

CS 188

Scalable Internet Services

Andrew Mutz

November 1, 2016



Announcements

Teams are now transitioning from building features to load testing.

This Friday we will have full-class demos in lab. Demos will be on AWS. Demo your features and functionality to the class.



Announcements

Please use -n when running Tsung

- `tsung -n -f simple.xml start`

Turn off CSRF protection

```
class ApplicationController < ActionController::Base
```

```
  # protect_from_forgery with: :exception
```

```
end
```



Motivation

Internet services increasingly mediate more and more interactions in modern society

- I want to buy dinner.
- I want to save important documents.
- I want to manage my investments.

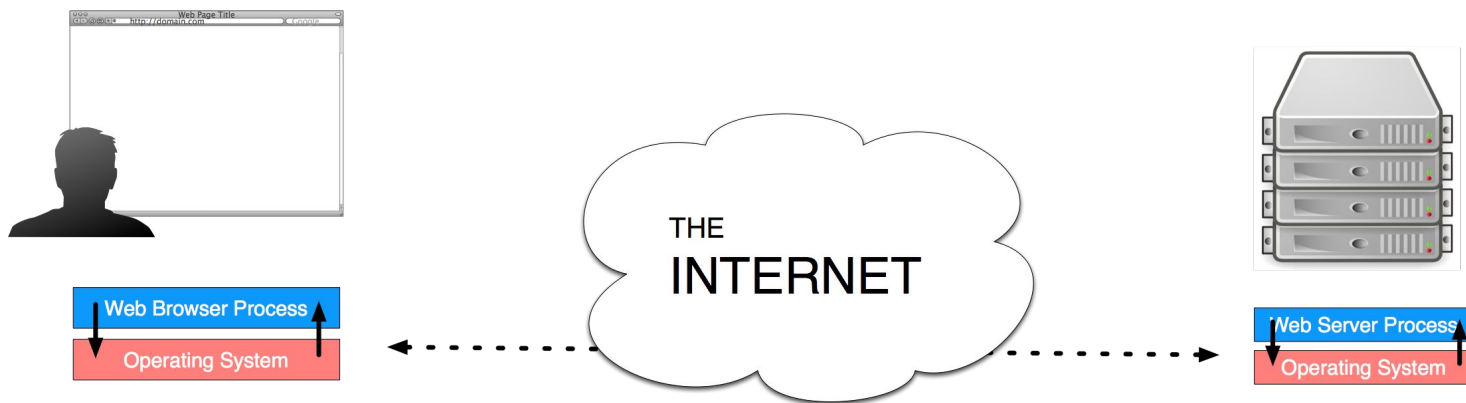
Every day, billions of people use the same suite of technologies to solve these problems.

How do we keep these interactions secure?



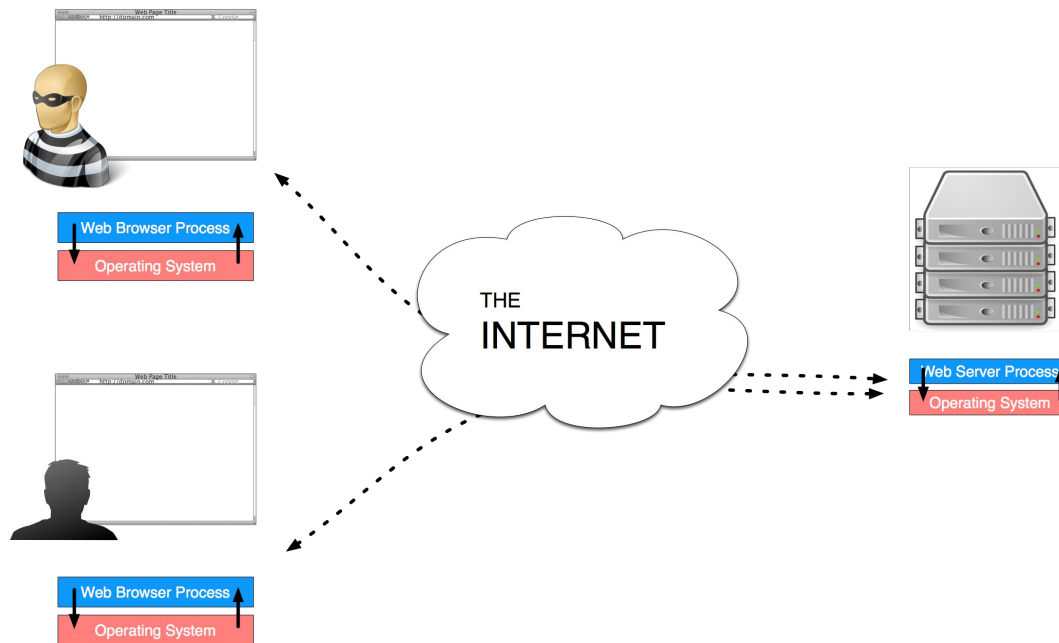
Motivation

What sort of bad actors can we see in the system?



Motivation

What sort of bad actors can we see in the system?

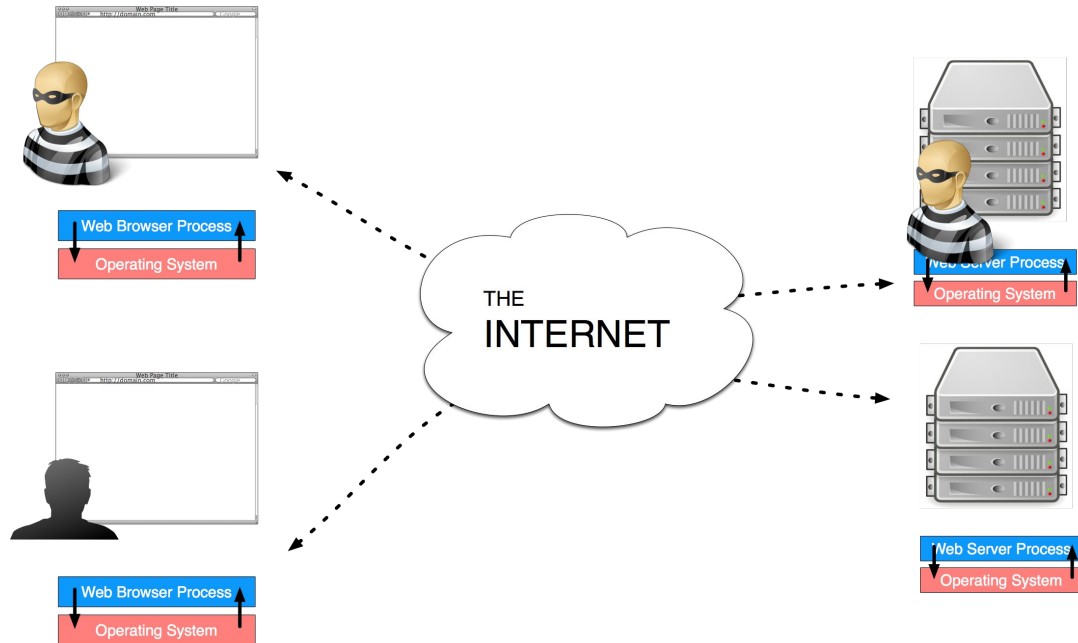


Bad clients?



Motivation

What sort of bad actors can we see in the system?

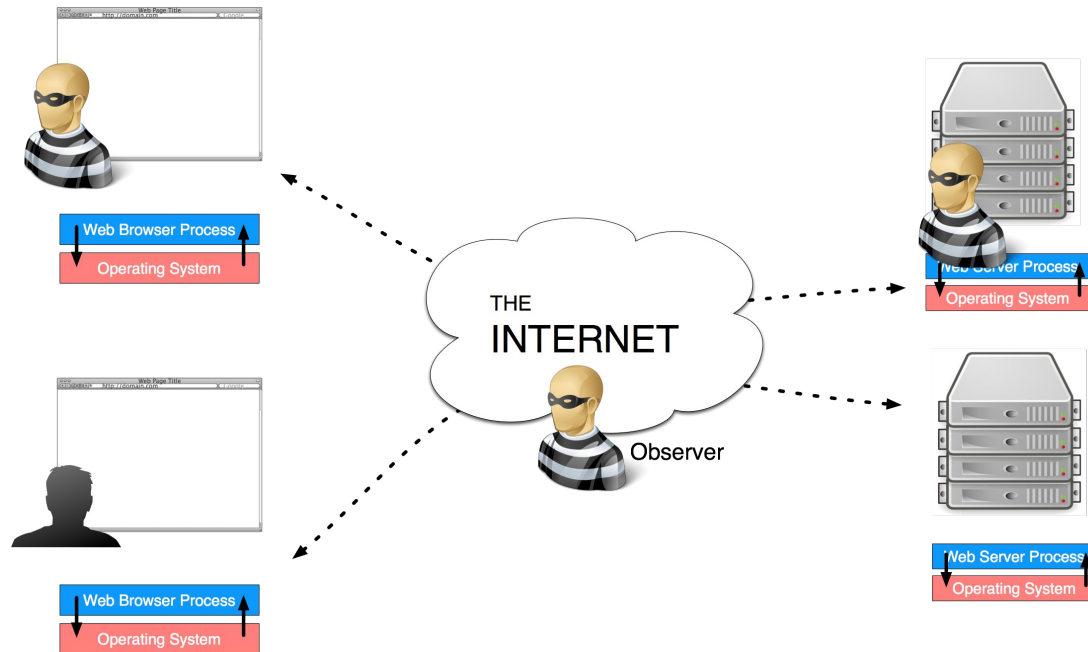


Bad servers?



Motivation

What sort of bad actors can we see in the system?

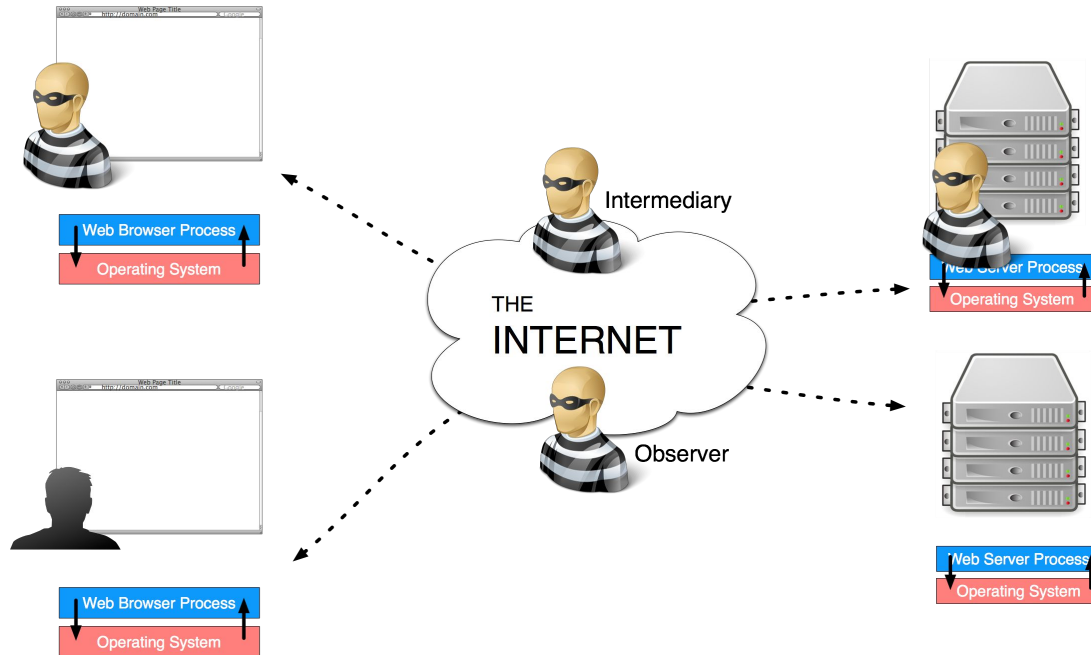


Bad middle-men
who can snoop?



Motivation

What sort of bad actors can we see in the system?

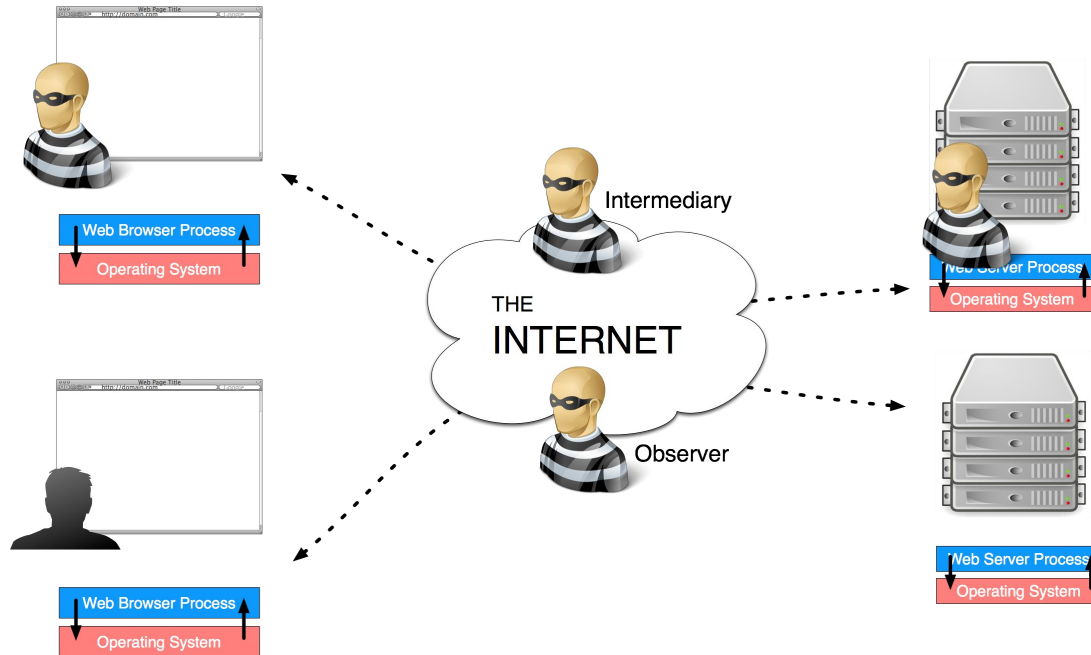


Bad middle-men
who can modify
traffic?



Motivation

What sort of bad actors can we see in the system?



All of the above.



Goals

What do we want?

- **Privacy**

- My private data can not be read by third parties.

- **Authentication**

- The client knows it's talking to the right server, and the server knows it's talking to the right client.

- **Integrity**

- Data (at rest or in flight) can't be tampered with.



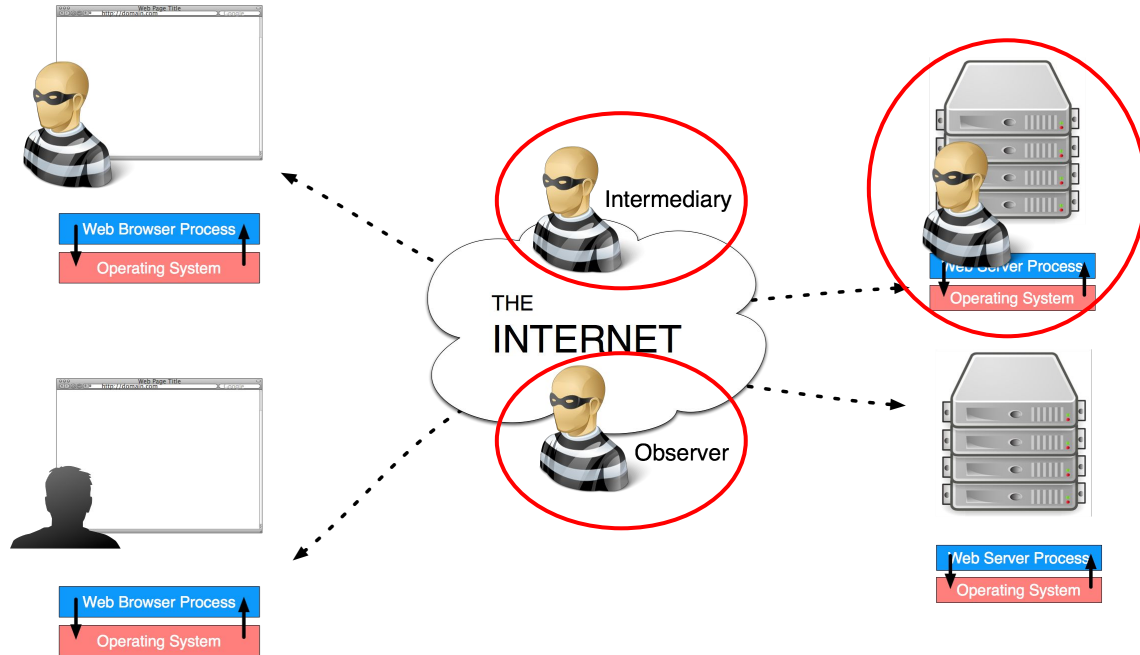
Today's Agenda

Web Security Basics

- HTTPS
- Firewalls
- SQL Injection
- Cross-Site Scripting
- Cross-Site Request Forgery



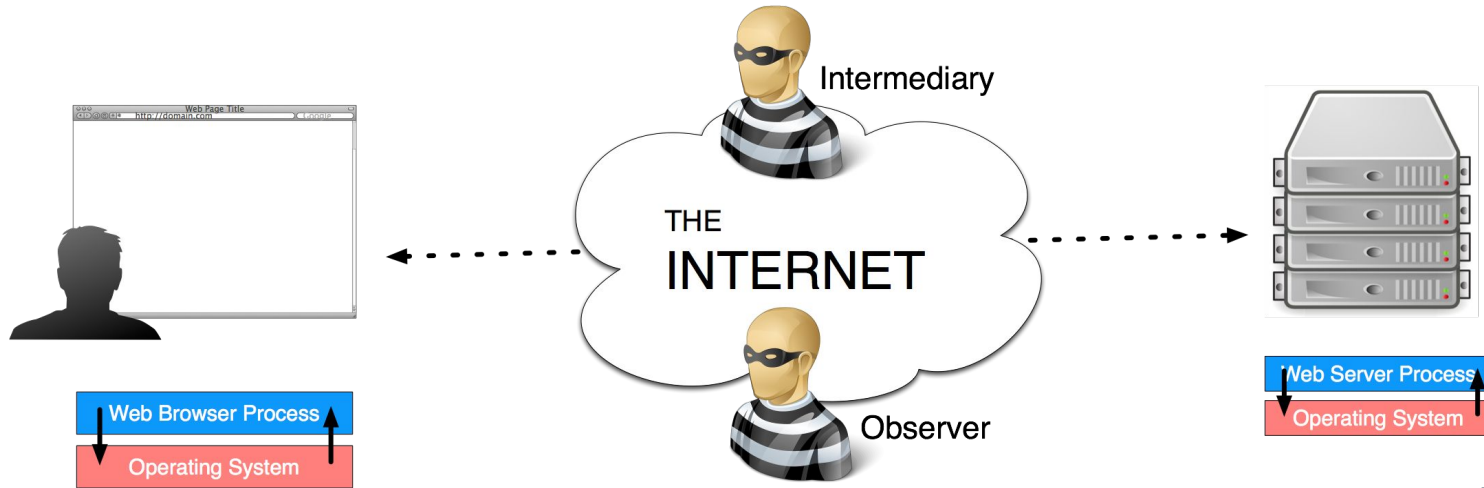
HTTPS



HTTPS is designed to protect us against malicious intermediaries and malicious servers.



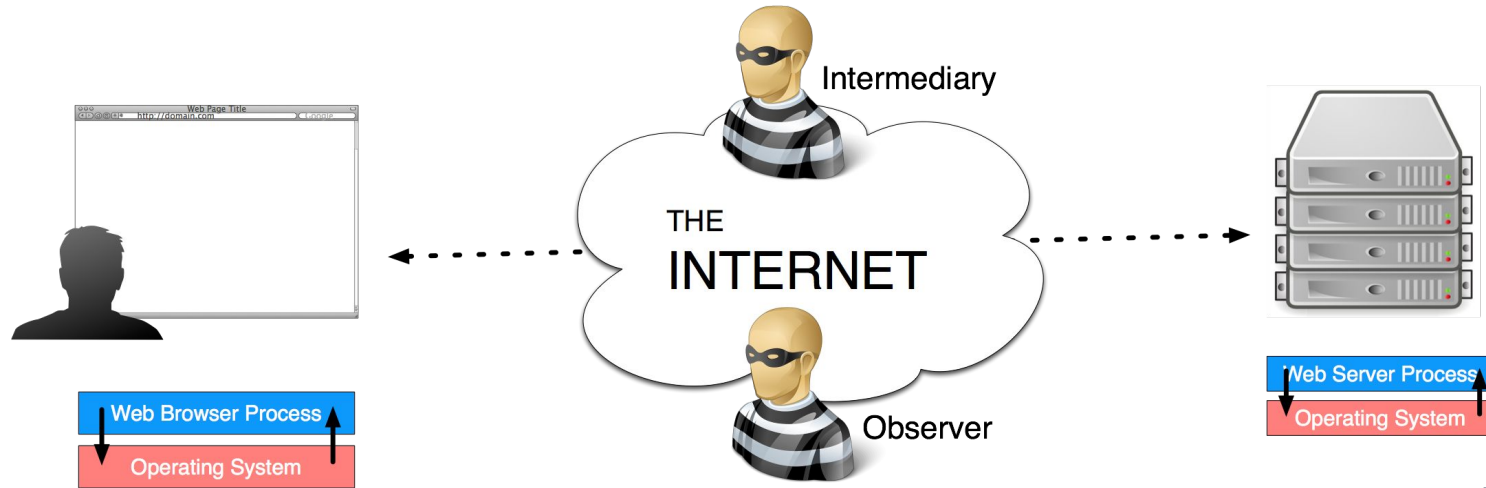
HTTPS



Building our application on TCP gives us an initial start at security on the internet. **How does TCP protect us against one of the above bad actors?**



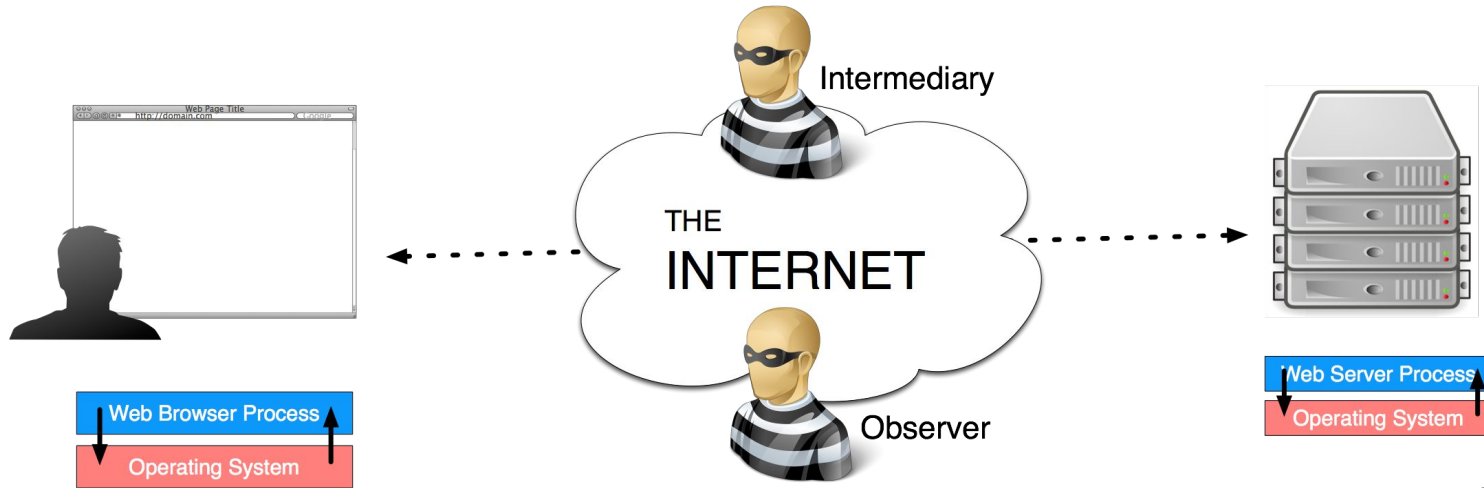
HTTPS



TCP sequence numbers mean that an observer of our HTTP traffic can read everything, but can't really tamper with our session. An intermediary, on the other hand, has complete control.



HTTPS



Also, because we tend to use cookies for session management, an observer can issue additional requests on our behalf

Conclusion: TCP alone doesn't protect much.



HTTPS - Goals

What do we want from a secure sockets layer?

- **Privacy**

- My communications should be hidden from the observers of my traffic

- **Integrity**

- My communications should not be tampered with by intermediaries

- **Authentication**

- I can be sure I am speaking to the intended server, and not a malicious third-party.



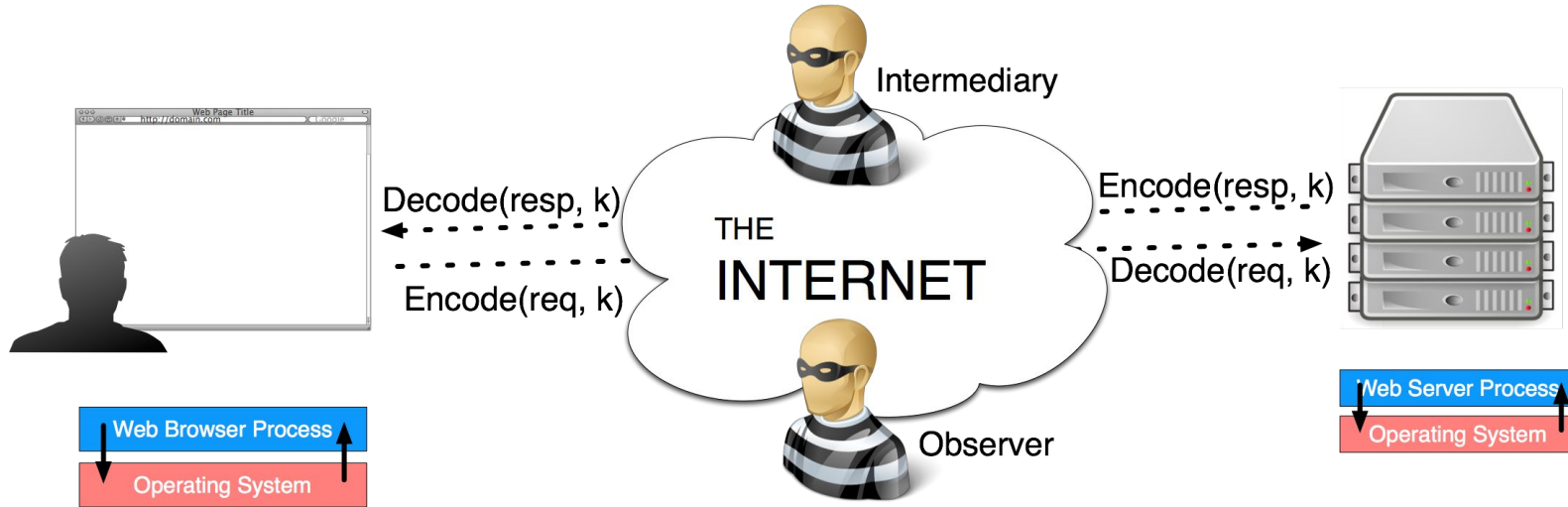
HTTPS

One slide introduction to cryptography:

- Symmetric Cryptographic Algorithm
 - A function that can encode and decode data using a key, k :
 - $\text{encode}(p, k) = c, \text{decode}(c, k) = p$
 - Common implementations: **AES**, DES, Threefish
- Asymmetric Cryptography
 - A pair of keys, one public (k_1) and one private (k_2).
 - $\text{encode}(p, k_1) = c, \text{decode}(c, k_2) = p$
 - $\text{encode}(p, k_2) = c, \text{decode}(c, k_1) = p$
 - By keeping k_2 private, encoding with k_2 is called “signing”.
 - Common implementations: **RSA**, DSA, Diffie-Hellman*



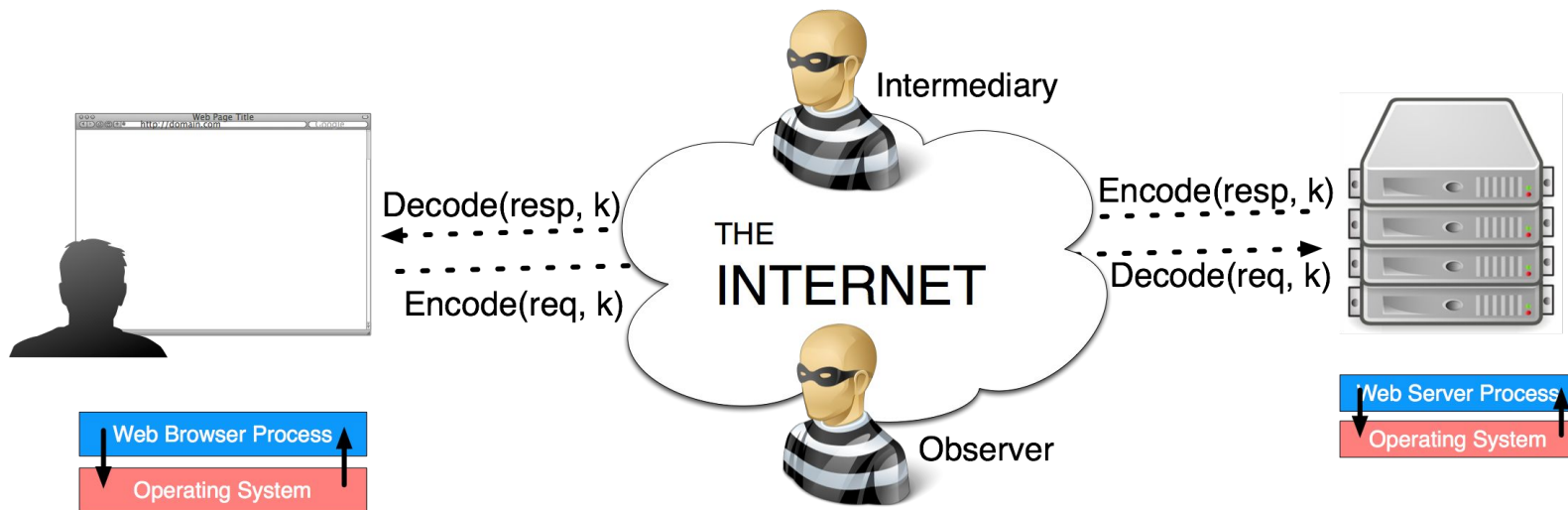
HTTPS



How far can symmetric cryptography alone get us?



HTTPS



How far can symmetric crypto alone get us?

- If we have a shared key: Privacy, Integrity & Authentication
- **But how do we get a shared key?**



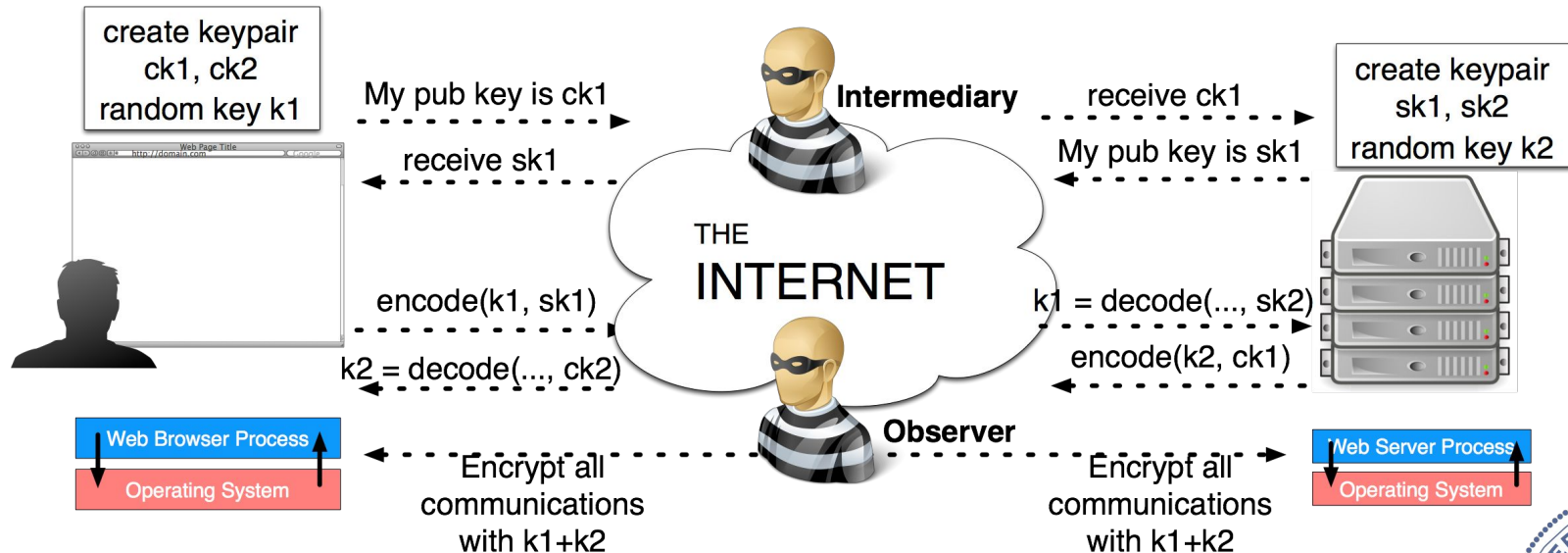
HTTPS

We want the web to work with combinations of arbitrary clients and servers, so there are no *a priori* shared secrets.

How do we establish a shared symmetric key without intermediates knowing it?



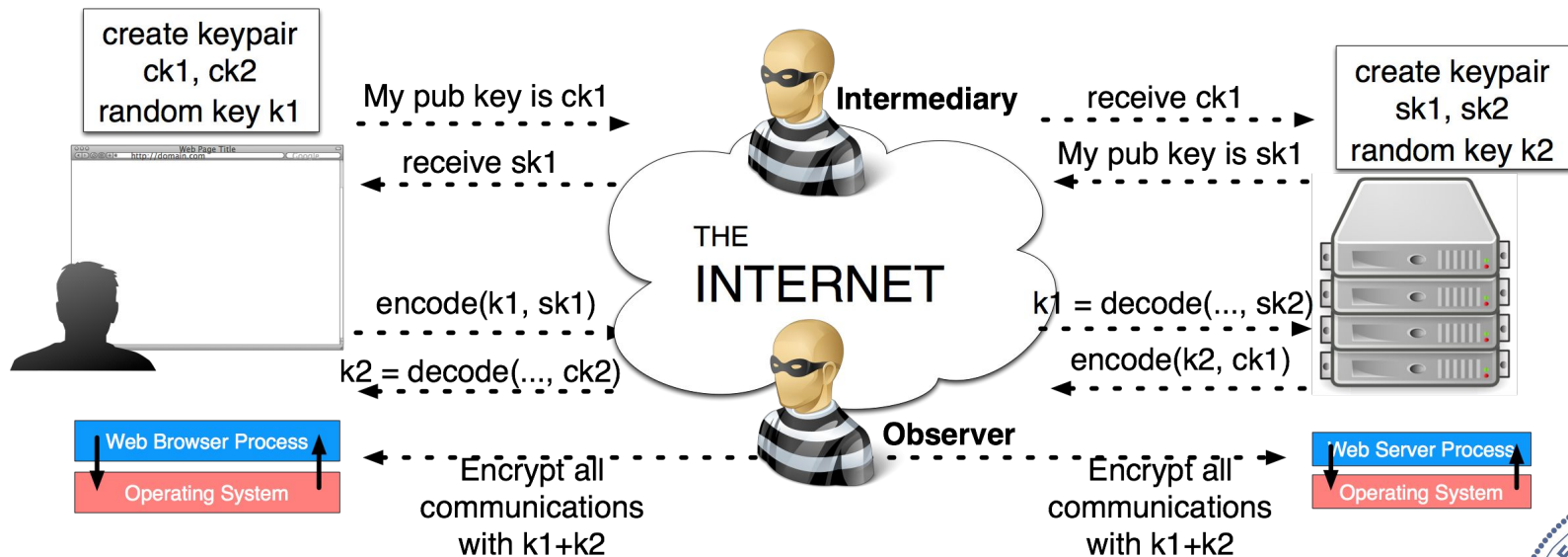
HTTPS



Let's use asymmetric cryptography to establish a shared session key.



HTTPS



This works against the observer, but not the intermediary.
Why?

HTTPS

Man in the middle attack:

- Intermediary creates his own keypair.
- Presents his public key to the server as the client's key
- Presents his public key to the client as the server's key
- Establishes one shared key with the client and another with the server.
- Can shuttle requests and responses back and forth, inspecting and modifying with complete control.



HTTPS

How can we prevent this?



HTTPS

How can we prevent this?

- If the browser knew everyone's public key *a priori*, the man in the middle attack wouldn't work.
 - But there are too many servers, and new ones go up all the time.
- Hint: keys can sign other keys...



HTTPS - Certificates

Insight: we can use private key signatures to transitively authenticate someone. Example:

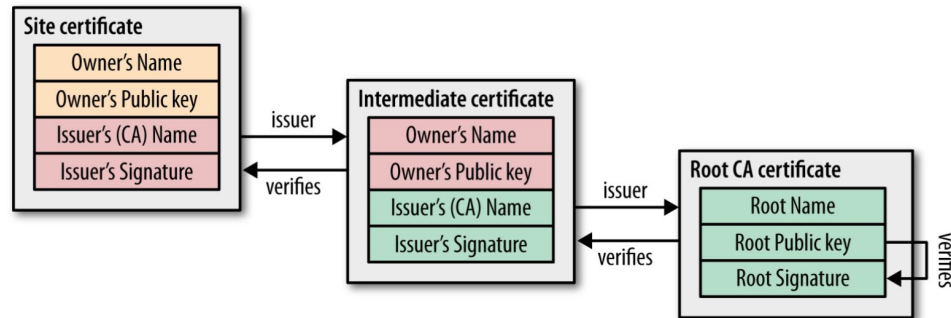
- Alice trusts Bob and has his public key
- Bob knows Charlie and has his public key
- Charlie wants to talk to Alice and presents his public key
- Alice doesn't know if this is really Charlie, or someone else who has handed over their own private key.
- In order to solve this, Charlie has Bob write down "Charlie's public key is [...]" and sign it with his private key.
- Because Alice trusts Bob, she knows Charlie is legitimate when he provides the Bob-signed document.
- **This document is known as a certificate.**



HTTPS - Certificates

Solution: Browser maintains a small list of trusted Certificate Authorities (CAs)

- Any website can present a certificate issued by a CA and the browser can trust that the party is legitimate
- Certificates can be chained.
 - “Root” CA vs. “intermediate” CA



HTTPS

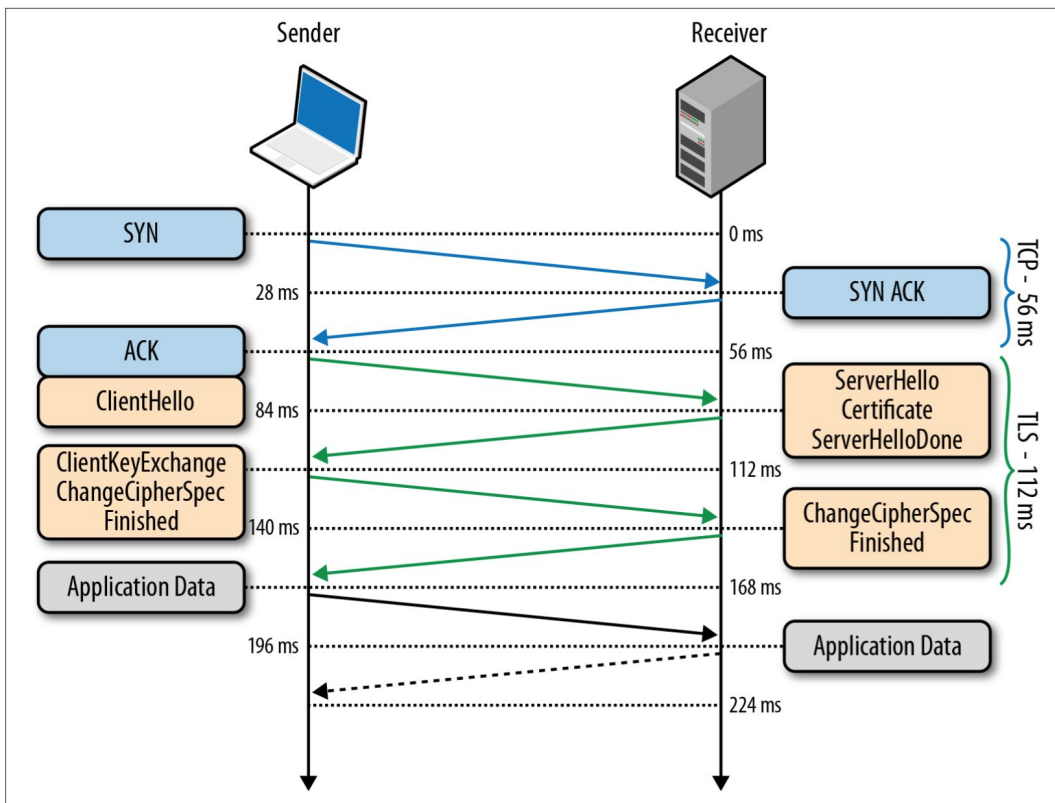
Summary:

- **Certificates** are used to verify the identity of the server
- **Asymmetric cryptography** is used to establish a shared key.
- Once a shared key has been established, **symmetric cryptography** is used for the session.

TLS allows combinations of different cipher suites to be used.



HTTPS - SSL Handshake



After TCP setup:

1. Client initiates SSL handshake with a list of CipherSuites and a random number
2. Server responds with its cert, selected CipherSuite and a random number
3. Client creates session key based on exchanged randomness, encrypts with server's public key
4. Both sides switch to using symmetric key.

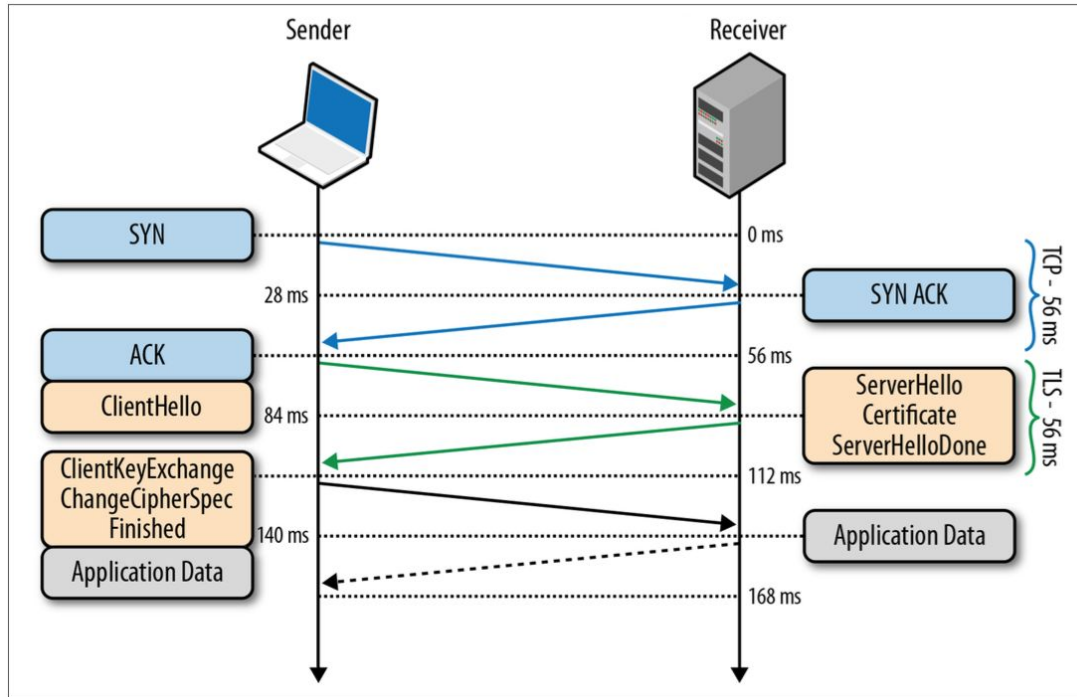
HTTPS - SSL Handshake

Latency involved in two extra round trips is expensive.

For repeated access by a client we can speed this up...



HTTPS - Abbreviated SSL Handshake



After TCP setup:

1. Session ID is added to connections with new hosts
2. Client initiates SSL handshake with previously-used Session ID
3. Server acknowledges previously used Session ID
4. Each side computes session key based on the remembered random numbers.
5. Both sides switch to using symmetric key.

HTTPS - Goals

Have we achieved our goals?

- **Privacy**
 - Because no other party can know the three random numbers with which we generated the session key, our data can not be read by intermediaries
- **Integrity**
 - Each TLS frame includes a Message Authentication Code (MAC) that prevents tampering.
- **Authentication**
 - Because the public key used by the server has an associated certificate, and because I trust the CA, I know this is the intended party



HTTPS - Certificates

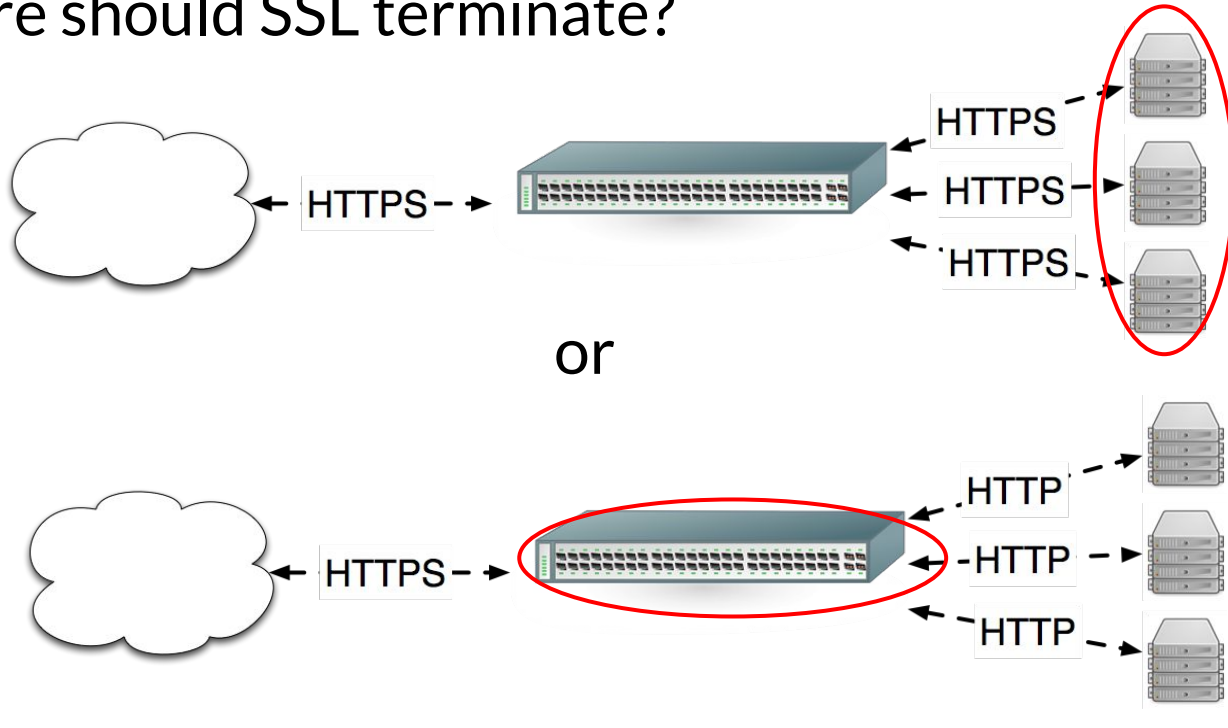
Certificates can be revoked.

- Why might we need to revoke a certificate?
 - Private key compromised
 - Intermediate CA was compromised
- Two main mechanisms for certificate revocation
 - Certificate Revocation List
 - Periodically get a list of all revoked certificates. If a connection is attempted with a revoked certificate, do not allow it.
 - Online Certificate Status Protocol (OCSP)
 - At request time, query the CA to check if revoked.
 - Advantages of each?



HTTPS

Where should SSL terminate?



HTTPS Termination

Advantages of terminating SSL at the load balancer

- If each app server maintained its own session cache, it would frequently miss.
- Load balancer can be built with hardware acceleration for TLS handshake.
- Load balancer can see inside each request, potentially improving its balancing decisions.
- Private key just sits in one place

Advantages of terminating SSL at the App Server

- Data between load balancer and app servers stays private



HTTPS Strict Transport Security

HTTPS gives us very good security for the web, but users aren't always aware of it.

- Lock icon can be subtle

Strict Transport Security gives us the option of telling a browser “for all future communications, use HTTPS”.

- Response sent HTTP Header
- Strict-Transport-Security: max-age=31536000



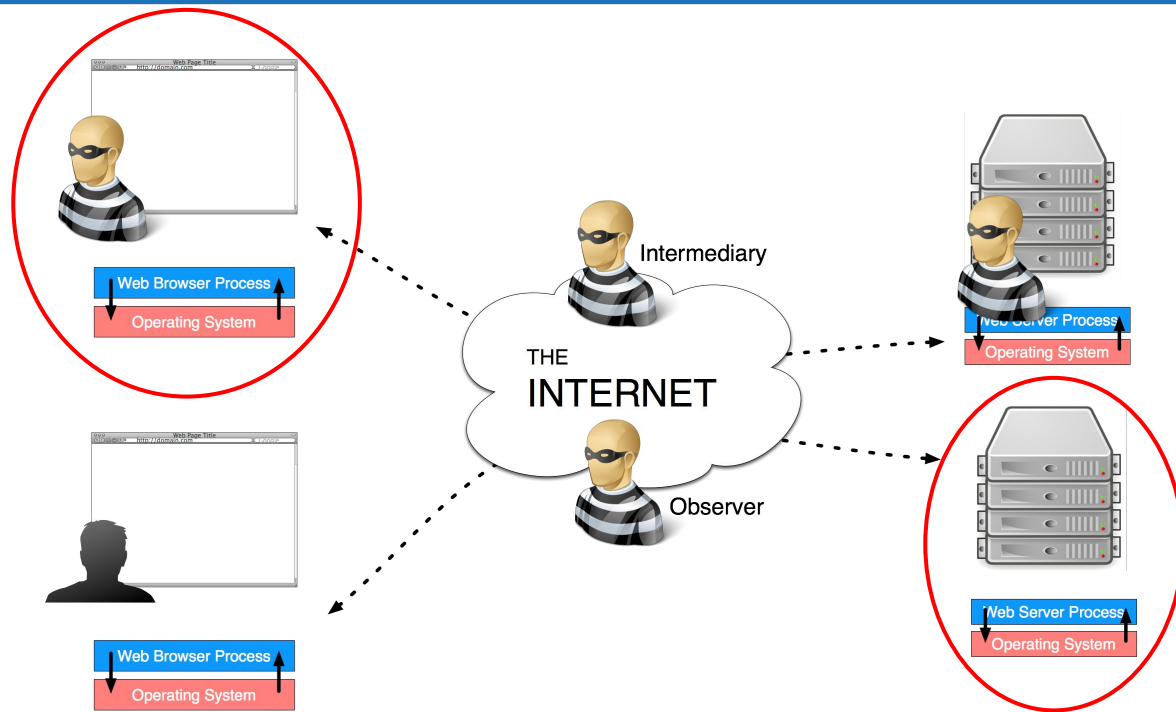
Three common attacks

Next, we will look at three common security vulnerabilities on the web, and how to mitigate them:

- **SQL injection**
 - Getting a database to execute bad SQL
- **Cross-site scripting**
 - Getting a browser to execute bad javascript
- **Cross-site Request Forgery**
 - Getting a browser to submit bad data



SQL Injection



SQL Injection

A SQL injection attack is when a malicious user submits a carefully crafted HTTP request that causes your app server to interact with the database in a manner you did not intend.



SQL Injection

How could we manipulate this code?

```
def create
  user_id = params['user_id']
  comment_text = params['comment_text']
  sql = <<-SQL
    INSERT INTO comments
      user_id=#{user_id},
      comment=#{comment_text}
  SQL
  ActiveRecord::Base.connection.execute(sql)
end
```



SQL Injection

```
def create
  user_id = params['user_id']
  comment_text = params['comment_text']
  sql = <<-SQL
    INSERT INTO comments
    user_id=#{user_id},
    comment='#{comment_text}'
  SQL
  AR::Base.connection.execute(sql)
end
```

What if a user submitted the following parameters?

```
user_id=5
comment_text='; UPDATE users SET
admin=1 where user_id=5 and '1'='1
```



SQL Injection

```
sql = <<-SQL
  INSERT INTO comments
  user_id=#{user_id},
  comment='#{comment_text}'
SQL
```



```
INSERT INTO comments
  user_id=#{user_id},
  comment=''; UPDATE users SET
admin=1 where user_id=5 and '1'='1'
```



SQL Injection

How do we mitigate this?

- Never insert user input directly into a SQL statement without sanitizing it first.
- Rails does a lot of the work here for you:
 - Access through the AR ORM is safe
 - If you need to sanitize custom sql, you can use the sanitize method:

```
SELECT *
```

```
FROM comments
```

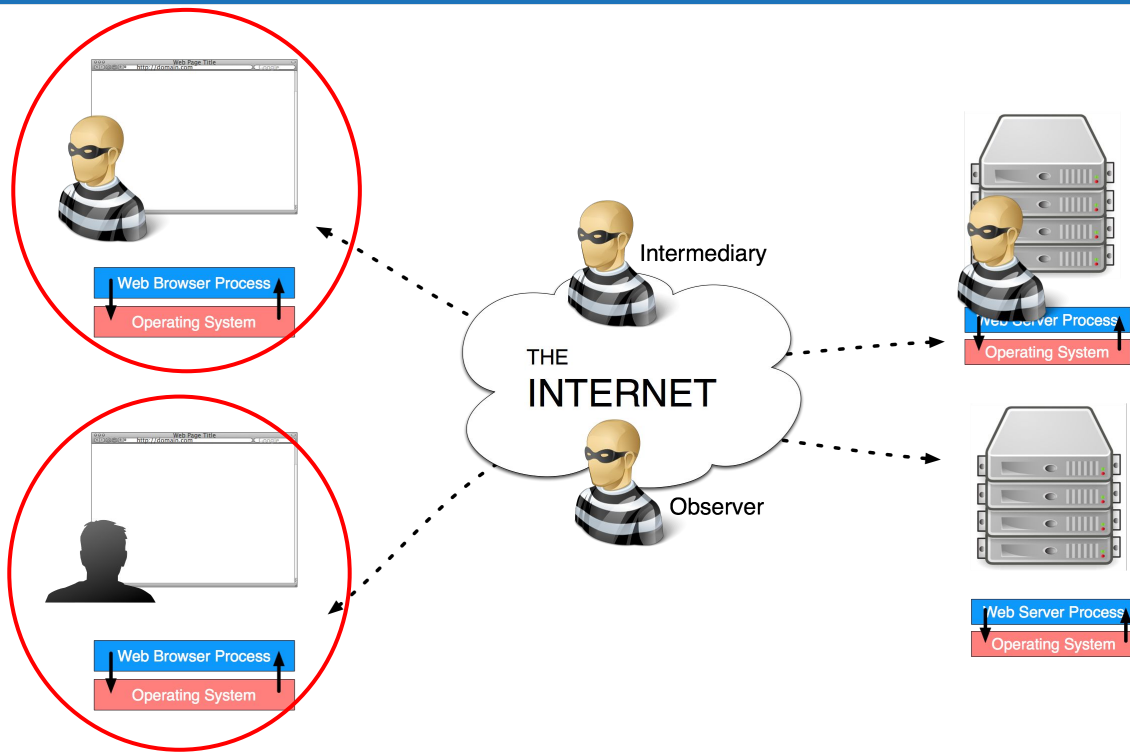
```
WHERE id=#{Comment.sanitize(params[:id])}}
```



SQL Injection



Cross-site scripting



Cross-site Scripting

Javascript is powerful:

- If I can get another user to execute arbitrary javascript in the context of another page, it can issue arbitrary HTTP requests to the server.
- Ajax requests are limited to the domain that the JS originated from, but requests back to that domain will include all relevant cookies.
- For example, if I can execute arbitrary JavaScript in the browser that is running <http://wellsfargo.com>, all Ajax requests will occur with the user's current cookies (and session)



Cross-site Scripting

Basics of a Cross-site Scripting (XSS) attack:

- One user submits data that will be displayed to other users of the web application.
 - User comments are a common example
- When another user visits the page, the server includes this data as part of the body of the webpage.
- When the victim's browser encounters this data, it executes it in the same way it executes all javascript that came from the server



Cross-site Scripting

Example:

New submission

Title

Url

Community

Create Submission

[Back](#)

[Log In](#) | [Sign Up](#)

A blue arrow pointing from the 'New submission' form to the submission details.

Submission was successfully created.

Title:

Url: <http://cs290.com>

Community: [Programming](#)

Comments

[Edit](#) | [Back](#)

[Log In](#) | [Sign Up](#)

A Chrome browser alert dialog box. It has the Chrome logo on the left. The text inside says "The page at localhost:3000 says:" followed by "oops" on the next line. There is an "OK" button in the bottom right corner.

The official seal of the University of California, Los Angeles (UCLA). It is a circular seal with "THE UNIVERSITY OF CALIFORNIA" around the top and "UCLA" at the bottom. In the center is a shield with an open book and a star, with the motto "LET THERE BE LIGHT" below it.

Cross-site Scripting

Because the application developer is just redisplaying the data submitted by one user into the DOM of another user, one user can do anything that the other, logged-in user can do.

- In an email website, it could send emails on your behalf, or read your mail
- On a bank website, it could potentially transfer money to an attacker

How might we prevent this?



Cross-site Scripting

Sanitize the data that you are displaying to the user.

Rails does this for you. Example:

If I enter form data as:

```
<script>alert("oops");</script>
```

Rails will save it to the database as exactly that.

But when I go to display it to the user:

```
<%= submission.title %>
```

Rails takes the title (exactly as above) and automatically converts it to
<script>alert("oops.");</script>



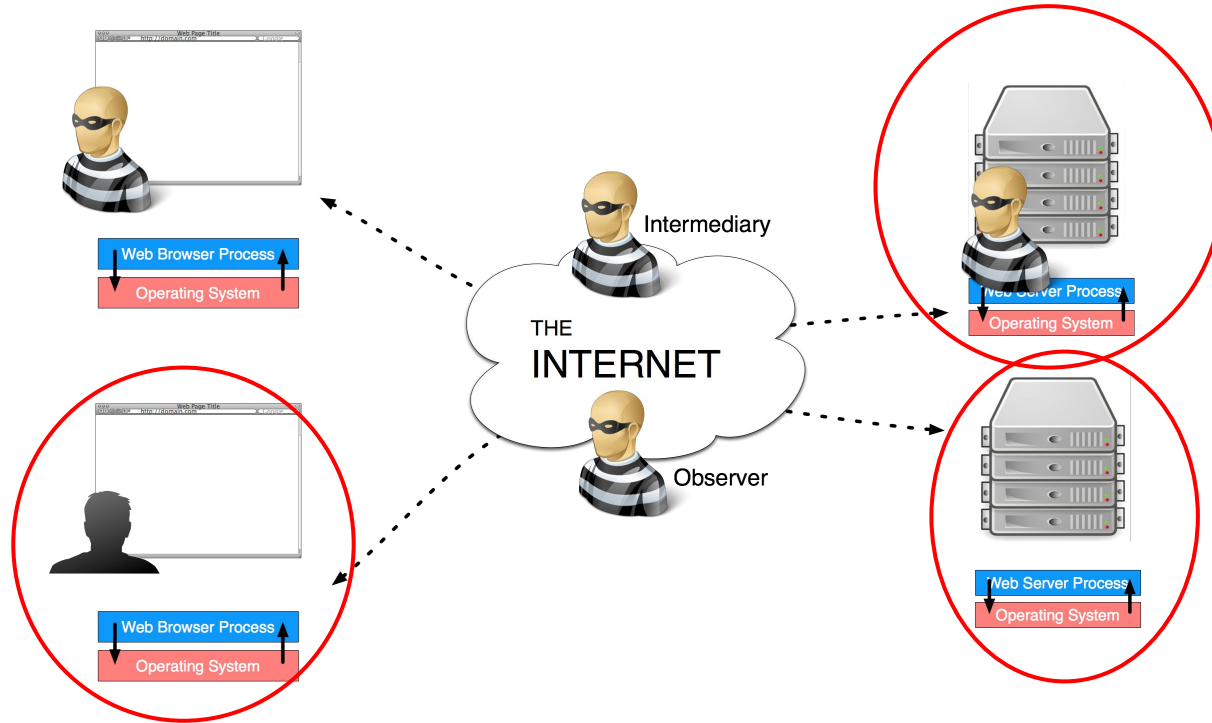
SQL Injection & XSS prevention

How do we make sure we aren't making injection errors?

- Fuzzing: provide semi-random input to the application and watch for errors
- Tarantula: An automated tester that crawls your application and fuzzes for injection errors:
 - <https://github.com/relevance/tarantula>



Cross-site Request Forgery



Cross-site Request Forgery

Ingredients for this attack:

- One maliciously-controlled server
- One benign web server
- One well-meaning client who happens to visit the maliciously-controlled server.



Cross-site Request Forgery

Ajax requests can't cross domains, but some requests can:

- I can set up a form on domain1.com to POST to domain2.com.
- Alternately, I can use an img tag on domain1.com to GET a resource on domain2.com that has side-effects
- I can't do it with Ajax, so I can't see the result, but it can still do damage.
- Like with XSS, All session cookies, etc. will be transmitted to the server



Cross-site Request Forgery

<https://evil.com>

Check this image out:

```

```

When viewed, the image tag attempts to load, performing HTTP GET with session cookies



Cross-site Request Forgery


`https://evil.com`

Sign up for a fun service!

Email:

`https://wellsfargo.com/txn/new`

Your bank transfer is complete!


Actually submits to



Cross-site Request Forgery

How do we mitigate this?

- First, make proper use of HTTP semantics. GETs shouldn't have side effects.
- For form POSTing, a common technique is to add a random token as a hidden field in each form rendered.
 - If the form is submitted without the token, the request is denied.



Cross-site Request Forgery

How does this look in Rails?

```
class ApplicationController < ActionController::Base
  # Prevent CSRF attacks by raising an exception.
  protect_from_forgery with: :exception
end
```

This causes a hidden form field to silently be added to all forms:

```
<input name="authenticity_token" type="hidden"
value="SLNTcHBDnzl21gPHVoF3DAUEGZLAXWYqZ1FQxBBlmek=">
```



Cross-site Request Forgery

Note: these authenticity tokens make load testing with Tsung difficult, so I recommend disabling CSRF protection while load testing in this class.



Firewalls

Used to secure devices from outside access

- Enforces access control policy between two networks
- Two designs: restrict “bad traffic” or permit “good traffic”
- Designed to operate at different layers of the network stack

Can be standalone hardware devices or software

- Often included in multi-purpose device e.g. switch or load balancer



When to use a Firewall

- Use firewalls only when they significantly reduce risk
- Employ firewalls to protect sensitive data
 - Critical PII, PCI compliance, etc.
- Firewalls should be treated like perimeter security
 - Like the locks on your house
- Consider the value of what you are protecting and the cost to firewall it
 - Like your house, there are some things that are not worth protecting
 - Can you think of examples?



When to use a Firewall

Firewalls are often overused

“Failed firewalls are the #2 driver of site downtime after failed databases”

- **Scalability Rules**, by Martin Abbott

Can create difficult to scale chokepoint for either network traffic or transaction volume

May have impact on availability

- DDoS attacks on session state memory



Common Firewalls

- Software
 - Included with operating systems (ipfw)
 - Can also buy standalone
- Hardware
 - Cisco ASA, Citrix AppFirewall, F5 AFM
- EC2 Security Groups
 - Allow specific protocols and ports to access server
 - Can restrict machines to only accept traffic from Elastic Load Balancer
- A typical large scale web service will use both hardware and software firewalls



For Next Time...

Full-class demos this week!

Thursday's Guest lecture **Chehai Wu, Principal Software Engineer at Appfolio**

Please be on time and ask lots of questions.

