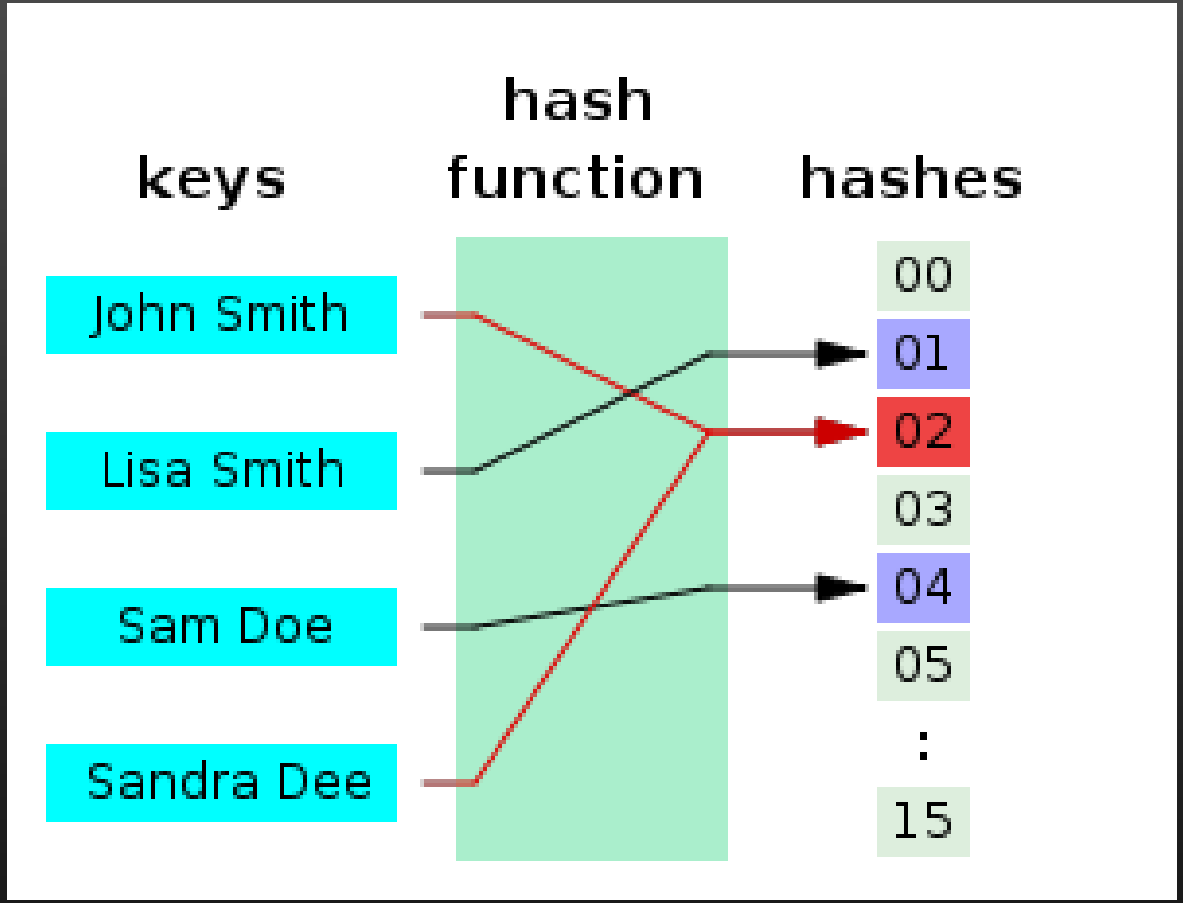
Base64 encoded

# Basic Auth Node Module

```
var basicAuth = require('basic-auth-connect')

app.use(basicAuth(auth))

var auth = function(user, pass) {
    return userMap[user].password == pass
}
```

Middleware on all routes added after this line

# Hashing




```
MD5("The quick brown fox jumps over the lazy dog") =
9e107d9d372bb6826bd81d3542a419d6
MD5("The quick brown fox jumps over the lazy dog.") =
e4d909c290d0fb1ca068ffaddf22cbd0
```

# HTTP AUTH Digest Challenge

```
HTTP/1.1 401 Unauthorized
X-Powered-By: Express
WWW-Authenticate: Digest realm="webdev-dummy@herokuapp.com",
                  qop="auth",
                  nonce="16d6a21279852f4292d9980b213610dd",
                  opaque="1018c187c32e0c5f66c3f0aeff5633de"
Content-Type: text/html; charset=utf-8
Content-Length: 46
```

```
Authorization: Digest username: 'rice',
  realm: 'webdev-dummy@herokuapp.com',
  nonce: '16d6a21279852f4292d9980b213610dd',
  uri: '/', qop: 'auth', nc: '00000001',
  response: '7a5e2bf103d0cc7643c124fcc5c2db7d',
  opaque: '1018c187c32e0c5f66c3f0aeff5633de',
  cnonce: 'a909c92d1ef4070b'
```

Password is in response

```
Digest realm="webdev-dummy@herokuapp.com",
qop="auth",
nonce="16d6a21279852f4292d9980b213610dd",
opaque="1018c187c32e0c5f66c3f0aeff5633de"
```

*I* "tied" together opaque and nonce. This way you must know both the nonce and the opaque value to hack into the system.

```
var nonce = _sec.getNonce();
res.header('WWW-Authenticate',
    'Digest realm="'+_sec.realm
    +'",qop="'+_sec.qop
    +'",nonce="'+nonce
    +'",opaque="'+_sec.getOpaque(nonce)
    +'"')
```

```
_sec = (function() {
    var SECRET = md5("This is my secret message")
    // this should be an LRU
    var nonceCache = {}

    function getOpaque(nonce) {
        return md5(nonce + SECRET)
    }

    return {
        realm: 'webdev-dummy@herokuapp.com',
        qop: 'auth',
        getNonce: getNonce,
        getOpaque: getOpaque
    }
})()
```

# Recall the basic challenge

```
app.get('/', index)
app.get('/logout', logout)

function index(req, res) {
    var a = req.headers.authorization
    if (!a || !isAuthorized(a)) {
        res.header('WWW-Authenticate', 'Basic')
        res.status(401).send("Log in by entering username:password=rice:h00t
    } else {
        res.send('authorized')
    }
}

function logout(req, res) {
    var a = req.headers.authoriza
    if (a && isAuthorized(a)) {
        res.header('WWW-Authentic
        res.status(401).send("Log
    } else {
        res.send("Logged Out")
    }
}
```

```
function isAuthorized(auth) {
    var as = auth.split(' ')
    if ('Basic' == as[0]) {
        var userpass = atob(as[1])
        console.log('basic auth', userpass)
        return ('rice:h00t' == userpass)
    } else {
        console.err('non basic auth', as)
    }
    return false
}
```

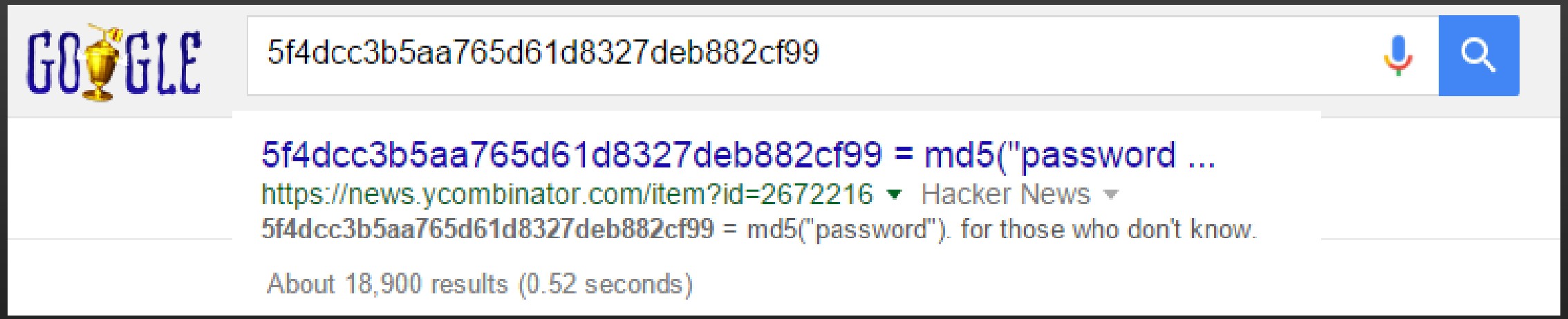
# Digest Authentication

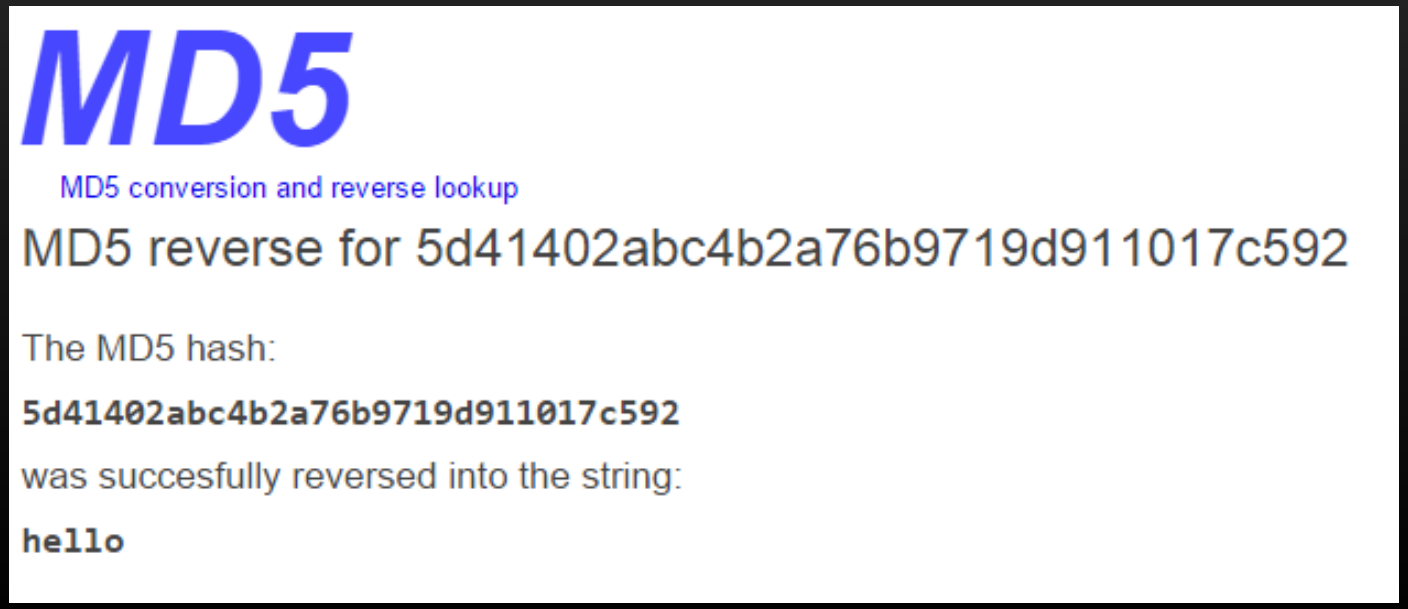
```
var kv = {}
as.forEach(function(v) {
    var s = v.replace(/,$/,'')
             .replace(/"/g, '')
             .split('=')
    kv[s[0]] = s[1]
})
```

```
Authorization: Digest username: 'rice',
  realm: 'webdev-dummy@herokuapp.com',
  nonce: '16d6a21279852f4292d9980b213610dd',
  uri: '/', qop: 'auth', nc: '00000001',
  response: '7a5e2bf103d0cc7643c124fcc5c2db7d',
  opaque: '1018c187c32e0c5f66c3f0aeff5633de',
  cnonce: 'a909c92d1ef4070b'
```

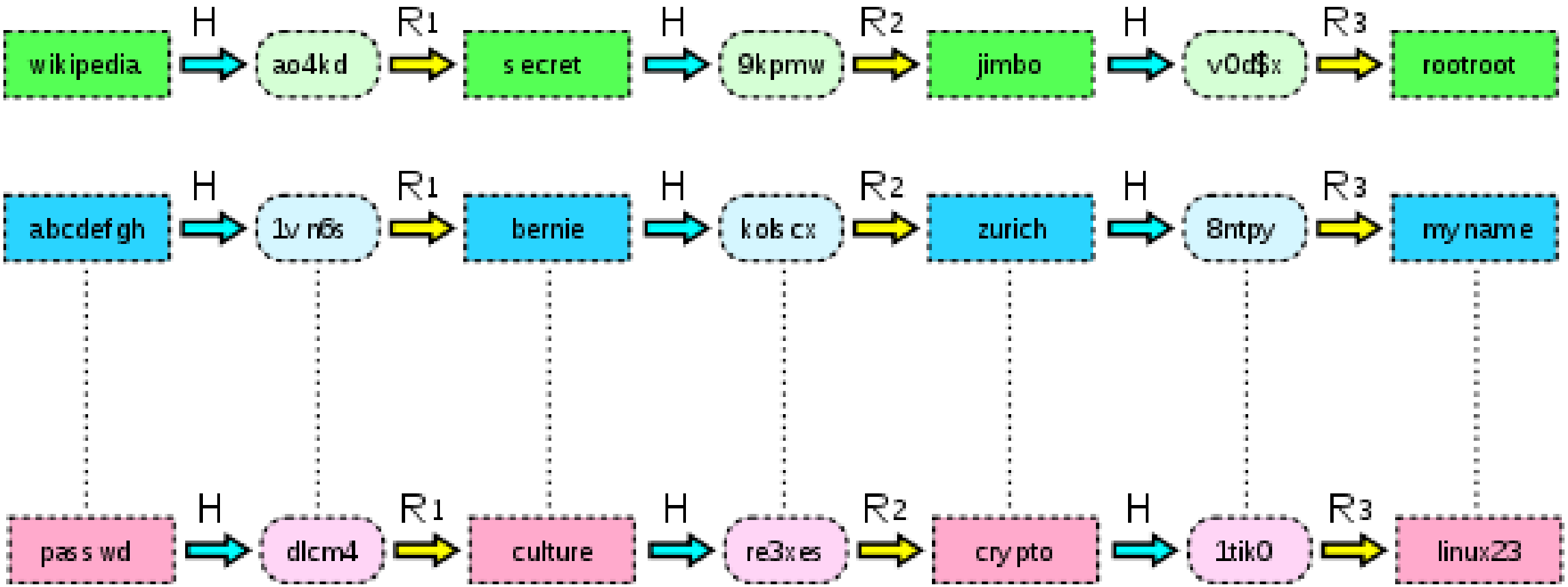
```
// validate the nonce and opaque match
if (_sec.getNonce(kv.opaque) != kv.nonce) {
    console.warn("Nonce for opaque did not match.")
    return false
}

// we *never* need to know this
var password = 'h00t'
// instead store kv.username -> ha1 in our database
var ha1 = md5([kv.username, kv.realm, password].join(':'))

var ha2 = md5([req.method, req.url].join(':'))
var response = md5( [ha1, kv.nonce, kv.nc, kv.cnonce, kv.qop, ha2 ].join(':') )
return (response == kv.response)
```

# Hash lookup


GOOGLE
5f4dcc3b5aa765d61d8327deb882cf99
5f4dcc3b5aa765d61d8327deb882cf99 = md5("password ...
https://news.ycombinator.com/item?id=2672216 Hacker News
5f4dcc3b5aa765d61d8327deb882cf99 = md5("password"). for those who don't know.
About 18,900 results (0.52 seconds)

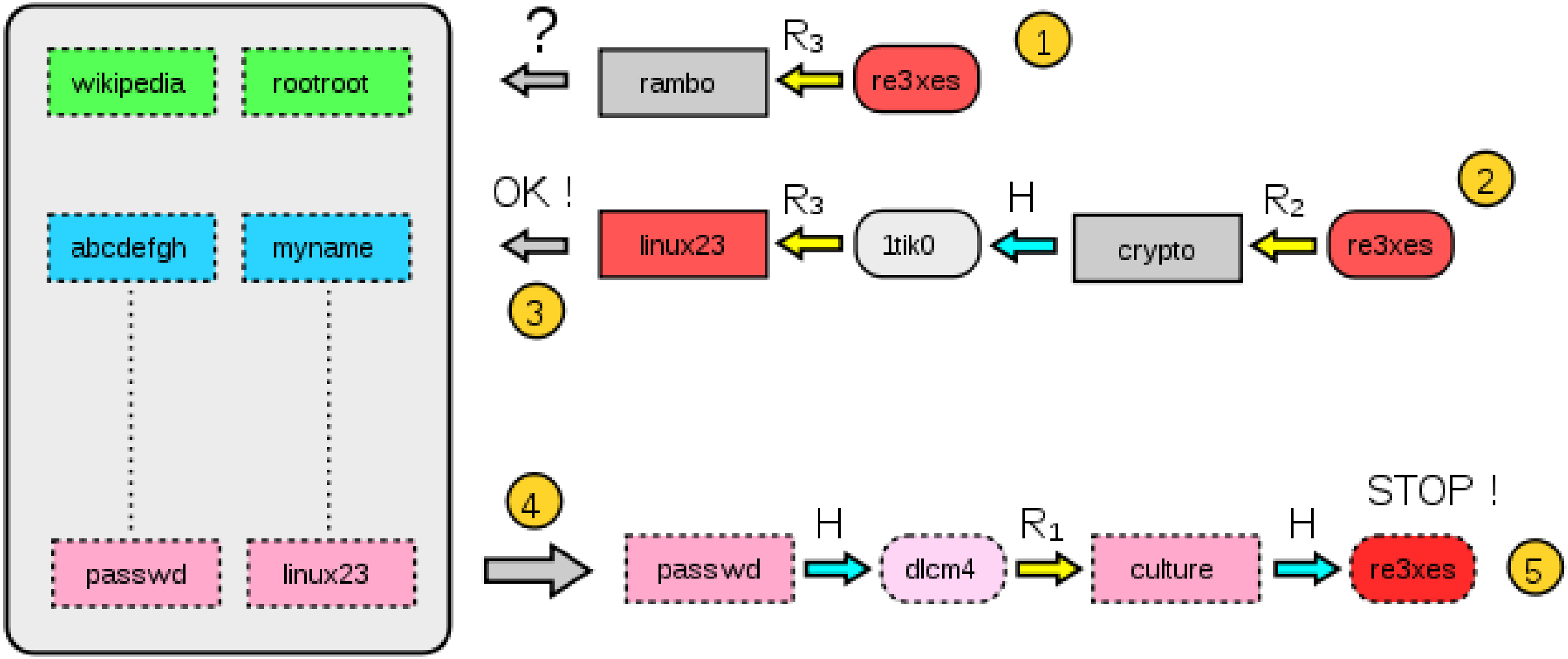

MD5
MD5 conversion and reverse lookup
MD5 reverse for 5d41402abc4b2a76b9719d911017c592
The MD5 hash:
5d41402abc4b2a76b9719d911017c592
was succesfully reversed into the string:
hello

# Rainbow Table




https://en.wikipedia.org/wiki/Rainbow_table

# Using Rainbow Table




https://en.wikipedia.org/wiki/Rainbow_table

# Is your Windows password already cracked?

| length: | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|
| LM-enabled | completely cracked | completely cracked | completely cracked | completely cracked | completely cracked | completely cracked | completely cracked | safe |
| simple passwords | completely cracked | lower, upper, numeric cracked | numeric cracked | numeric cracked | numeric cracked | safe | safe | safe |
| complex passwords | completely cracked | CLN cracked | CLN cracked | CLN cracked | safe | safe | safe | safe |

All data is based on rainbow tables available at FreeRainbowTables.com. Rainbow tables can be used to "decrypt" a password.
"LM-enabled" means the computer has LM password storage turned on. This is the default for Windows 2003/Vista and earlier.
"Complex" means it contains characters from at least three of the four categories (uppercase, lowercase, numbers, symbols).
"CLN" means it is in capital-lowercase-number format (ex. Password123).
Data compiled by Nick Brown ( http://nick-brown.com ). Last updated: February, 2012.

https://en.wikipedia.org/wiki/Rainbow_table

# Defense: Salting

A rainbow table is ineffective against one-way hashes that include large salts. For example, consider a password hash that is generated using the following function (where "+" is the concatenation operator):

```
saltedhash(password) = hash(password + salt)
```

Or

```
saltedhash(password) = hash(hash(password) + salt)
```

The salt value is not secret and may be generated at random and stored with the password hash. A large salt value prevents precomputation attacks, including rainbow tables, by ensuring that each user's password is hashed uniquely. This means that two users with the same password will have different password hashes (assuming different salts are used).

# Salted Passwords

- Pre-Salt Plan of Attack:
  - Create a look up table of every *n*-character password to hash (slow)

    OR
  - Use a rainbow table of every *n*-character password to hash (faster)

- The salt is typically public
  - Now they have to have a larger *n*-character lookup table
- Salted Plan of attack:
  - Take the salt, generate a table from it
  - We're in!

*It just takes time…*

# Peppering

*…security through obscurity*

- Note that we have a different salt for each user
- This salt is in the database
- If the database is compromised an attacker can get it by making a lookup table
- Pepper is a secret code on the server, not in the database

```
var pepper = md5("This is my secret pepper")

var password = getPasswordFromRequest()
var salt     = getSaltForUserFromDB( getUserFromRequest() )
var answer   = getHashForUserFromDB( getUserFromRequest() )
var hash     = md5( salt + password + pepper )
```

# Security, security, security

You don't want to be hacked

- Hash on the browser? Sure.
- Hash on the server? Definitely

- MD5 and SHA-1 are now "trivial" do not use them

H(H(H(H(H(H(H(H(….. H(password + salt) + salt) + salt) …..)

- Instead use a Key-Derivation Function (KDF)

such as PBKDF2 or bcrypt / scrypt

# In-Class Exercise: Add Register and Login to App

- Create a User model in *model.js*
- Add the `POST /register` endpoint to your app
- Take the password add a random salt and create a hash
- Store username, salt, and hash in the database
- Add the `POST /login` endpoint to your app
- For the supplied username, lookup the salt in the database
- Combine the salt with the password to derive the hash
- Compare the derived hash with the hash from the database
- If they match, set a cookie for the user (`npm install cookie-parser --save`)
  - Store the session id in an in-memory map from session to user

*Salt is a random string/user*

*you can use md5 for hashing*

```
npm install md5 --save
```

*see slide 5*

***Turnin auth.js and model.js to COMP431-S16:inclass-21***