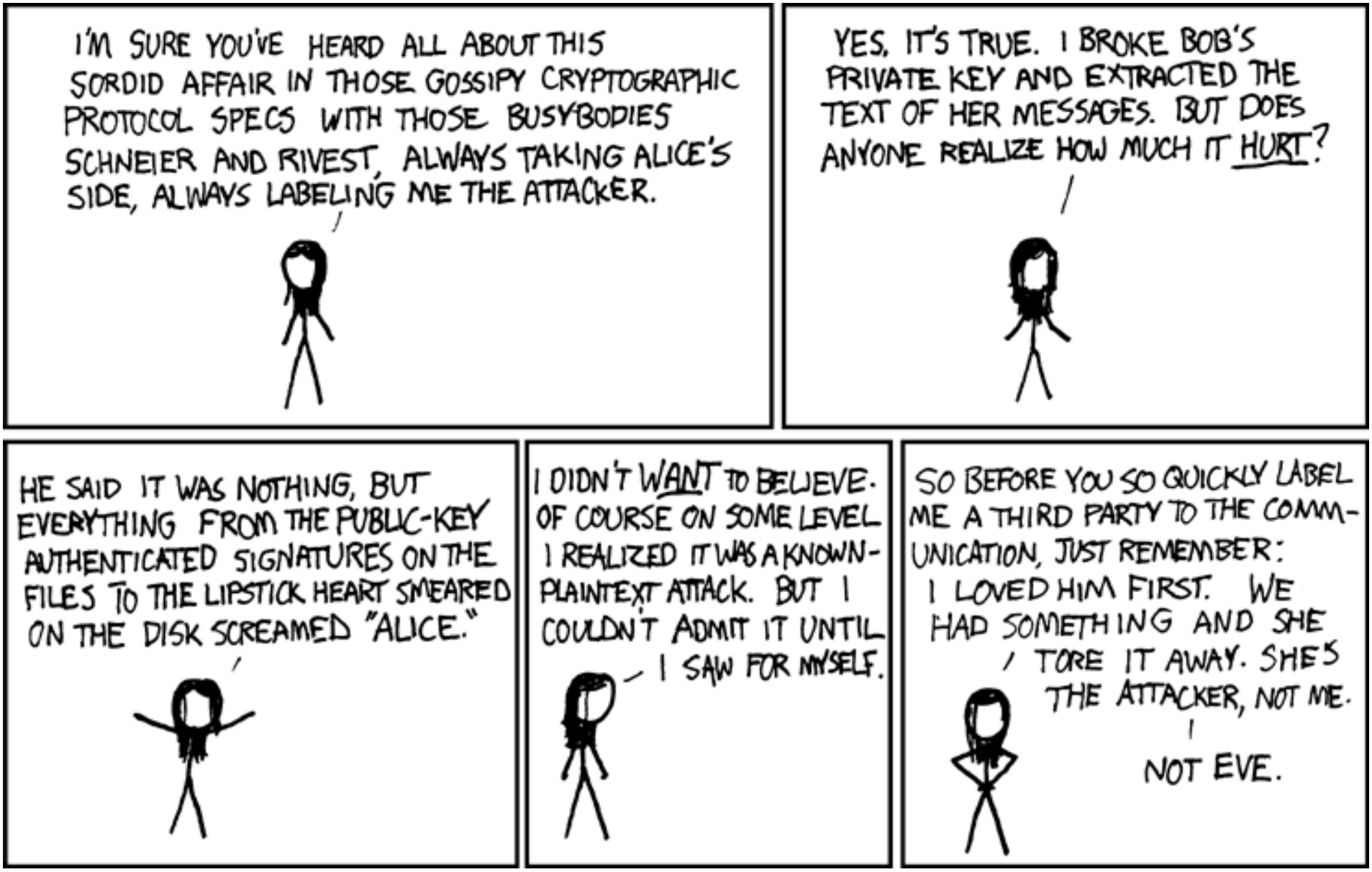
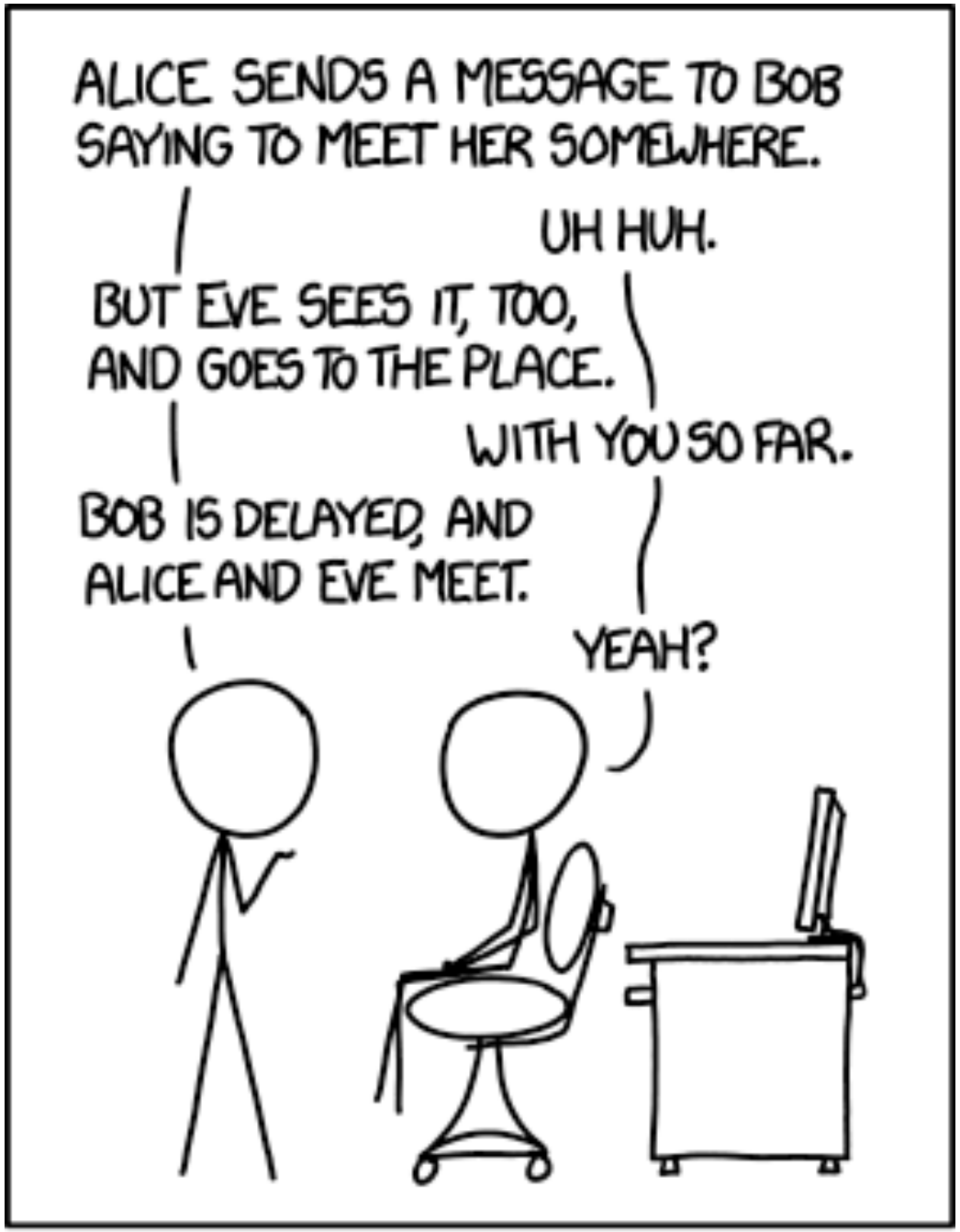



https://www.xkcd.com/177/




http://xkcd.com/1323/

# Security

owasp.org

# What could go wrong?

- Web app could allow attacker to
  - view information  (leak information)
  - modify information
  - perform operations on behalf of user
  - attack user's machine

# Who are you worried about?

- Insiders
- Criminals
- Commercial competitors
- Nation states (intelligence agencies and their proxies)
- Law enforcement
- Vandals, “security researchers”, “script kiddies”

# Threat Categorization

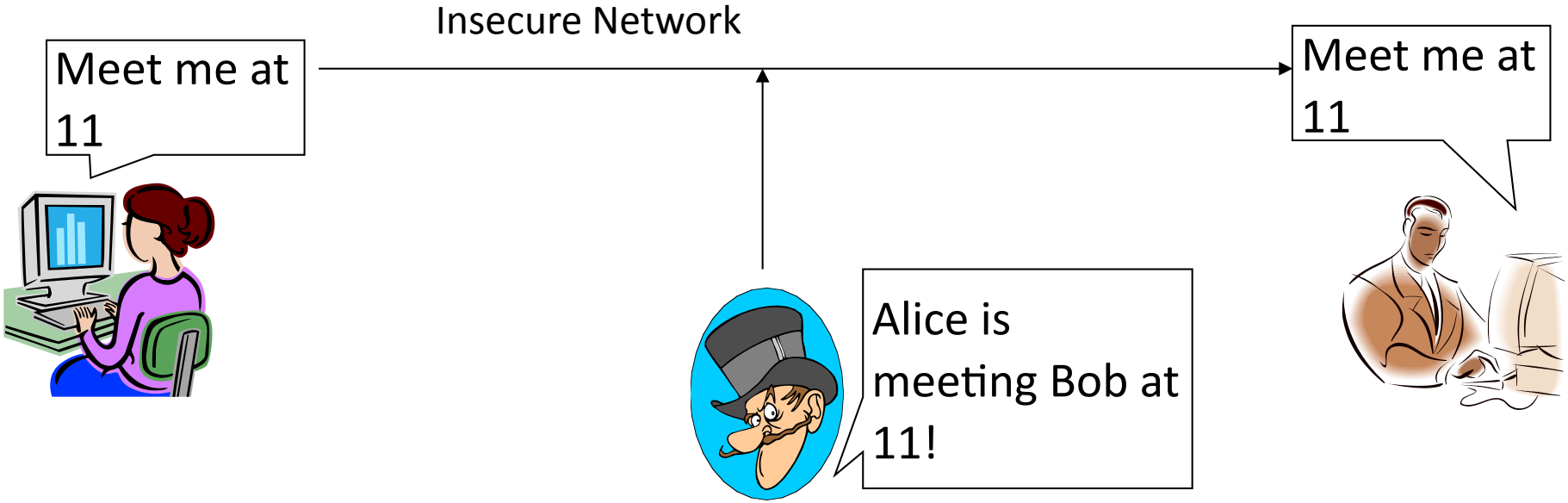
| | |
|---|---|
| Spoofing | illegally access and use another users's credentials |
| Tampering | maliciously change persistent data and the alteration of data in transit |
| Repudiation | perform illegal operations in system that doesn't track these operations properly |
| Information disclosure | ability to read data that you were not granted access to, or read data in transit |
| Denial of Service | prevent normal operation of a web service by making it temporarily unavailable or unusable |
| Elevation of privilege | gain privileged access to resources |

# Network attacks

- Assume someone can intercept network packets. What can they do?
- Passive:
  - eavesdrop on network traffic
- Active:
  - inject network packets
  - modify packets
  - reorder, replay packets
  - block packets

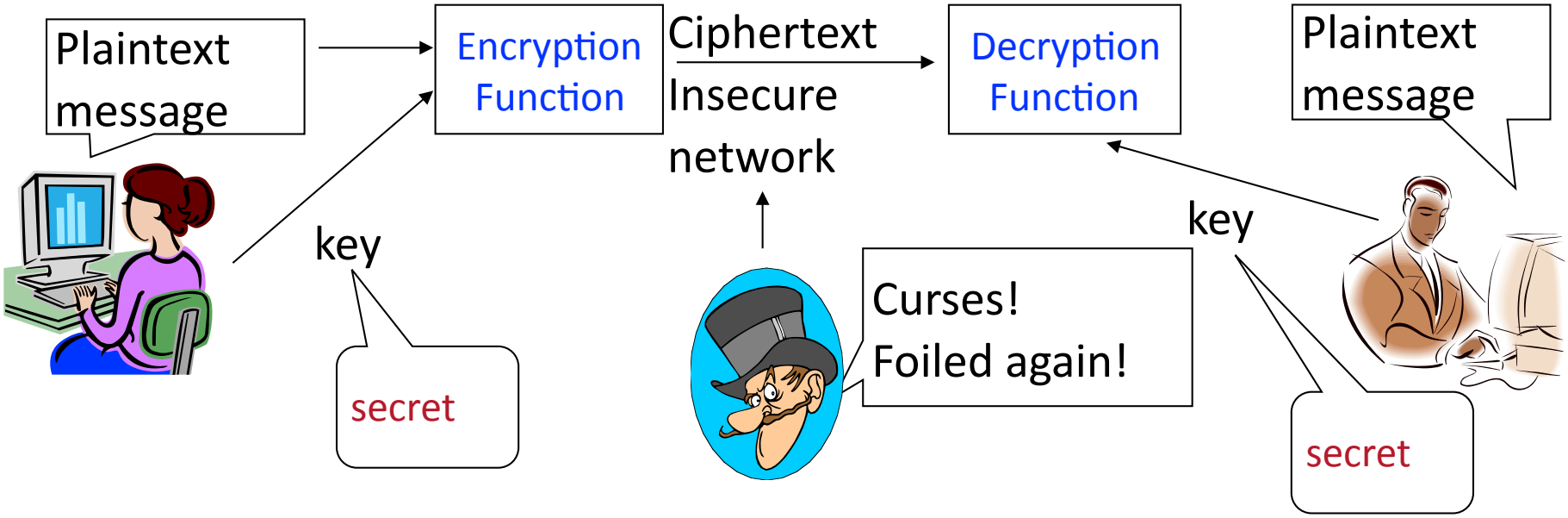
# Cryptography to the rescue

# Passive Network Attacks




- *Eavesdropping* - read network packets that were not intended for attacker
  - Does not interfere with delivery of message to intended recipient
  - Defense: obfuscate message contents so attacker gains no information from intercepting the message (i.e., encrypt the message)

# Encryption basics




Keys used for encryption and decryption may be same (symmetric) or different (public key)

# Diffie Hellman Merkle Key Exchange

- Function: g modulo p (where p is prime)
- Alice and Bob publicly agree to use
  - p = 23, g = 5
- Alice chooses secret 6
  - sends Bob $5^6$ mod 23 = 8
- Bob chooses secret 15
  - sends Alice $5^{15}$ mod 23 = 19
- Alice computes s = $19^6$ mod 23 = **2**
- Bob computes s = $8^{15}$ mod 23 = **2**

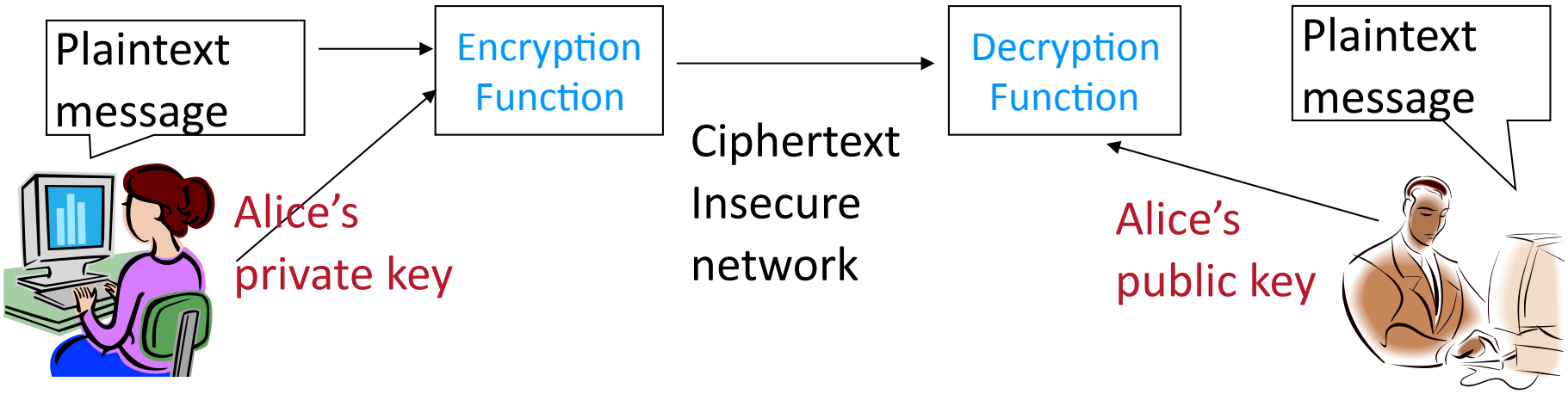
Eve can know p, g, 8, 19, and it is *hard* to compute 2

**2** is the secret symmetric key

# Encryption Keys

- Keys used for encryption and decryption can be the same (symmetric or secret-key) or different (public-key)
- Secret-key cryptography
  - Fast encryption/decryption algorithms are known
  - Distributing shared secret is a problem
- Public-key cryptography
  - Pairs of keys, one to encrypt, one to decrypt
  - Encryption key can be published, decryption key is kept secret; knowledge of one key does not reveal the other
  - Encryption/decryption algorithms ~1000x slower

# Message signing

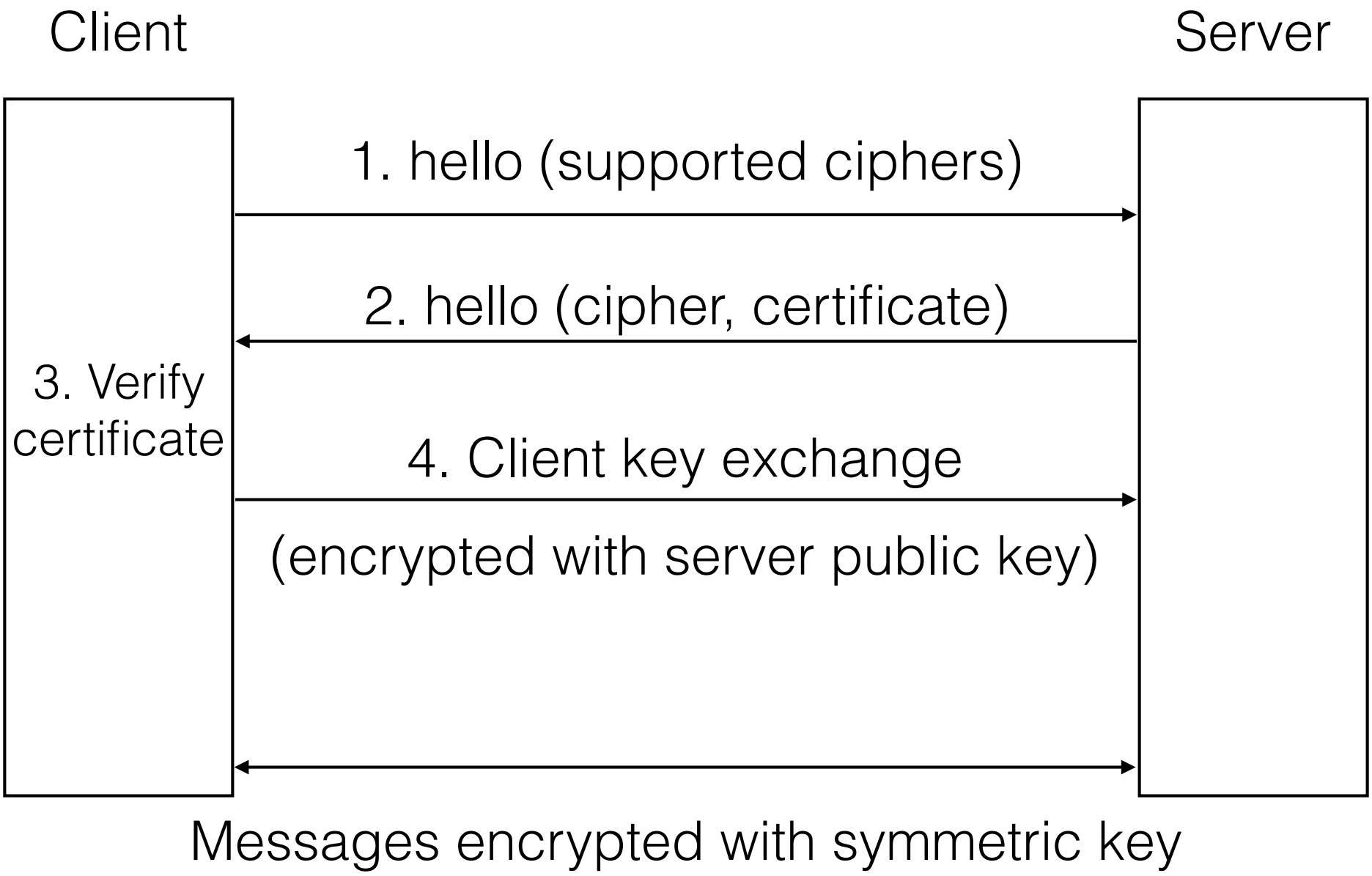



- Alice "signs" her message by appending a copy of her message encrypted with her private key.
- Bob uses the public key to decrypt the "signature" and can match the decrypted text to the plaintext, to make sure the text has not been tampered with.
- Bob can be sure that Alice sent the message.

# HTTP vs HTTPS

- HTTP packets are sent in plaintext
- HTTPS encrypts entire request.  Only destination address and port are visible.
- TLS (formerly SSL) is the protocol used for encryption, so initial handshake messages are not encrypted

# TLS (Transport Layer Security)



# TLS

- Different possible algorithms for generating session keys:
  - Public key:
    - Client encrypts random number using server's public key.
    - Server decrypts using its private key.
    - Now they have a shared key for symmetric encryption
  - Diffie-Helman

# Certificates

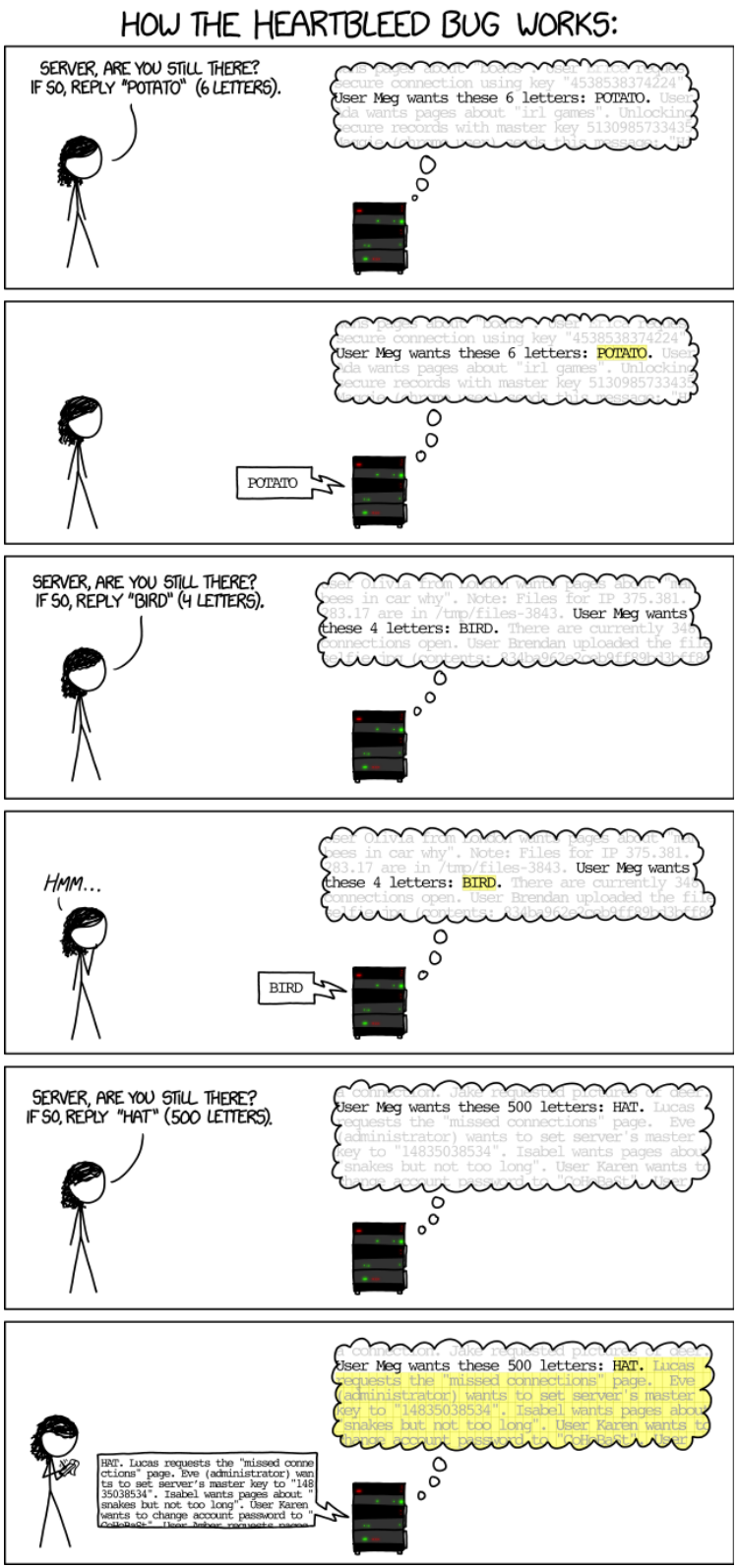
- Certificates are issued by a trusted CA (Certificate Authority)
  - Server requests a certificate and sends its public key
  - CA wraps public key in Certificate
  - client can check with CA to be sure server is who they say they are
- Self-signed certificate
  - created by the server
  - still can be used to encrypt traffic
  - but clients can't fully trust certificate (man-in-the middle attacks are possible.)

# Problem: TLS Stripping

- Common use pattern: user browses site with HTTP, upgrades to HTTPS for checkout.
- Active network attacker interposes on communication
- When server returns pages with HTTPS links, attacker changes them to HTTP.
- When browser follows those links, attacker intercepts requests, creates its own HTTPS connection to server, and forwards requests via that.
- As a result, the attacker sees all client packets (e.g., passwords).
- Browser provides feedback to user about whether HTTPS is in use, but most users won't notice the difference.

# Problem: “Just in time” HTTPS

- Login page displayed with HTTP
  - Form posted with HTTPS
    - Active attack corrupts login page and sends password elsewhere during form post
    - TLS stripping during form post - no evidence that actual connection did not use TLS
- Solution: before server returns HTML for login page, check for HTTPS. If page fetched by HTTP, redirect to HTTPS version




# Case Study

## Heartbleed

- A bug in the Heartbeat protocol for TLS (the successor to SSL)
- Heartbeat protocol: send a packet to server with an arbitrary payload. Server response tells client session is still alive.

- Bug: client sends a 2-byte payload_size. Server sends payload_size bytes back to client, even when payload_size is much larger than the payload sent by client.
- Client receives put to 64KB of memory from the server process that might include passwords, and other secure data.

http://xkcd.com/1354/

# What not to do…

- ignore security issues until later
- write your own cryptographic algorithm

# OWASP Top 10

1. Injection
2. Broken Authentication and Session Management
3. Cross Site Scripting (XSS)
4. Insecure Direct Object References
5. Security Misconfiguration
6. Sensitive Data Exposure
7. Missing function level access control
8. Cross Site Request Forgery (CSRF)
9. Using Components with Known Vulnerabilities
10. Unvalidated Redirects and Forwards

# 1. Injection

- SQL Injection
  - User inputs data that you want to store in the database
  - You insert that data into an SQL query
  - E.g.

```
String query = "SELECT * FROM accounts WHERE
custID='" + request.getParameter("id") + "'";
```

```
foo' or 'x'='x
```

```
String query = "SELECT * FROM accounts WHERE
custID=' foo' or 'x'='x ' ";
```

- returns all customer information

# Example

- Suppose we want to identify a user by their email address.
- Input field named “email”
- Using the same techniques as above, we can figure out parts of the database schema, and make an educated guess about the SQL query and the email address of a victim.

```
SELECT email, passwd, login_id, full_name
  FROM members
 WHERE email = 'x';
       UPDATE members
       SET email = 'steve@unixwiz.net'
       WHERE email = ‘bob@example.com';
```

- Now we can follow the normal password reset and hijack Bob’s account

# Real shell example

- Instructor writes shell script to checkout student repositories (**`checkoutrepos.sh`**)
- Each checkout needs instructor's password
- Instructor does not want to save his password in plaintext in script in his account, so he writes the script to take the password as command line argument.

```
$ ./checkoutrepos.sh fakepassword
$ ps aux | grep reid
reid@wolf:~$ ps aux |grep reid
```

Anyone with an account can run this.

```
reid     50029  0.0  0.0  22124  7280 pts/228  Ss   23:33   0:00 -bash
reid     50301  0.0  0.0   9520  2412 pts/227  S+   23:36   0:00 /bin/bash
./checkoutrepos.sh fakepassword
reid     50302  0.0  0.0   4348   640 pts/227  S+   23:36   0:00 sleep 120
reid     50325  0.0  0.0  15568  2252 pts/228  R+   23:36   0:00 ps aux
reid     50326  0.0  0.0   8868   768 pts/228  S+   23:36   0:00 grep reid
```

# Injection Prevention

- Use parameterized queries
- Escape everything
- Validate input
- Never trust raw input

# 2. Auth and Sessions

- Vulnerabilities
  - User credentials not stored using hashing or encryption
  - Credentials can be guessed or overwritten
  - Session IDs exposed in URL
  - Session IDs don’t time out
  - Authentication tokens (single sign-on) not properly invalidated on log out
  - Passwords, session IDs sent over unencrypted connections
- Recommendation: Use multi-factor authentication

# Passwords

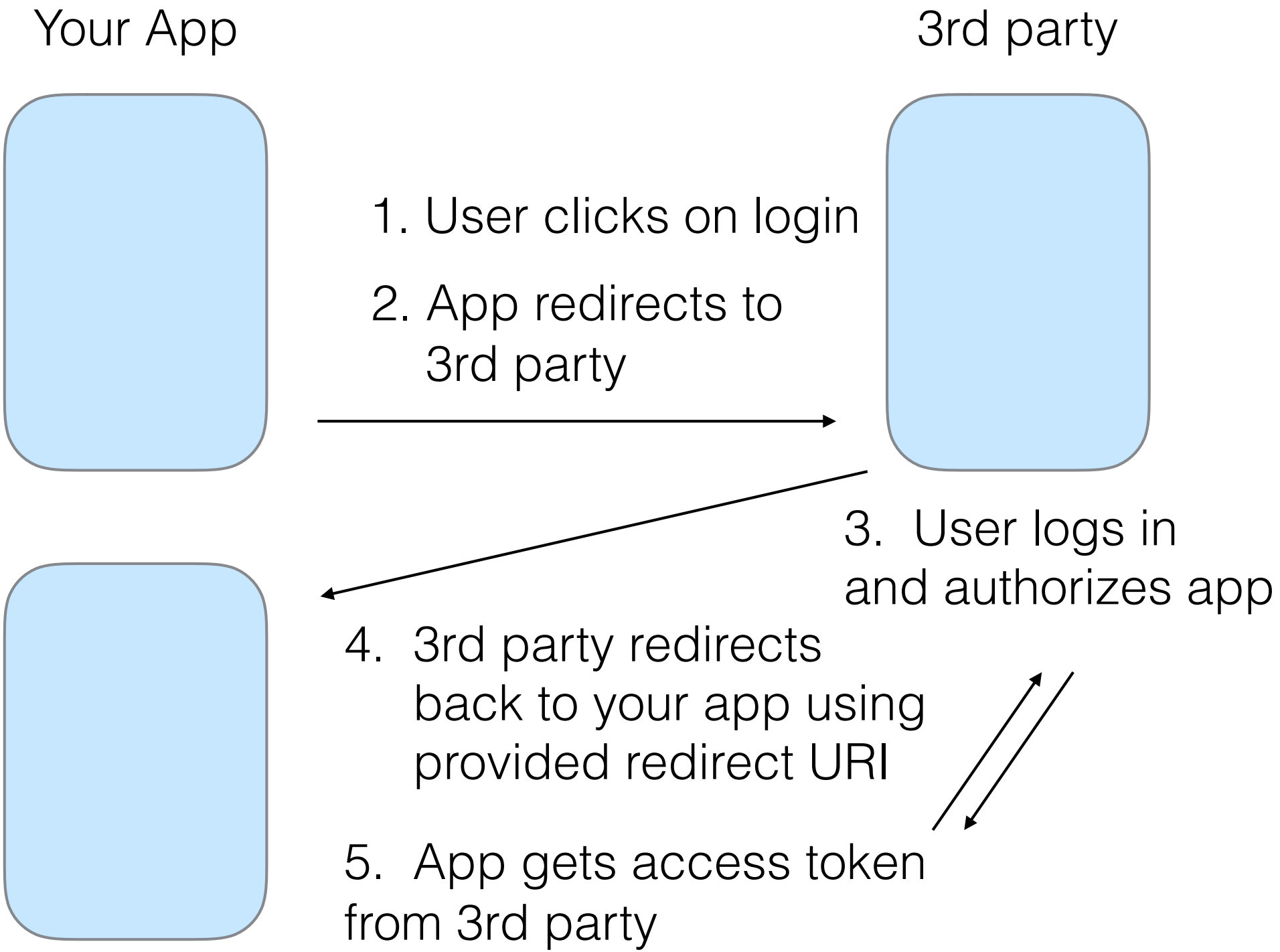
- Attacks
  - dictionary brute-force (too many people use obvious passwords)
  - rainbow tables - plaintext+encrypted
- Protections:
  - salt passwords (use known libraries

# Authentication Protection

- Require strong passwords
- Rate-limit number of attempts
- Implement account lockout (temporary)
- Log all login failures (and monitor logs)
- Secure password recovery features:
  - change password requires but does not reveal old password
  - never email a password in clear text

# OAuth

- Authorization framework that delegates user authentication to a service that hosts the user account
- Must register app with the auth service, including the URL to redirect to
- Auth service redirects to a registered URL with an authorization code  (using HTTPS of course)


Your App
3rd party
1. User clicks on login
2. App redirects to 3rd party
3. User logs in and authorizes app
4. 3rd party redirects back to your app using provided redirect URI
5. App gets access token from 3rd party

# 3. Sensitive Data Exposure

- Which data is sensitive enough to require extra protection?
  - passwords
  - credit cards
  - personal information
- Is any of this data ever transmitted in clear text?
- Encrypt all sensitive data

# Examples

- Example 1:
  - Automatic database encryption of passwords
  - This means it is also automatically decrypted
  - Passwords still vulnerable to SQL injection attack
- Example 2:
  - Application doesn't use TLS for all authenticated pages
  - Network packet snooping can read session cookies
  - Replay session cookies to hijack users's session

# 5. Broken Access Control

- Attacker is an authorized user
- Attacker can gain access to objects by guessing parameters

```
http://example.com/app/accountInfo?
acct=notmyacct
```

- Prevention: ensure user is authorized for access

# Broken Access Control

- Modifies URL, changes parameter and gets access to data attacker is not authorized for
- Often URL in frameworks refers directly to objects
  - Internal ids appear in URLs (REST)
- Check access authorization on every object
- Not sufficient to simply not show privileged operations to unprivileged users in the UI

# Examples

- Exposing internal ids:
  - www.example.com/profile/3032
- Using real file names:
  - www.example.com/reports?name=feb2016report.pdf
- Directory Traversal
  - http://example.com/%2e%2e/%2e%2e/%2e%2e/ %2e%2e/%2e%2e/%2e %2e/etc/passwd

# 6. Security misconfiguration

1. Is any of your software **out of date**? This includes the OS, Web/App Server, DBMS, applications, and all code libraries (see new A9).

2. Are any unnecessary features enabled or installed (e.g., ports, services, pages, accounts, privileges)?

3. Are **default accounts** and their **passwords** still enabled and unchanged?

4. Does your error handling reveal **stack traces** or other overly informative error messages to users?

5. Are the **security settings** in your development frameworks (e.g., Struts, Spring, ASP.NET) and libraries not set to secure values?

6. *Without a concerted, repeatable application security configuration process, systems are at a higher risk.*

# 7. Cross-Site Scripting

- Data enters web application and is included in dynamic content sent to a web user without being validated for malicious content.
- Usually JavaScript but may also include HTML and other data
- Three types
  - Stored
    - injected script is stored permanently on server
    - victim retrieves script from server through normal requests
  - Reflected
    - victim is conned into clicking on a link containing malicious script code that will be executed by browser thanks to vulnerability in app
  - DOM-based
    - Modifies DOM, but don't see the vulnerability in request or response.
- Escape all untrusted data

# example

DOM-based vulnerability:

```
<script>
    document.write("<b>Current URL<b> :" +
                   document.baseURI);
</script>
```

send the following HTTP

```
http://www.example.com/test.html#<script>alert(1)</script>
```

# 9. Using components with know security vulnerabilities

- Development teams and Dev Ops people must be vigilant
- Be careful what you include in your application
- Update software always

# X. Cross-Site Request Forgery

- Some content on unrelated site includes a POST to your application
- If a user of your app navigates to compromised site while logged into your app, the malicious POST request can pretend to be the user, and steal the user's info from your app
- Prevention: Include an unpredictable token with each HTTP request (usually in a hidden field)
- Edged out of the top 10 because frameworks are better at prevention

# Example

```
<form method="POST" action="http://example.com/admin/
changeRole" target="iframeHidden">

<h1> You are about to win a brand new iPhone!</h1> <h2>
Click on the win button to claim it...</h2>

<input type="hidden" name="accountId" value="67887"/>

<input type="hidden" name="newRoleId" value="1"/>

<input type="submit" value="Win !!!"/> </form>

<iframe name="iframeHidden" width="1" height="1"/>
```

Trick admin user to click on this button. Admin can escalate someone's role, so sets this user to privileged role. Attack is possible because cookie are sent with this request.

http://cdn.oreillystatic.com/pdf/securing-node-applications.pdf

“Arguing that you don’t care about the right to privacy because you have nothing to hide is no different than saying you don’t care about free speech because you have nothing to say.”

–Edward Snowden

Go watch
“Citizen Four”

“If you think technology can solve your security problems, then you don’t understand the problems and you don’t understand the technology.”

–Bruce Schneier

“The only truly secure system is one that is powered off, cast in a block of concrete and sealed in a lead-lined room with armed guards.”

– Gene Spafford

Feeling good about the world?