

---

# Internet Security [1]

VU 188.366

## *Web Application Security (1/2)*

Paolo Milani Comparetti, Christian Platzer,  
**Gilbert Wondracek**, Markus Huber, Edgar Weippl  
[inetsec@iseclab.org](mailto:inetsec@iseclab.org)

# News from the Lab

Int. Secure Systems Lab  
Vienna University of Technology

---

- About 88 people solved *Challenge 1*
  - good work!
  
- *Challenge 2* will start tomorrow
  - start of challenge scheduled at 10:00
  - SQL Injection

# Overview

*Int. Secure Systems Lab  
Vienna University of Technology*

- Web Application Security (I + II)
  - basics
  - the (2010) OWASP Top Ten Web application vulnerabilities
  - SQL injections
  - parameter injections
  - broken authentication
  - Cross Site Scripting
  - web-based buffer overflows / mitigation
  - insecure storage
  - DoS attacks

# Web Application Security

Int. Secure Systems Lab  
Vienna University of Technology

- Web Application
  - very short definition: a program that runs on a server, accepts input from “outside” via the web, processes it, and finally returns some answer
- Typical setting
  - a web application is deployed
  - it accepts HTTP requests from about anyone.
  - this means that your web application code is part of your *security perimeter*

# Web Application Security

*Int. Secure Systems Lab  
Vienna University of Technology*

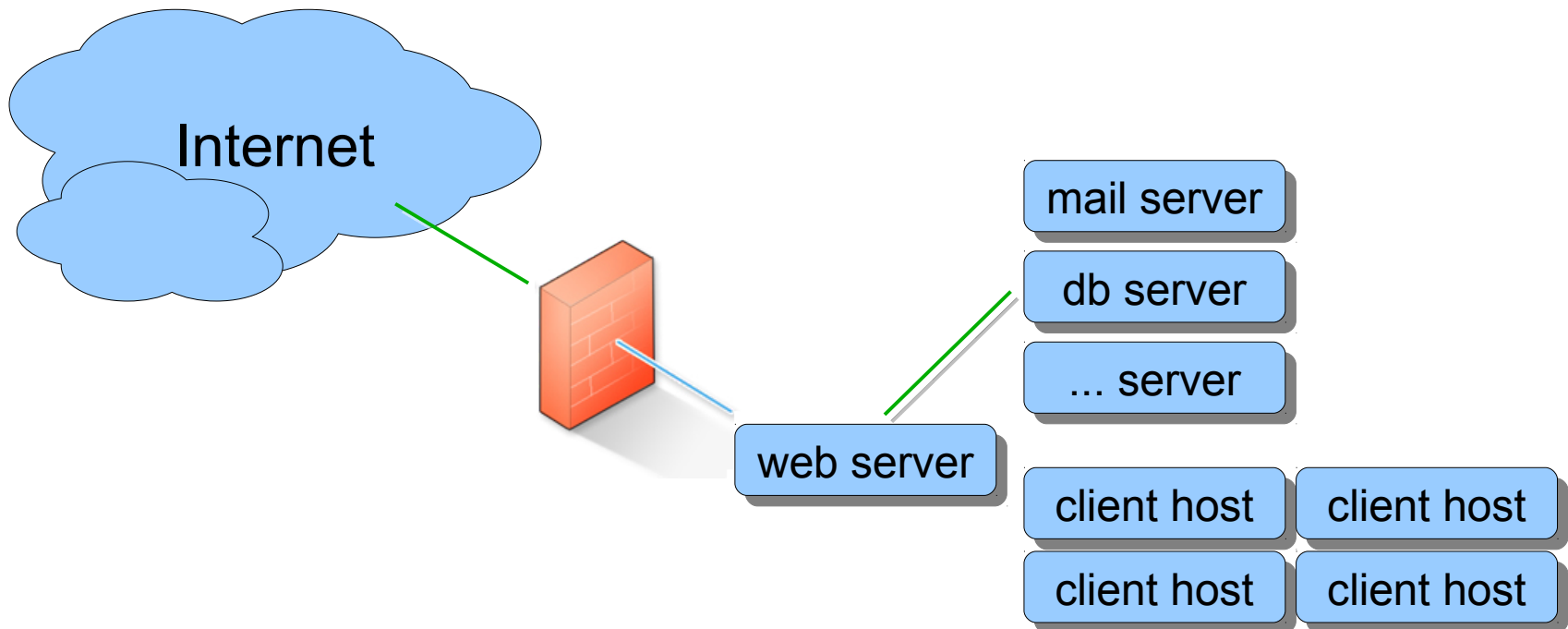
- Perimeter based security
  - controlling access on all entry / exit points of the network



# Web Application Security

*Int. Secure Systems Lab  
Vienna University of Technology*

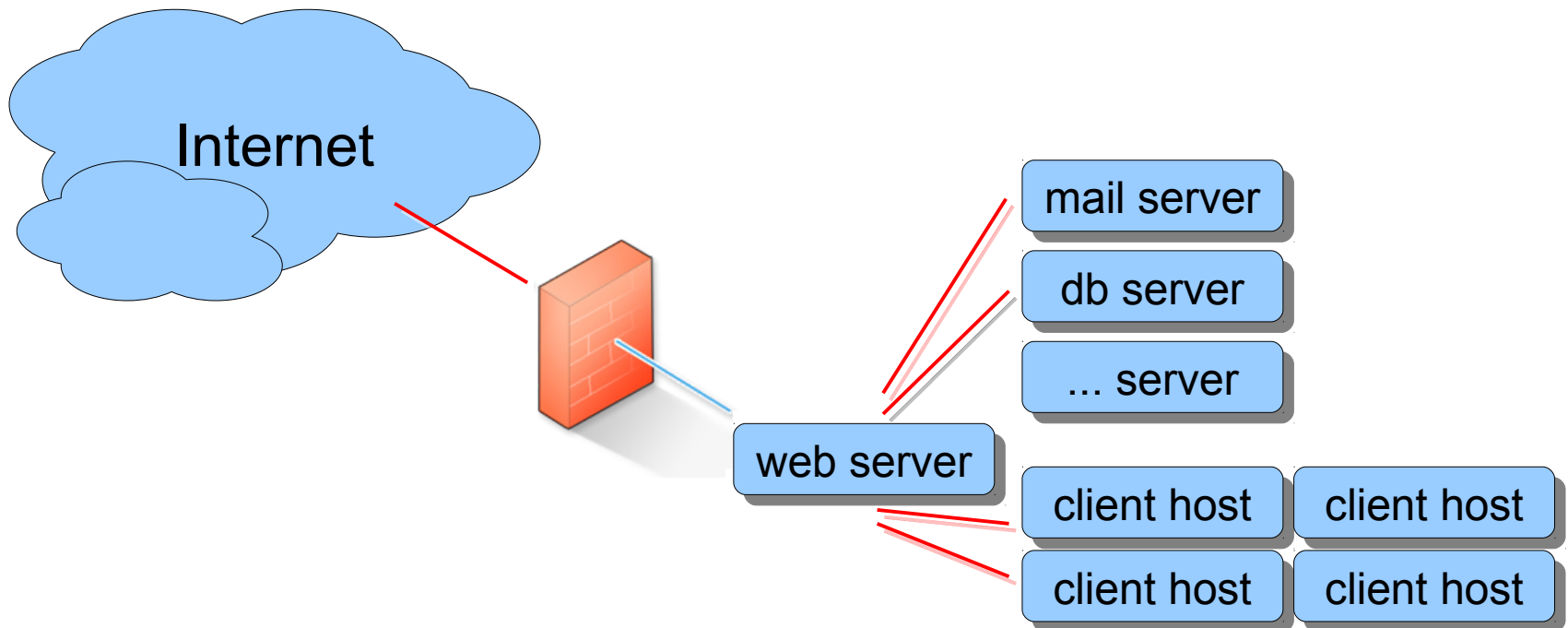
- Perimeter based security
  - controlling access on all entry / exit points of the network



# Web Application Security

*Int. Secure Systems Lab  
Vienna University of Technology*

- Perimeter based security
  - controlling access on all entry / exit points of the network



# On a typical Web server...

*Int. Secure Systems Lab  
Vienna University of Technology*

- Your host has port 80 open
- Some server-side software is running
  - OS
  - web server
    - main application (e.g., Apache)
    - plugins
    - servlets
    - script interpreters (CGI – Common Gateway Interface, Perl, ...)
  - big vulnerability surface
  - attacks that are hidden in valid HTTP requests often pass firewalls without notice

# HTTP and Web Application Basics

Int. Secure Systems Lab  
Vienna University of Technology

- HTTP
  - protocol used to talk to web applications (request web site, send data, receive response)
- HTTP transactions follow the same general format
  - 3 part client request / server response
    - request or response line
    - header section
    - entity body
  - client initiates a transaction as follows:

```
GET /index.html?param=value HTTP/1.0
```

# HTTP and Web Application Basics

- All HTTP transactions follow the same general format
  - 3 part client request / server response
    - request or response line
    - header section
    - no entity body
  - client initiates a transaction as follows:

```
GET /search?q=searchterm HTTP/1.1
Host: www.google.at
User-Agent: Mozilla/5.0 ... Firefox/3.5.8
Accept: text/html,...
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
```

# HTTP and Web Application Basics

Int. Secure Systems Lab  
Vienna University of Technology

- All HTTP transactions follow the same general format
  - 3 part client request / server response
    - request or **response line**
    - **header section**
    - **entity body**
  - server replies to a transaction as follows:

```
HTTP/1.1 200 OK  
Date: Fri, 09 Apr 2010 12:40:23 GMT  
Content-Type: text/html; charset=UTF-8
```

```
<html><head>  
<title>searchterm - Google-Search</title>  
</head><body bgcolor="#e5eccc">
```

# HTTP and Web Application Basics

- All HTTP transactions follow the same general format
  - 3 part client request / server response
    - request or **response line**
    - **header section**
    - **entity body**
  - server replies to a transaction as follows:

```
HTTP/1.1 200 OK
Date: Fri, 09 Apr 2010 12:40:23 GMT
Content-Type: text/html; charset=UTF-8
Content-Encoding: gzip

e0a
.....r...=_.....P.(.*.....6.$..t..tg...
```

# HTTP and Web Application Basics

Int. Secure Systems Lab  
Vienna University of Technology

- All HTTP transactions follow the same general format
  - 3 part client request / server response
    - request or response line
    - header section
    - entity body
- After sending the request and headers, the client *may* send additional data
  - form data
  - mostly used by CGI programs using the POST method
  - for the GET method, the parameters are encoded into the URL

# HTTP and Web Application Basics

Int. Secure Systems Lab  
Vienna University of Technology

- All HTTP transactions follow the same general format
  - 3 part client request / server response
    - request or response line
    - header section
    - entity body
  - client initiates a transaction as follows:

```
POST /search HTTP/1.1
Host: www.google.at
...
Content-Type: application/x-www-form-urlencoded
Content-Length: 12

q=searchterm
```

# Web Server Scripting

*Int. Secure Systems Lab*  
*Vienna University of Technology*

- HTTP alone is usually not enough to create web apps
  - scripting languages are used to increase the functionality
  - examples: Perl, Python, ASP, JSP, PHP
- Script interpreters are installed on the Web server
  - usually return HTML output that is then forwarded to the client
- Template engines are often used to power web sites
  - e.g., Cold Fusion, Cocoon, Zope
  - these engines often use scripting languages

# Web Application Example

- Objective: Write an application that accepts a username and password and echoes (displays) them
- First, we write HTML code and use forms

```
<html><body>  
<form action="/scripts/login.pl" method="post">  
Username: <input type="text" name="username"> <br>  
Password: <input type="password" name="password"> <br>  
<input type="submit" value="Login" name="login">  
</form>  
</body></html>
```

# Web Application Example

- Second, here is the corresponding Perl script that prints the username and password passed to it:

```
#!/usr/local/bin/perl

uses CGI;

$query = new CGI;

$username = $query->param("username");

$password = $query->param("password");

...

print "<html><body> Username: $username <br>
           Password: $password <br>
</body></html>";
```

# Know Your Enemy

*Int. Secure Systems Lab  
Vienna University of Technology*

- Most web app users will be benign, but...
  - even if you think you are too small for hackers to target you, you should expect attacks → automated attacks happen all the time
    - automated SQL injections often hit thousands of websites
  - even Intranet applications can be vulnerable from outside
  - malicious content delivered through Web browsing can compromise or hijack intranet client nodes and cause them to attack an intranet web application
  - possible measure against insider attacks: Define policies so that internal users cannot access your web application

# OWASP

Int. Secure Systems Lab  
Vienna University of Technology



## **Open Web Application Security Project** [www.owasp.org](http://www.owasp.org)

- dedicated to help organizations understand and improve the security of their web applications and web services.
- *Top Ten vulnerability list* was created to point corporations and government agencies to the most serious of these vulnerabilities.
- web application security has become a hot topic as companies race to make content and services accessible though the web. At the same time, attackers are turning their attention to the common weaknesses created by application developers.

# Top Ten Web Application Vulnerabilities

*Int. Secure Systems Lab  
Vienna University of Technology*

## A1 – Injection

- Injection flaws occur when untrusted data is sent to an interpreter as part of a command or query.
  - Examples: SQL, OS, and LDAP injection,
- Data sent by the attacker is being interpreted as commands in the application context

```
SELECT * FROM T WHERE X=[5]
```

```
SELECT * FROM T WHERE X=[5; DELETE * FROM T;]
```

# Top Ten Web Application Vulnerabilities

*Int. Secure Systems Lab  
Vienna University of Technology*

## A2 – Cross Site Scripting (XSS)

- XSS flaws occur whenever an application takes untrusted data and sends it to a web browser without proper validation and escaping.
- XSS allows attackers to execute script in the victim's browser which can hijack user sessions, deface web sites, or redirect the user to malicious sites.

# Top Ten Web Application Vulnerabilities

*Int. Secure Systems Lab  
Vienna University of Technology*

## A3 – Broken Auth. and Session Management

- Application functions related to authentication and session management are often not implemented correctly, allowing attackers to compromise passwords, keys, session tokens, or exploit implementation flaws to assume other users' identities.

# Top Ten Web Application Vulnerabilities

*Int. Secure Systems Lab  
Vienna University of Technology*

## A4 – Insecure Direct Object References

- A direct object reference occurs when a developer exposes a reference to an internal implementation object, such as a file, directory, or database key. Without an access control check or other protection, attackers can manipulate these references to access unauthorized data.

# Top Ten Web Application Vulnerabilities

*Int. Secure Systems Lab  
Vienna University of Technology*

## A5 – Cross Site Request Forgery (CSRF)

- A CSRF attack forces a logged-on victim's browser to send a forged HTTP request, including the victim's session cookie and any other authentication information, to a vulnerable web application.
- This allows the attacker to force the victim's browser to generate requests the vulnerable application thinks are legitimate requests from the victim.

# Top Ten Web Application Vulnerabilities

*Int. Secure Systems Lab  
Vienna University of Technology*

## A6 – Security Misconfiguration

- Security depends on having a secure configuration defined for the application, framework, web server, application server, and platform.
- All these settings should be defined, implemented, and maintained as many are not shipped with secure defaults.

# Top Ten Web Application Vulnerabilities

*Int. Secure Systems Lab  
Vienna University of Technology*

## A7 – Insecure Cryptographic Storage

- Many web application do not properly protect sensitive data, such as credit cards, SSNs, and authentication credentials, with appropriate encryption or hashing.
- Attackers may use this weakly protected data to conduct identity theft, credit card fraud, or other crimes.

# Top Ten Web Application Vulnerabilities

*Int. Secure Systems Lab  
Vienna University of Technology*

## A8 - Failure to Restrict URL Access

- Many web applications check URL access rights before rendering protected links and buttons.
- However, applications need to perform similar access control checks when these pages are accessed, or attackers will be able to forge URLs to access these hidden pages anyway.

# Top Ten Web Application Vulnerabilities

*Int. Secure Systems Lab  
Vienna University of Technology*

## A9 – Insufficient Transport Layer Protection

- Applications frequently fail to encrypt network traffic when it is necessary to protect sensitive communications.
- When they do, they sometimes support weak algorithms, use expired or invalid certificates, or do not use them correctly.

# Top Ten Web Application Vulnerabilities

*Int. Secure Systems Lab  
Vienna University of Technology*

## A10 – Unvalidated Redirects and Forwards

- Web applications frequently redirect and forward users to other pages and websites, and use untrusted data to determine the destination pages.
- Without proper validation, attackers can redirect victims to phishing or malware sites, or use forwards to access unauthorized pages.

# More Web Application Vulnerabilities

Int. Secure Systems Lab  
Vienna University of Technology

## *Former5* – Buffer Overflows

- Buffer overflows: Web application components in languages that do not properly validate input can be crashed and, in some cases, used to take control of a process.
- These components can include CGI, libraries, drivers, and web application server components.

# More Web Application Vulnerabilities

Int. Secure Systems Lab  
Vienna University of Technology

## *Former7* – Improper Error Handling

- Error conditions that occur during normal operation are not handled properly.
- If an attacker can cause errors to occur that the web application does not handle, they can gain detailed system information, deny service, cause security mechanisms to fail, or crash the server.

# More Web Application Vulnerabilities

Int. Secure Systems Lab  
Vienna University of Technology

## *Former9* – Denial of Service (DoS)

- Attackers can consume web application resources to a point where other legitimate users can no longer access or use the application
- Attackers can also lock users out of their accounts or even cause the entire application to fail.

# More Web Application Vulnerabilities

Int. Secure Systems Lab  
Vienna University of Technology

## *Former1* – Unvalidated Input

- Information from web requests is not validated before being used by a web application. Attackers can use these flaws to attack backend components through a web application.

*Root cause for many attacks – current Top 10 more precise on vulnerability classes*

# Unvalidated Input

- Web applications use input from HTTP requests (and occasionally files) to determine how to respond.
  - attackers can tamper with any part of an HTTP request, including the URL, query string, headers, cookies, form fields, and hidden fields, to try to bypass the site's security mechanisms.
  - common input tampering attempts include XSS, SQL injection, hidden field manipulation, parameter injection
- Some sites attempt to protect themselves by filtering malicious input.
  - problem: there are many different ways of encoding information

# Unvalidated Input

- Many web applications rely on client-side mechanisms to validate user input
  - client side validation mechanisms are easily bypassed, leaving the web application without any protection against malicious parameters
- How to determine if you are vulnerable?
  - the traditional way: have a detailed code review, searching for all the calls where information is extracted from an HTTP request
  - easy to miss code parts, manual effort high

# Unvalidated Input

- Taint analysis can be used to find vulnerabilities
  - initially, taint (“mark“) each user provided input.
  - propagate information during code execution:
    - variable assignments, modifications, etc.
  - remove taint status when content is sanitized.
  - do not allow tainted data as arguments for security relevant system interaction:
    - executing commands, accessing database, etc.
- Perl
  - built in support for taint analysis
- Pixy
  - PHP taint engine (<http://pixybox.iseclab.org>)

# Unvalidated Input

- Perl script that looks for files:

```
my $arg=shift;

my $arg_len=length($arg);
if ($arg_len <= 0) {
    print "boring\n";
    exit(1);
}

print "displaying files with filter '$arg':\n";
system("ls $arg");
```

**No Validation!**

# Unvalidated Input

- Looking for files (revisited), *bad sanitation*:

```
my $arg=shift;
...
if ($arg =~ m/;/) {
    print "my mother told me to sanitize input!\n";
    exit(1);
}
print "displaying files with filter '$arg':\n";
system("ls $arg");
```

# Unvalidated Input

- How to protect yourself?
  - the best way to prevent parameter tampering is to ensure that all parameters are validated before they are used.
  - a centralized component or library is likely to be the most effective, as the code performing the checking should be all in one place.
- Parameters should be validated against a “positive” specification that defines:
  - data type (string, integer, real, etc...);
  - allowed character set; minimum and maximum length;
  - if null is allowed; if the parameter is required or not; if duplicates are allowed; numeric range; specific legal values (enumeration); specific patterns (regular expressions) .....

# Unvalidated Input

- Looking for files (once again), *better sanitation*:

```
my $arg=shift;
...
if ($arg =~ m /^[A-Za-z0-9_\-\.*]*\.[A-Za-z0-9_\-\.*]*$/) {
    print "displaying files with
           filter '$arg':\n";system("ls $arg");
}
else {
    print "my mother told me to sanitize input!\n";
}
```

# SQL Injections

- Injection flaws allow attackers to relay malicious code through a web application to another system
  - these attacks include calls to the operating system via system calls, the use of external programs via shell commands, as well as calls to backend databases via SQL
- SQL injection is a particularly widespread and dangerous form of injection attack
  - to exploit a SQL injection flaw, the attacker must find a parameter that the web application uses to construct a database query.

# SQL Injections

- By carefully embedding malicious SQL commands into the content of the parameter, the attacker can trick the web application into forwarding a malicious query to the database
- The consequences are particularly damaging, as an attacker can obtain, corrupt, or destroy database contents.



# Simple SQL Injection Example 2

- If the user enters a ' (single quote) as the password, the SQL statement in the script would become:
  - `SELECT * FROM users`  
`WHERE username=' ' AND password = ' ''`
  - SQL error message would be generated
- If the user enters (injects): ' or username='john' as the password, the SQL statement in the script would become:
  - `SELECT * FROM users`  
`WHERE username=' ' AND password = ''`  
`or username= 'john'`
  - hence, a *different* SQL statement has been injected than what was originally intended by the programmer!

# Obtaining Information using Errors

- Errors returned from the application might help the attacker (e.g., ASP – default behavior)
  - Username: ' union select sum(id) from users --  
Microsoft OLE DB Provider for ODBC Drivers error '80040e14' [Microsoft] [ODBC SQL Server Driver][SQL Server]Column 'users.id' is invalid in the select list because it is not contained in an aggregate function and there is no GROUP BY clause.  
/process\_login.asp, line 35
- Make sure that you do not display unnecessary debugging and error messages to users.
  - for debugging, it is always better to use log files (e.g., error log).

**thanks for the  
info :-)**

# Some SQL Attack Examples

- `select * ...; INSERT INTO user VALUES("user","h4x0r");`
  - attacker inserts a new user into the database
- The attacker could use “stored procedures” (e.g., in SQL Server)
  - `xp_cmdshell()`
  - “bulk insert” statement to read any file on the server
  - e-mail data to the attacker’s mail account
  - play around with the registry settings
- `SELECT *... ; DROP table SensitiveData;`
- Appending “;” character does not work for all databases. Might depend on the driver (e.g., MySQL)

# Advanced SQL Injection

Int. Secure Systems Lab  
Vienna University of Technology

- Web applications will often escape the ' and " characters (e.g., PHP).
  - this will prevent many SQL injection attacks... but there might still be vulnerabilities
- In some applications, database fields might not be strings but numbers. Hence, ' or " characters are not necessary (e.g., ... **WHERE id=1**)
- Attacker might still inject strings into a database by using the "char" function (e.g., SQL Server):
  - **INSERT INTO users (id, name)**  
**VALUES (666,char(0x63)+char(0x65)...)**

# Blind SQL Injection

- A typical countermeasure is to prohibit the display of error messages. But, is this enough?
  - no, your application may still be vulnerable to *blind SQL injection*
- Let's look at an example:
  - suppose there is a news site
  - press releases are accessed with `pressRelease.jsp?id=5`
  - SQL query is created and sent to the database:

```
SELECT title, description
FROM pressReleases WHERE id=5;
```
  - any error messages are smartly filtered by the application

# Blind SQL Injection

- How can we inject statements into the application and exploit it?
  - we do not receive feedback from the application so we can use a trial-and-error approach
  - first, we try to inject

```
pressRelease.jsp?id=5 AND 1=1
```
  - the SQL query is created and sent to the database:

```
SELECT title, description
FROM pressReleases WHERE id=5 AND 1=1
```
  - if there is an SQL injection vulnerability, the *same* press release should be returned
  - if input is validated, `id=5 AND 1=1` should be treated as value

# Blind SQL Injection

- When testing for vulnerability, we know  $1=1$  is always true
  - however, when we inject other statements, we do not have any information
  - what we know: If the same record is returned, the statement must have been true
  - for example, we can ask server if the current user is "h4x0r":  
`pressRelease.jsp?id=5 AND user_name()='h4x0r'`
  - by combining subqueries and functions, we can ask more complex questions (e.g., extract the name of a database character by character)

# Second Order SQL Injection

- SQL is injected into an application, but the SQL statement is invoked at a later point in time
  - e.g., guestbook, statistics page, etc.
- Even if application escapes single quotes, second order SQL injection might be possible
  - attacker sets user name to: `john'--`, application safely escapes value  
(`--` is used for expressing comments in SQL Server)
  - at a later point, attacker changes password (and “sets” a new password for victim *john*):

```
UPDATE users SET password= ...  
WHERE database_handle("username")='john'--'
```

# SQL Injection Solutions

Int. Secure Systems Lab  
Vienna University of Technology

- Developers must never allow client-supplied data to modify SQL statements
  - best protection is to isolate application from SQL
  - all SQL statements required by application should be stored procedures on the database server
  - the SQL statements should be executed using safe interfaces such as JDBC *CallableStatement* or ADO's *Command Object*
  - both prepared statements and stored procedures compile SQL statements before user input is added

# SQL Injection Solutions

- Let us use `pressRelease.jsp` as an example. Here our code:

```
String query = "SELECT title, description from  
    pressReleases WHERE id= "+  
    request.getParameter("id");  
Statement stat = dbConnection.createStatement();  
ResultSet rs = stat.executeQuery(query);
```

- The first step to secure the code is to take the SQL statements out of the web application and into DB

```
CREATE PROCEDURE getPressRelease @id integer  
AS  
SELECT title, description  
FROM pressReleases WHERE id = @id
```

# SQL Injection Solutions

- Now, in the application, instead of string-building SQL, call stored procedure:

```
CallableStatements cs =  
    dbConnection.prepareCall("{call  
    getPressRelease(?)}");  
  
i = Int.parseInt(request.getParameter("id"))  
cs.setInt(1, i);  
ResultSet rs = cs.executeQuery();
```

- In ASP.NET, there is a similar mechanism

# Discovering “clues” in HTML code

- Developers are notorious for leaving statements like *FIXME*, *Code Broken*, *Hack*, etc... inside the source code.
  - always review the source code for any comments denoting passwords, backdoors, or something doesn't work right.
- Hidden fields (`<input type="hidden"...>`) are sometimes used to store temporary values in Web pages.
  - these can be changed with ease (*hidden field tampering!*)
- Tools can support, facilitate this task
  - Firebug (Firefox), Dragonfly (Opera)

# Conclusion

- In this lecture, we took a first look at “higher-level” security issues.
  - web applications can have many weaknesses
  - we started with the web and will continue looking at it next week
- Don't forget about *Challenge 2 – Good luck!*
- Next week, we continue with web security
  - Same time, same place