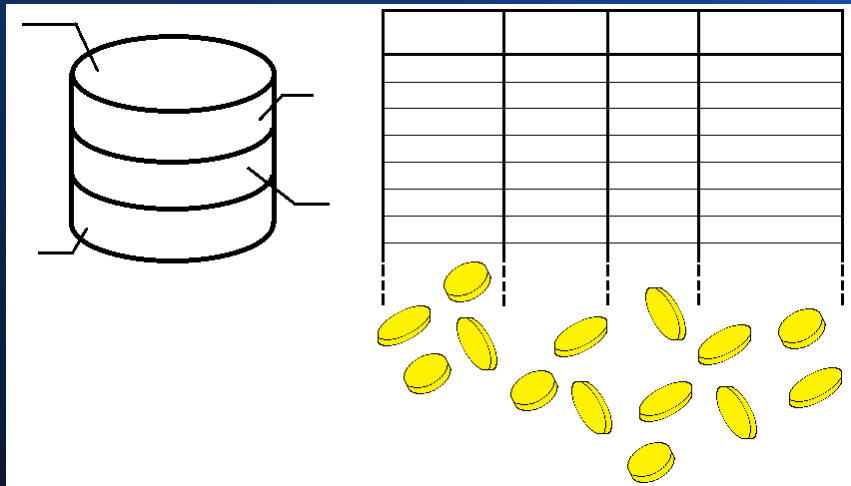


Database Security

Attacks on Databases &
Best Practices to Prevent them

Database Security

- Why is it important?
 - Contains valuable & sensitive information



- Backbone of most companies

Database Security Risks

- Excessive Privileges on Accounts
 - Users can do much more than they need to
- Privilege Abuse
- Unprotected backup data
- Unmaintained databases

Database Security Risks

- Database injection attacks
 - SQL/NoSQL injection attacks
- Weak Authentication
 - Brute-force attacks, etc...
- Human error
 - Social engineering, reusing passwords, etc...

Best Practices

- Excessive Privileges on Accounts
 - Managing user access rights
 - Query-level access control
 - Define specific read/write functions
 - Triggers
- Privilege abuse
 - Control policies on how data is accessed
 - Time of day, location, volume of data, etc...

Best Practices

- Unprotected backup data
 - Archive and encrypt backup data
- Unmaintained databases
 - Patching to fill security holes
 - Use Intrusion Prevention Systems (IPS) to monitor and block known exploits

Best Practices

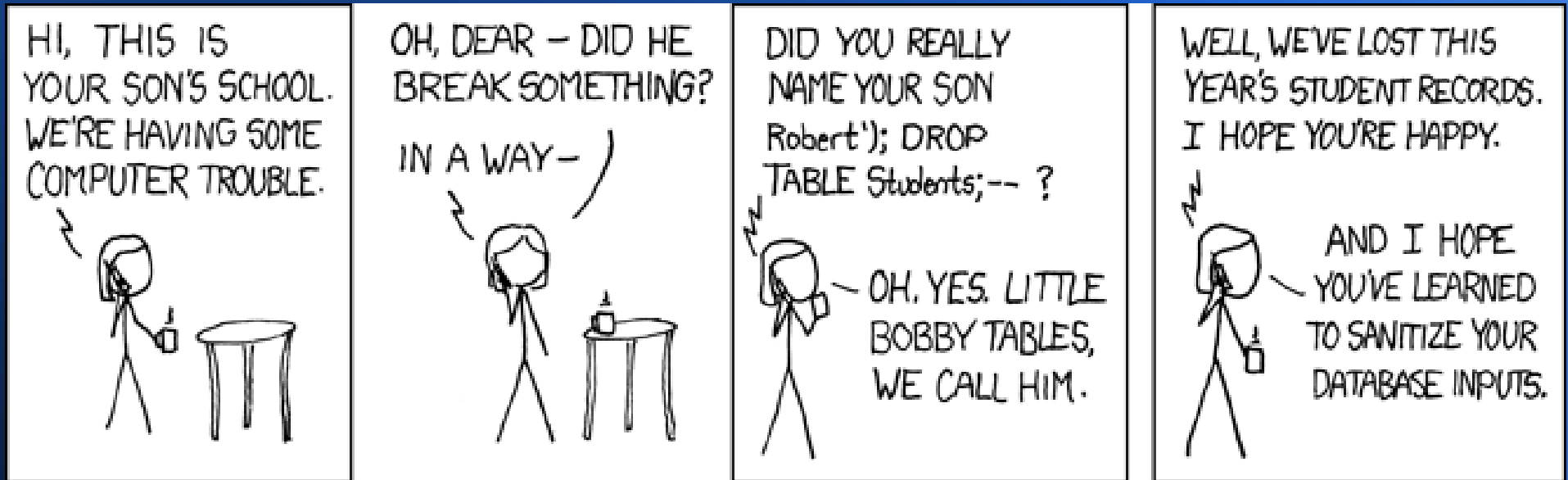
- Weak Authentication
 - Use of modern hashing schemes (Argon2, etc..)
 - 2-way Authentication
- Human error
 - Training employees
 - Detect phishing attacks
 - Internet & E-Mail usage
 - Password management

SQL Injection Attacks

- Especially important if the DB has a web interface
- „an attacker submits information that has been deliberately formulated in such a way that it results in that website misinterpreting it and taking unintended actions“

<https://www.esecurityplanet.com/threats/what-is-sql-injection.html>

SQL Injection Attacks



https://imgs.xkcd.com/comics/exploits_of_a_mom.png

SQL Injection Attacks

- Example

- Code:

- String query = "SELECT * FROM Accounts
WHERE username="" + username + ""
AND password="" + password + """;

- Input:
 - Username = "1' OR '1' = '1"
 - Password = "" OR " = ""

- Result:

- Query = "SELECT * FROM Accounts
WHERE username='1' OR '1' = '1'
AND password = " OR " = "";

SQL Injection Attacks

- Example

- Code:

- String query = "SELECT * FROM Accounts
WHERE username="" + username + ""
AND password="" + password + """;

- Input: - Username = "Joe');DROP TABLE Accounts; --"

- Result:

- Query = "SELECT * FROM Accounts
WHERE username='Joe');DROP TABLE Accounts;
--AND password ="";

SQL Injection Attacks

- Check your website for vulnerabilities
 - Automated SQL Injection Attack Tools
 - Havij (ITSecTeam, an Iranian security company)
 - SQLmap (open source; sqlmap.org)
 - jSQL (open source; github.com/ron190/jsql-injection)
 - TyrantSQL (open source; GUI version of SQLmap; sourceforge.net/projects/tyrantsql?source=directory)

SQL Injection Attacks

- Countermeasures:
 - Input sanitization
 - Check the input for dangerous characters
 - E.g. escape ' or "
 - Careful! Characters can be encoded differently, but still be interpreted by your system
 - Example Login over HTTP GET
 - Login.html?user=Joe';Drop Table Accounts;--
OR
 - Login.html?user=Joe%27%3bDrop%20Table%20Accounts%3b--

SQL Injection Attacks

- Countermeasures:
 - Validation
 - Check if the input data is in the format you want it to be
 - E.g.
 - E-mails contain an @
 - ID containing only numbers
 - Length of the input
 - White- or Blacklist characters

SQL Injection Attacks

- Countermeasures:
 - Prepared statements
 - Is given a SQL statement when created
 - Precompiled by the DBMS
 - Faster if the same statement is executed multiple times
 - Allows the use of parameters

SQL Injection Attacks

- Countermeasures:
 - Prepared statements (Parameterized Queries)
 - Represented by markers:
 - @ : ASP.NET
 - : : PHP
 - ? : JAVA
 - Parameter values are added to the query at execution time in a controlled manner
 - "The SQL engine checks each parameter that it is correct for its column and are treated literally, and not as part of the SQL to be executed"


SQL Injection Attacks

- Countermeasures
 - Prepared statements
 - In Java per functions: set*TYPE*(Index, Value)
 - setString, setLong, setDouble, setBytes, ...
 - Index is the number of the ? placeholder
 - Beginning with 1

```
Connection con = DriverManager.getConnection("jdbc:oracle://localhost:3306/", "bob", "mypassword");
String sql = "SELECT * FROM Accounts WHERE username = ? AND password = ?";
PreparedStatement prepStm = con.prepareStatement(sql);
prepStm.setString(1, "john");
prepStm.setString(2, "doe");
ResultSet result = prepStm.executeQuery();
```

SQL Injection Attacks

- Are prepared statements 100% safe?
 - Not necessarily...
 - The inner workings depend on the driver
 - Some only emulate prepared statements
 - Constructing the query using string concatenation with user input makes it unsafe
 - But! Parameters are not allowed for identifiers

```
 String column = getSelectedColumn();  
String sql = "SELECT " + column + " FROM Accounts WHERE username = ? AND password = ?";  
PreparedStatement prepStm = con.prepareStatement(sql);
```

SQL Injection Attacks

- Are prepared statements 100% safe?
 - 2nd order injection attacks
 - 1st order: the user input data is unsafe
 - But what if the database data is also unsafe?
 - It originates from the user most of the time anyway
 - If "Joe');DROP TABLE Accounts; --" is a stored user name
 - Do not use the user name with a simple string concat query!
- ➔ Proper use of prepared statements prevent most injection attacks, but there are always other attack vectors!

References

- <https://www.esecurityplanet.com/threats/what-is-sql-injection.html>
- <https://docs.oracle.com/javase/tutorial/jdbc/basics/prepared.html>
- <http://javabypatel.blogspot.com/2017/06/how-prepared-statement-works-internally-java.html>
- <https://security.stackexchange.com/questions/15214/are-prepared-statements-100-safe-against-sql-injection>
- <https://www.bcs.org/content/ConWebDoc/8852>