Web Development

COMP 431 / COMP 531

Third Party Authorization

Scott E Pollack, PhD

April 7, 2016

Part IIb – Back End Development

- 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

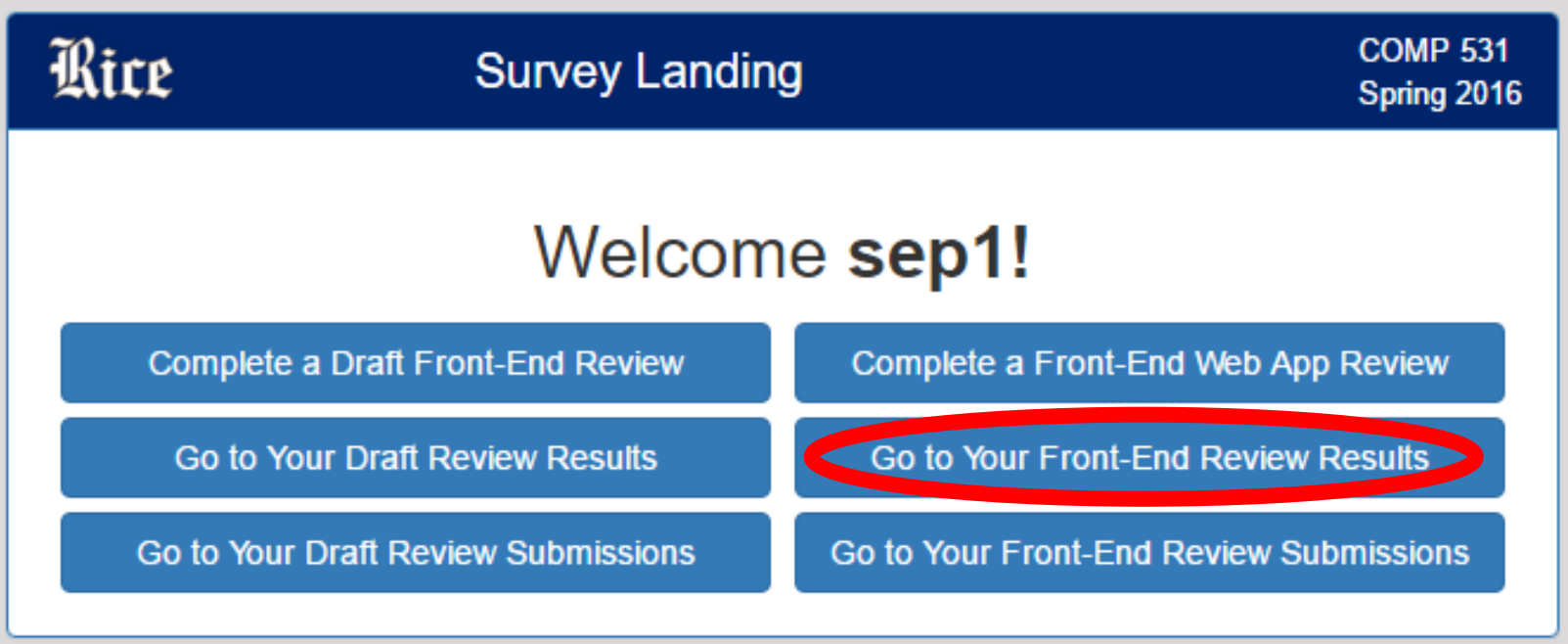
Service APIs

Integrating

Front-End Reviews *are available!*

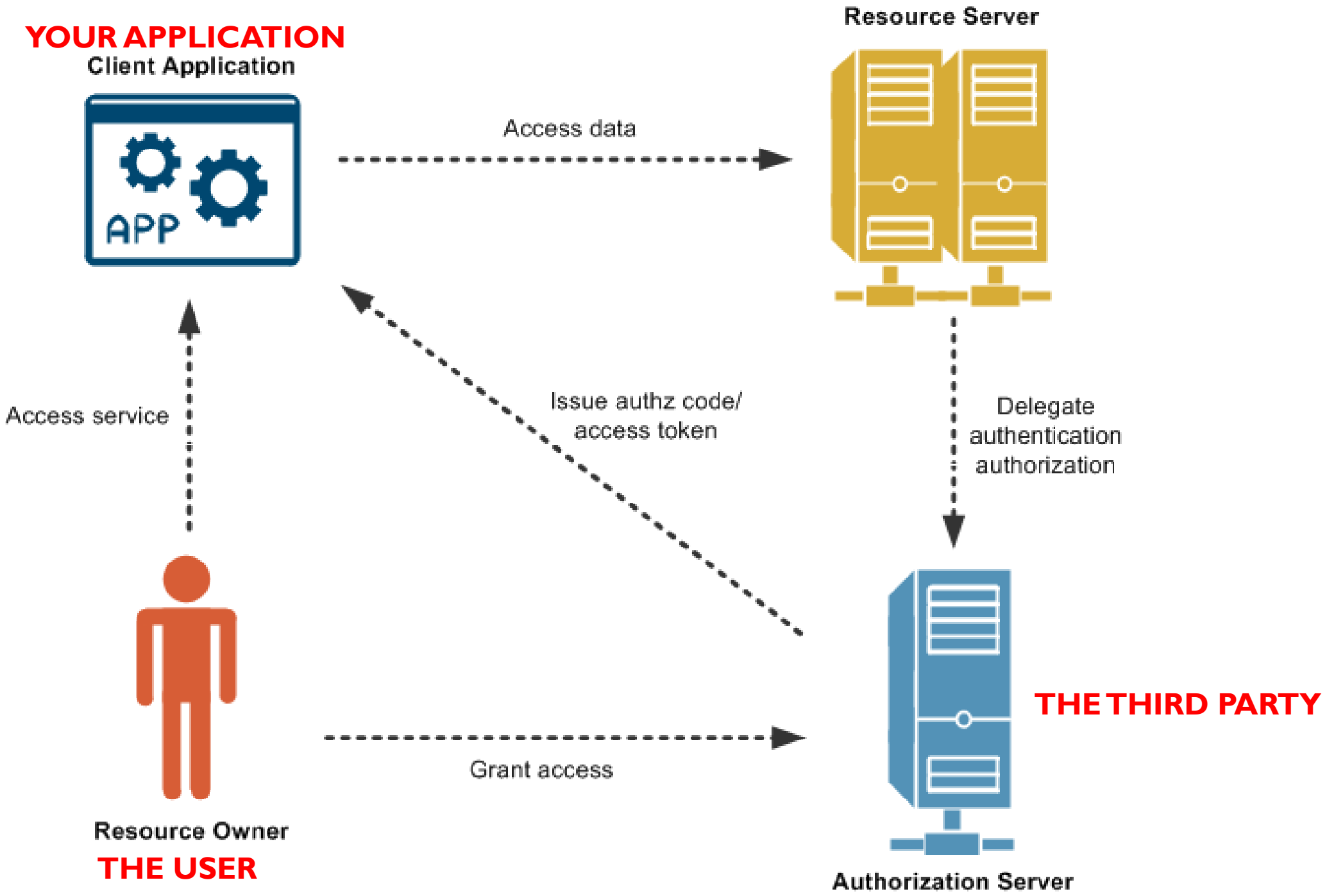
http://webdev-dummy.herokuapp.com/survey

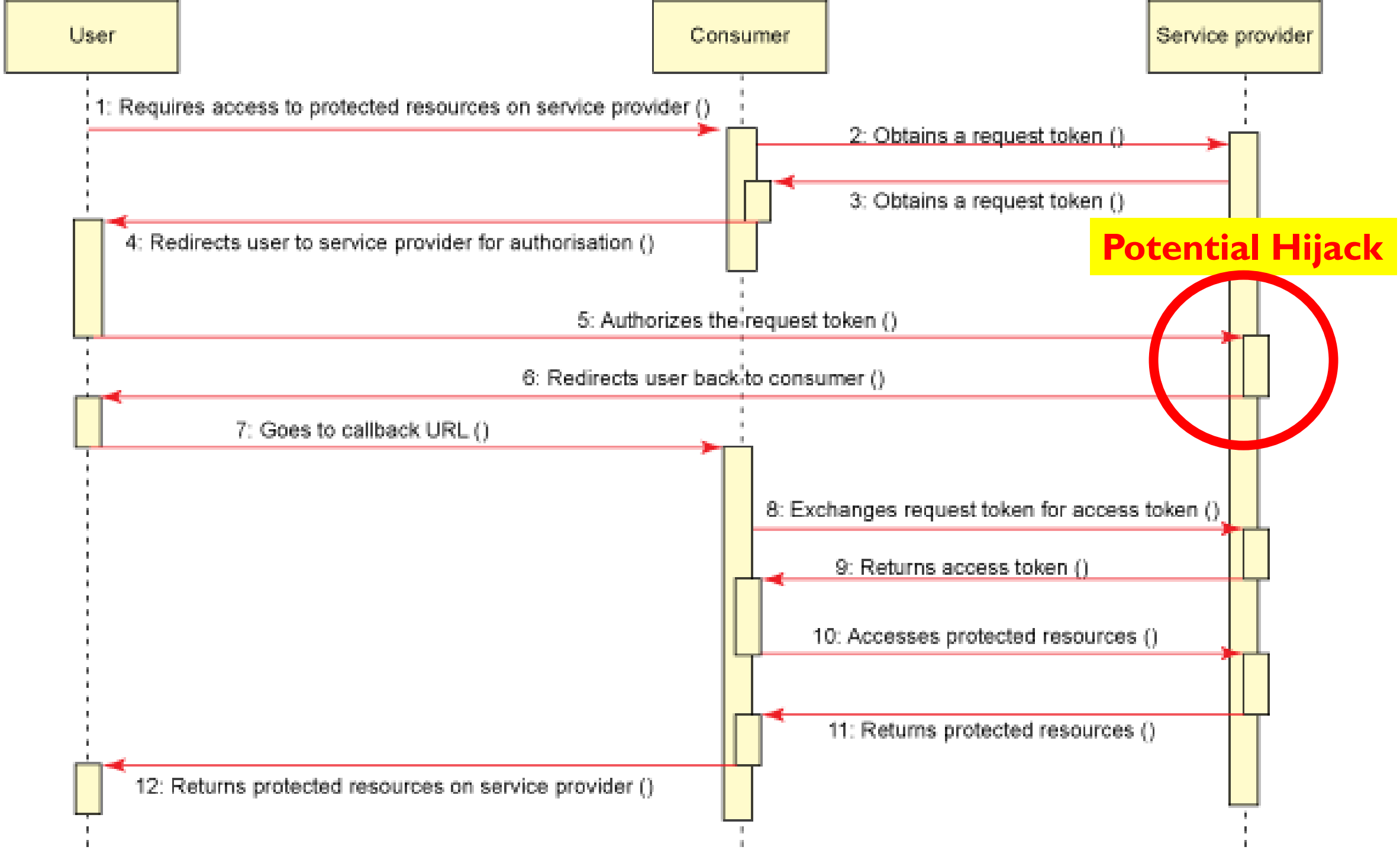
- Go to this address and log in using your netid and 3-word password supplied to you by email for the dummy server



Open standard for Authorization (OAuth)

- Credential delegation
- Resource Owner has relationship with Third-Party Authorization Server
- User authenticates with Third-Party
- Third-party issues access token for user to access Resource

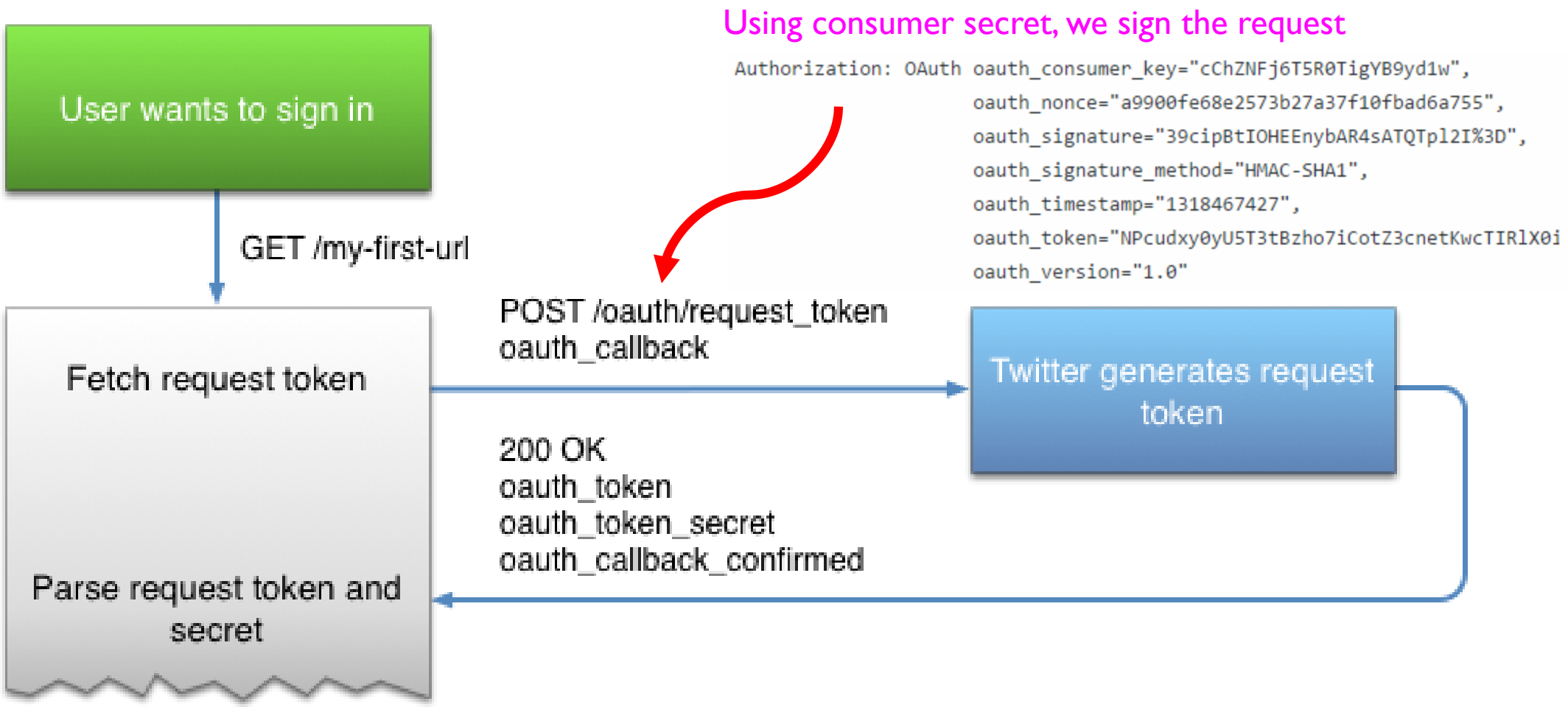

YOUR APPLICATION
Client Application
APP
Resource Server
Access data
Access service
Issue authz code/
access token
Delegate
authentication
authorization
Grant access
THE THIRD PARTY
Resource Owner
THE USER
Authorization Server


User
Consumer
Service provider
1: Requires access to protected resources on service provider ()
2: Obtains a request token ()
3: Obtains a request token ()
4: Redirects user to service provider for authorisation ()
Potential Hijack
5: Authorizes the request token ()
6: Redirects user back to consumer ()
7: Goes to callback URL ()
8: Exchanges request token for access token ()
9: Returns access token ()
10: Accesses protected resources ()
11: Returns protected resources ()
12: Returns protected resources on service provider ()

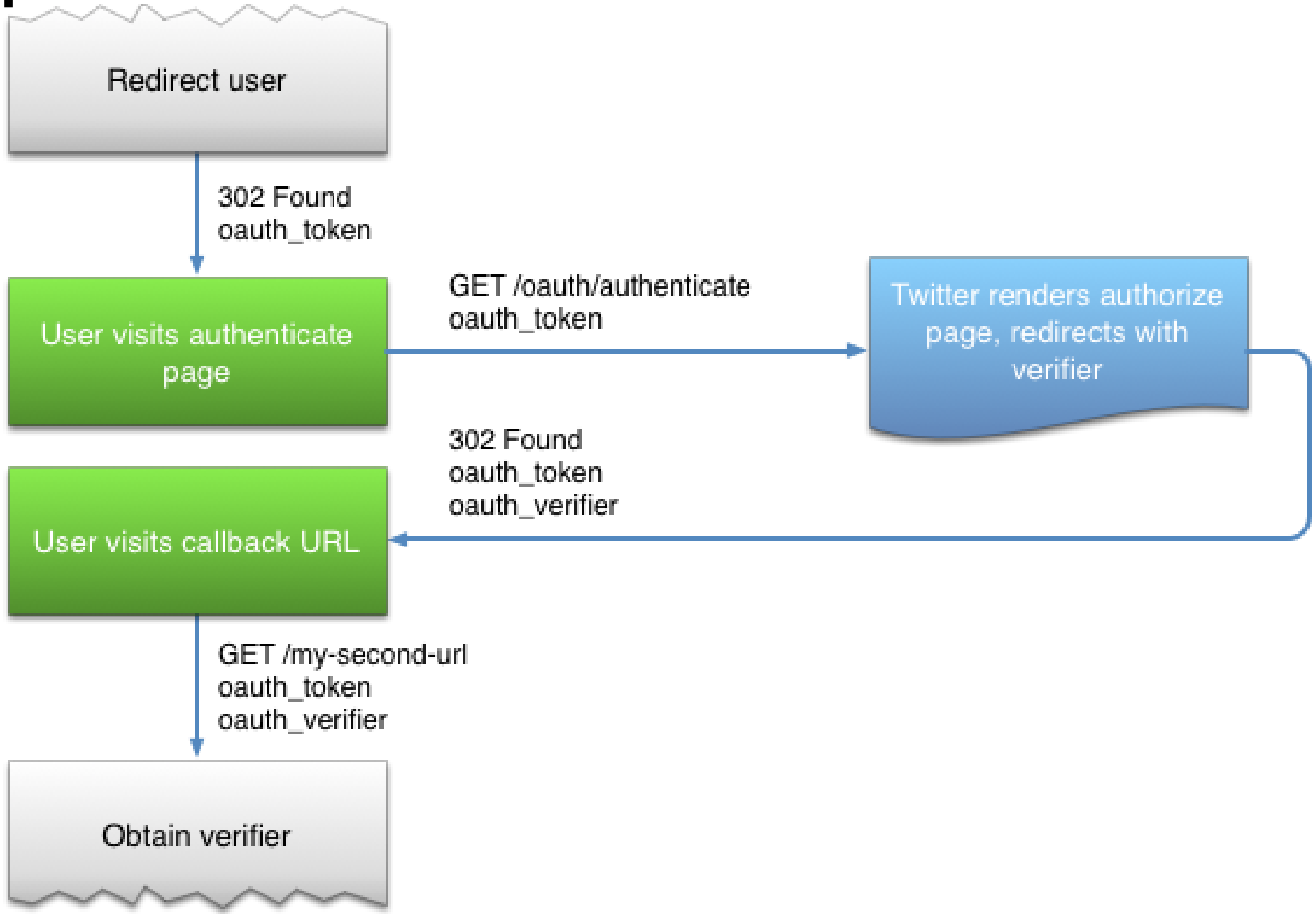
Credentials and Tokens

- The Client Application developer (that's you) registered with the Authorization Server and receives a client consumer key and secret
- The request token from the Server is arbitrary
- The Resource Owner (the user) takes the request token and gives it to the Authorization Server when they authenticate
  - Now the Authorization Server knows that the request token is for this specific user
  - The Server confirms that the Owner grants specific access for the Client
  - The Server then redirects the Owner to the site specified by the Client
- The Client then exchanges the request token for an Access Token

Example with Twitter: 1. Get Request Token



Example with Twitter: 2. Redirect User for Login



https://api.twitter.com/oauth/authenticate?oauth_token=ihOnQAAAAAAAh7E0AAABU

Sign up for Twitter ›

Authorize webdev-dummy to use your account?

webdev-dummy

webdev-dummy.herokuapp.com

Dummy Web Server

Username or email

Password

Remember me · Forgot password?

Sign In Cancel

This application will be able to:

- Read Tweets from your timeline.
- See who you follow, and follow new people.
- Update your profile.
- Post Tweets for you.

Will not be able to:

- Access your direct messages.
- See your Twitter password.

https://<CALLBACK_URL>?oauth_token=ihOnQAAVA71E0AAABUMR9phM&oauth_verifier=lhav0JQnDtV4APqfhKxk15ZB4wwB

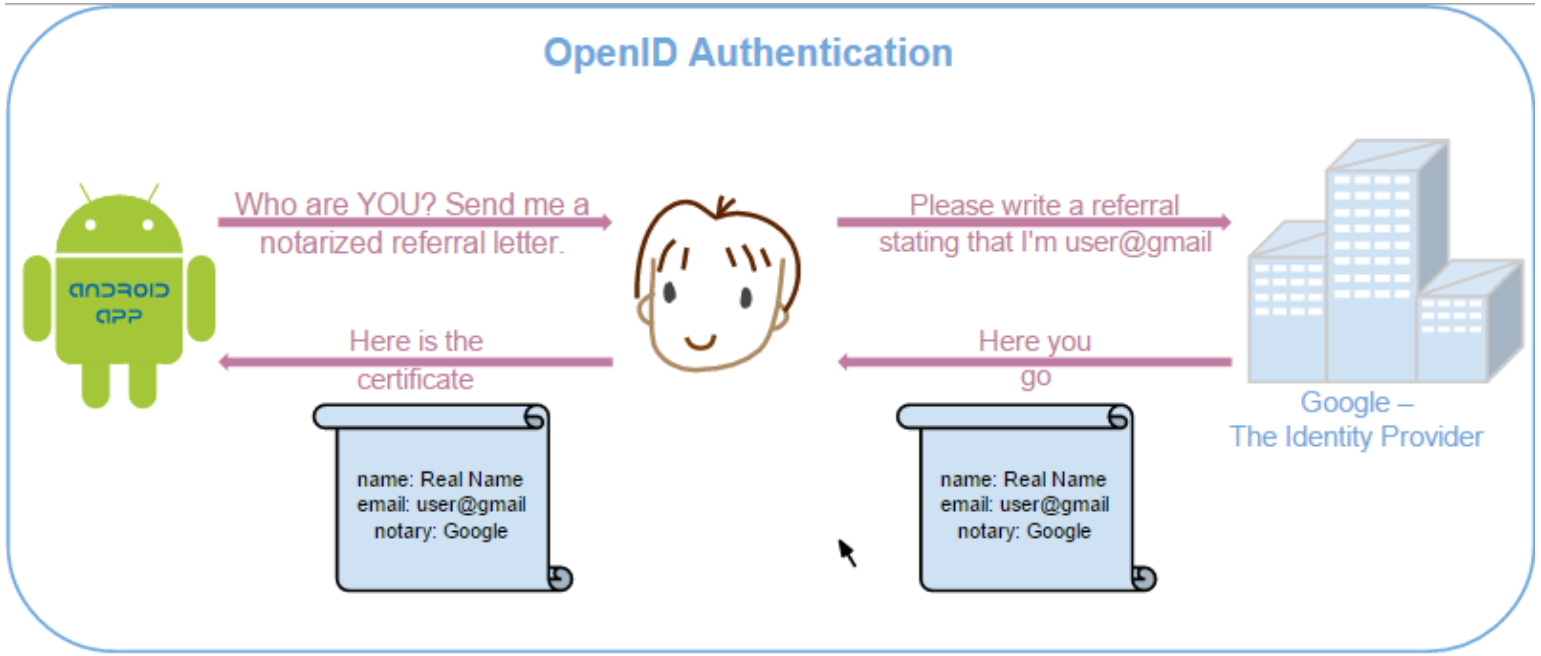
Example with Twitter: 3. Convert to Access Token

<u>Sweet and simple</u>

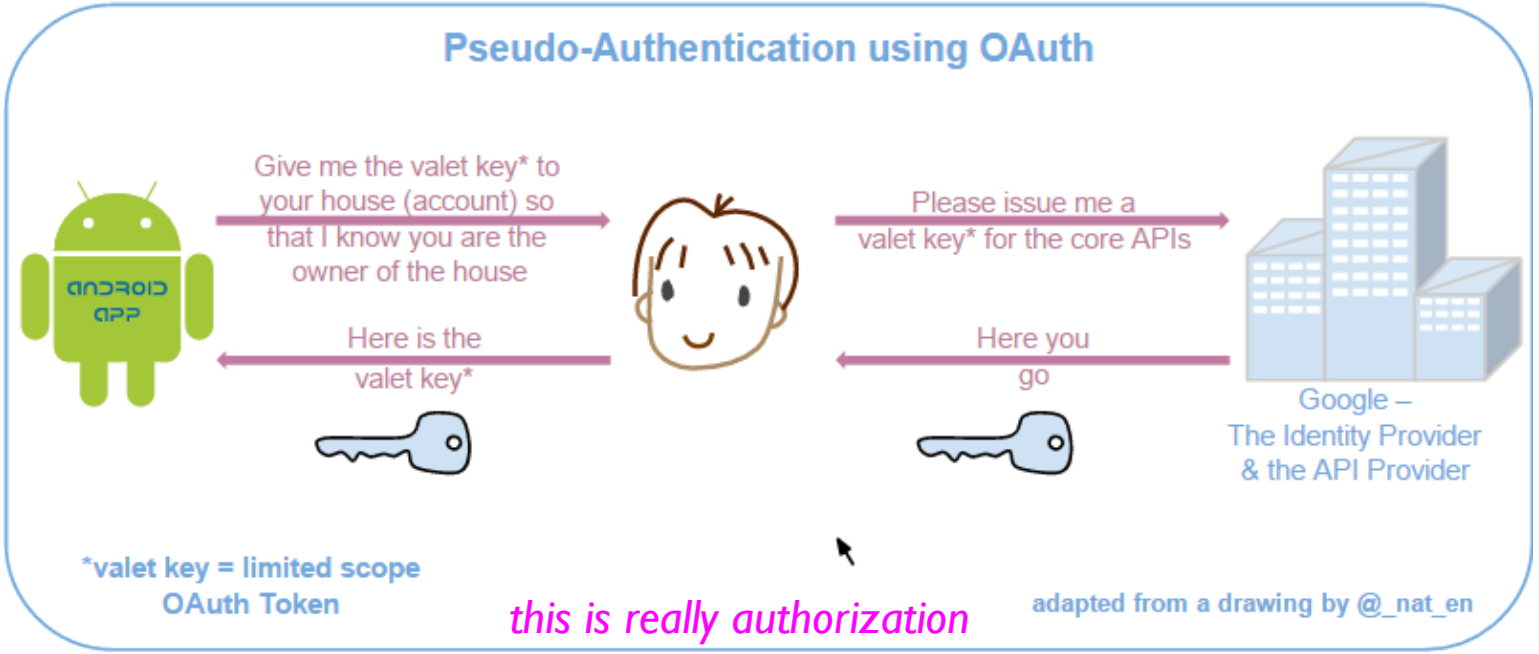
Now store the token … in a cookie?

What did we do?

- Traded username and password for token key and secret
- Why is this good?
  - Because the user doesn't have to remember another password
- How does this help us?
  - Well…
  - We still have all the same problems as before, just with the token now
  - Or do we?
  - Encrypt the token key and secret and store in browser (e.g., cookie)
  - This at least alleviates us from caching these values on the server
  - MITM/CSRF can still hijack the user's tokens


OpenID Authentication
Who are YOU? Send me a notarized referral letter.
Please write a referral stating that I'm user@gmail
Here is the certificate
Here you go
name: Real Name
email: user@gmail
notary: Google
name: Real Name
email: user@gmail
notary: Google
Google – The Identity Provider

vs.


Pseudo-Authentication using OAuth
Give me the valet key* to your house (account) so that I know you are the owner of the house
Please issue me a valet key* for the core APIs
Here is the valet key*
Here you go
Google – The Identity Provider & the API Provider
*valet key = limited scope OAuth Token
this is really authorization
adapted from a drawing by @_nat_en

OAuth2

- v2 is always better than v1?

hueniverse.com/2012/07/26/oauth-2-0-and-the-road-to-hell/

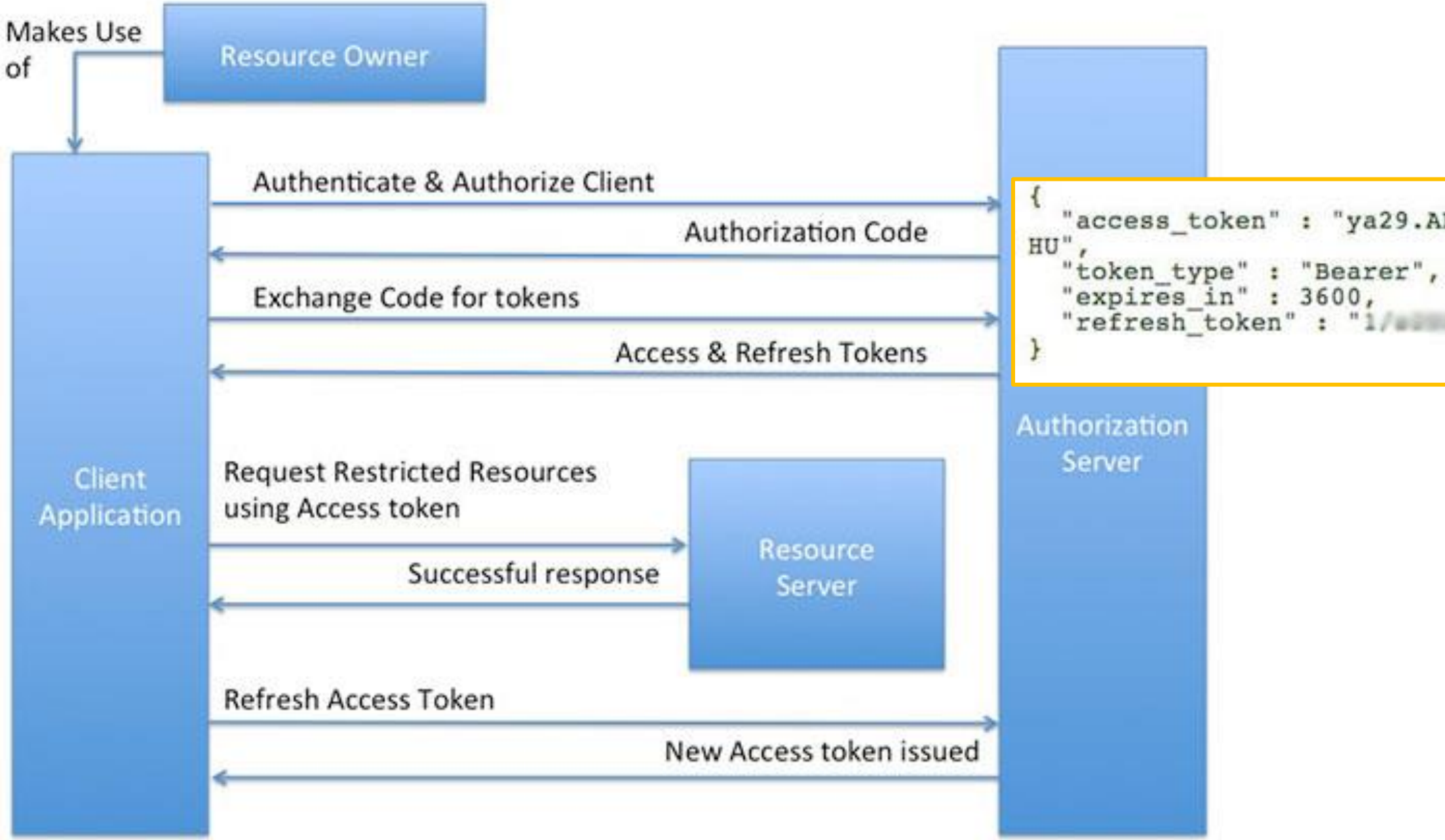
OAuth 2.0 and the Road to Hell

They say the road to hell is paved with good intentions. Well, that's OAuth 2.0.

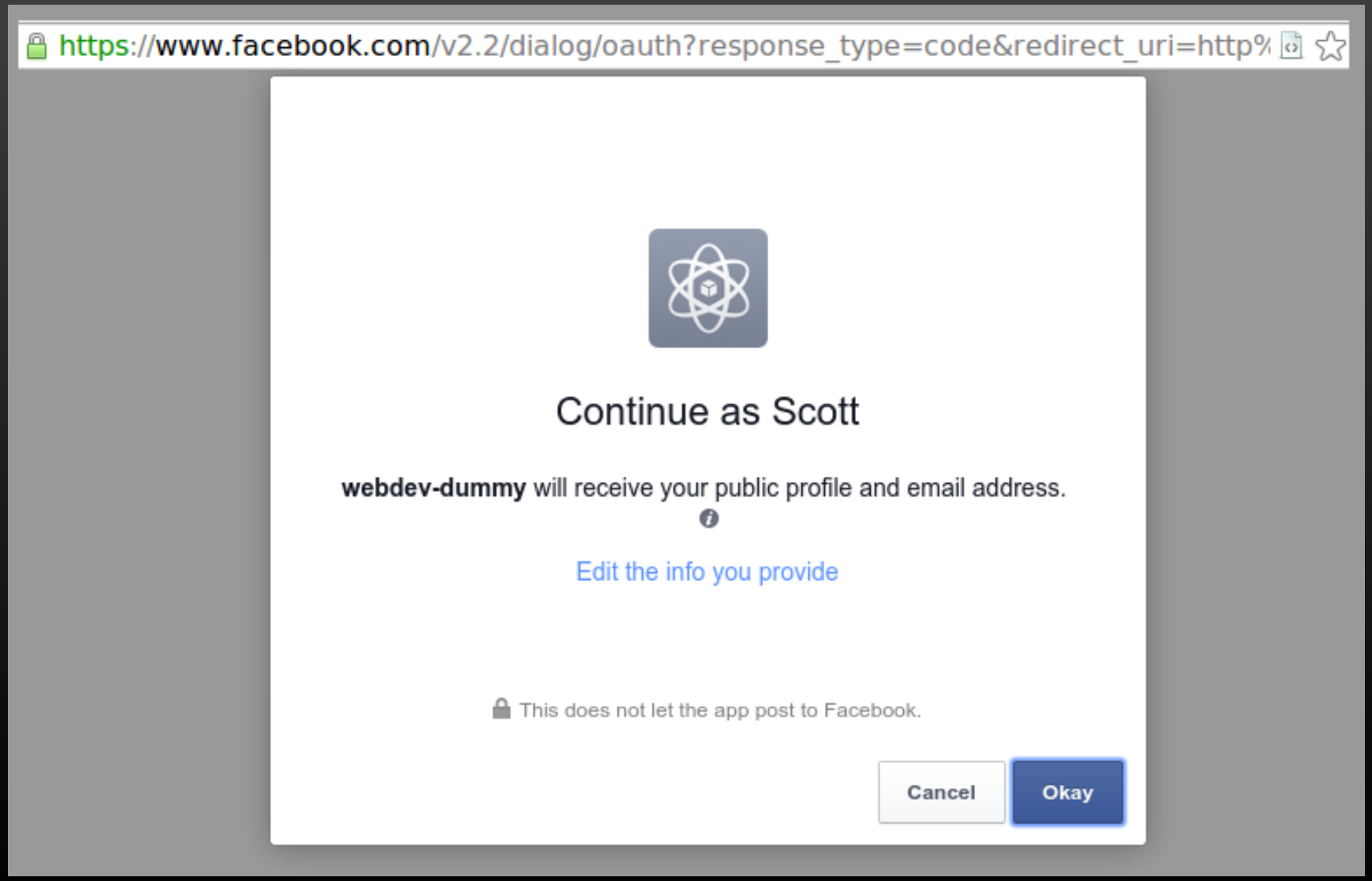
Last month I reached the painful conclusion that I can no longer be associated with the OAuth 2.0 standard. I resigned my role as lead author and editor, **withdraw my name from the specification**, and left the working group. Removing my name from a document I have painstakingly labored over for three years and over two dozen drafts was not easy. Deciding to move on from an effort I have led for over five years was agonizing.

Big Differences

- OAuth 2 should sit inside TLS/SSL
  - *Therefore no signing of every request (curl)*
- Native Applications (mobile apps, desktop apps, TV / game consoles)
  - *OAuth 1 was designed for browsers*
- OAuth 1 had "inherent complexity" and was "hard to use"
- Separation of Authentication Server from Resource Server
- Flows for various different types of clients
- Tokens have finite lives and must be refreshed


Makes Use of
Resource Owner
Client Application
Authenticate & Authorize Client
Authorization Code
Exchange Code for tokens
Access & Refresh Tokens
Request Restricted Resources using Access token
Successful response
Resource Server
Refresh Access Token
New Access token issued
Authorization Server
{
"access_token" : "ya29.A
HU",
"token_type" : "Bearer",
"expires_in" : 3600,
"refresh_token" : "1/
}

No Real Change for the User



OAuth in NodeJS

```
app.use('/login', login)
app.use('/callback', authCallback)
app.use('/get', get)

var authRequests = {}
var authTokens = {}

function login(req, res) {
    var oauth = {
        callback: LOCALHOST + '/callback',
        consumer_key: config.consumer_key,
        consumer_secret: config.consumer_secret
    }
    request.post({url: endpoints.OA_REQ, oauth:oauth}, function(err, req, body) {
        var req_data = qs.parse(body)
        authRequests[req_data.oauth_token] = req_data
        var uri = endpoints.AUTHENTICATE
            + '?' + qs.stringify({oauth_token: req_data.oauth_token})
        res.redirect(uri)
    })
}
```

OAuth in NodeJS

```
app.use('/login', login)
app.use('/callback', authCallback)
app.use('/get', get)

var authRequests = {}
var authTokens = {}

function login(req, res) {
    var oauth = {
```

```
function authCallback(req, res) {
    var auth_data = req.query
    var oauth = {
        consumer_key: config.consumer_key,
        consumer_secret: config.consumer_secret,
        token: auth_data.oauth_token,
        token_secret: authRequests[auth_data.oauth_token].oauth_token_secret,
        verifier: auth_data.oauth_verifier
    }
    request.post({ url: endpoints.OA_ACCESS, oauth:oauth }, function(err, req, body) {
        var perm_data = qs.parse(body)
        // Store the token as a cookie for the user
        // maybe better just to derive our own session key
        res.cookie('token', perm_data.oauth_token)
        authTokens[perm_data.oauth_token] = perm_data
        res.redirect(LOCALHOST + '/get')
    })
}
```

OAuth in NodeJS

```
app.use('/login', login)
app.use('/callback', authCallback)
app.use('/get', get)

var authRequests = {}
var authTokens = {}

function login(req, res) {

 LOCALHOST + '/callback',
key: config.consumer_key,
```

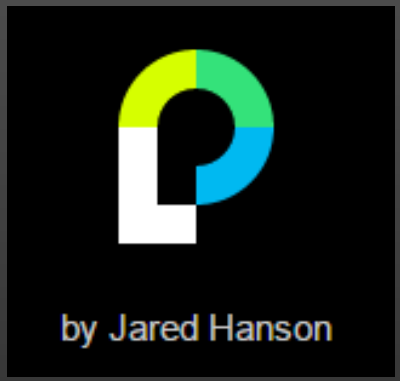
```
function authCallback(req, res) {
    var auth_data = req.query
    var oauth = {

    }
    re

    })
}
```

```
function get(req, res) {
    var token = req.cookies['token']
    var perm_data = authTokens[token]
    console.log('user is ', perm_data)
    var oauth = {
        consumer_key: config.consumer_key,
        consumer_secret: config.consumer_secret,
        token: token,
        token_secret: perm_data.oauth_token_secret
    }
    var params = {
        screen_name: perm_data.screen_name,
        user_id: perm_data.user_id
    }
    var url = endpoints.REST_ROOT + 'users/show.json'
    request.get({url:url, oauth:oauth, qs:params, json:true}, function (err, req, user) {
        console.log(user)
        res.send(user)
    })
}
```

PassportJS



- One stop shopping:

  HTTP Basic, HTTP Digest, OAuth, OAuth 2.0, OpenID, Facebook, Twitter, Google, … over 300 strategies…

```
var request          = require('request')
var qs               = require('querystring')
var express          = require('express')
var cookieParser     = require('cookie-parser')
var session          = require('express-session')
var passport         = require('passport')
var FacebookStrategy = require('passport-facebook').Strategy;

app = express()
app.use(session({ secret: 'thisIsMySecretMessageHowWillYouGuessIt'}))
app.use(passport.initialize());
app.use(passport.session())
app.use(cookieParser());
```

```
// serialize the user for the session
passport.serializeUser(function(user, done) {
  users[user.id] = user
  done(null, user.id)
})

// deserialize the user from the session
passport.deserializeUser(function(id, done) {
  var user = users[id]
  done(null, user)
})

passport.use(new FacebookStrategy(config,
  function(token, refreshToken, profile, done) {
    process.nextTick(function() {
      // register the user in our system
      return done(null, profile)
    })
  })
)
```

users[] is a proxy for db

```
function logout(req, res) {
  req.logout();
  res.redirect('/')
})

function isLoggedIn(req, res, next) {
  if (req.isAuthenticated()) {
    next()
  } else {
    res.redirect('/login')
  }
}

function profile(req, res) {
  res.send('ok now what?', req.user)
}
```

by Jared Hanson

```
app.use('/login',    passport.authenticate('facebook', { scope:'email' }))
app.use('/callback', passport.authenticate('facebook', {
  successRedirect: '/profile', failureRedirect :'/fail' }))
app.use('/profile', isLoggedIn, profile)
app.use('/fail', fail)
app.use('/logout', logout)
app.use('/', hello)
```

JSON Web Token

- Our backend is a REST server (The frontend is the app!)
- It should not have state
- It shouldn't keep track of user sessions – certainly not in memory
- Store the session on the client entirely
  - I.e., the token key and secret together encrypted
- A "jot" has three parts:

  header = type and hashing algorithm
  payload = data (visible to client – we'll use it in AngularJS)
  signature = encrypted hash of header + payload using your secret
  We'll use this to assure the JWT hasn't been tampered with

JWT to the client

```
passport.use(new FacebookStrategy(config,
  function(token, refreshToken, profile, done) {
    process.nextTick(function() {
      var expiry = new Date();
      expiry.setDate(expiry.getDate() + 1);
      var jsonToken = jwt.sign({
        token: token,
        profile: profile,
        exp: parseInt(expiry.getTime() / 1000)
      }, secret)
      return done(null, jsonToken )
    })
  })
)
```

```
function profile(req, res) {
  console.log(req.user)
  var decoded = jwt.verify(req.user, secret)
  var token = decoded.token
  var profile = decoded.profile
  res.send(req.user)
}
```

```
> s = jwt.split(".")
> s[0]
<· "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9"
> atob(s[0])
<· "{"typ":"JWT","alg":"HS256"}"
> atob(s[1])
<· "{"token":"CAAPypbsS6qoBABTI1lC2tgfbvJAK1J87ahO4VLFFnxuhmT7U
"
> atob(s[2])
✖ ▶Uncaught DOMException: Failed to execute 'atob' on          VM
'Window': The string to be decoded is not correctly encoded.
```

```
> decode = JSON.parse(atob(s[1]))
<· ▼Object {token: "CAAPypbsS6qoBABTI1lC2tgfbvJAK1J87
    exp: 1446524950
    iat: 1446438550
  ▼profile: Object
    ▶ json: Object
      _raw: "{"name":"Scott E Pollack","id":"[redacted]
      displayName: "Scott E Pollack"
      id: "[redacted]"
    ▶name: Object
      provider: "facebook"
    ▶ __proto__ : Object
    token: "CAAPypbsS6qoBABTI1lC2tgfbvJAK1J87ahO4VL
  ▶ __proto__ : Object
```

JWT Back to Server

- We'll send the JWT back to the server on all REST requests
- Instead of a cookie, we'll send an Authorization header

```
function getToken() {
  return $window.localStorage['webdev-dummy']
}
                    ...
headers: {
  Authorization: 'Bearer ' + getToken()
}
```

- Middleware such as express-jwt or various passport extensions can consume the Authorization header and provide req.token

In-Class Exercise: add OAuth login

- Decide which third-party you want to partner with, e.g., Facebook, Twitter, Google, …
- Become a developer on their site and register your (backend) app with them
- You'll get a client key and secret
- If you go with Twitter then you can use *request* for OAuth. For OAuth2 you can use *PassportJS* (you can use Passport for OAuth as well)
- Make a new endpoint to authenticate with your partner, e.g., `/auth/facebook`
- Upon successful authentication, register a session cookie
- Spin up your node backend and verify that it works using *curl* or *Adv REST Client*.
- Add a button to your frontend app (running using the python server) that hits the `/auth` endpoint and confirm you can login

turnin your* auth.js *with the endpoints and middleware

COMP431-S16:inclass-23