

# Secure Methods

by Jason Grembi

# Objectives

- Define Secure Coding
- Securing the Weakest Link
- Coding for Defense in Depth
- Code for the Least Privilege

# Secure Methods

- What is it?
  - Secure code means that the software has been development using industry standard best practices using a justifiably high confidence - but not guaranteeing absolutely.
  - Do not allow unauthorized user to access the application's assets

# Secure Methods

- Know the Types of Attacks
  - Social engineering attacks
  - Attacks against the application's software
  - Attacks against the supporting infrastructure
  - Physical attacks

# Secure Methods

- Social Engineering Attacks (continued)
  - Types of social engineering attacks:
    - *Organization penetration*
    - *IT infrastructure exploration*
    - *Phishing*
    - *Spam*
    - *Spoofing*
    - *Man in the middle*

# Secure Methods

- Attacks Against the Software Itself (continued)
  - Types of software attacks:
    - *Cross-site scripting (XSS)*
    - *Buffer overflows*
    - *SQL code injection*
    - *Time/logic bombs*
    - *Back door*

# Secure Methods

- Attacks Against the Infrastructure (continued)
  - Types of infrastructure attacks:
    - *Denial of service (DOS)*
    - *Virus*
    - *Worm*
    - *Trojans*
    - *Spyware*
    - *Adware*

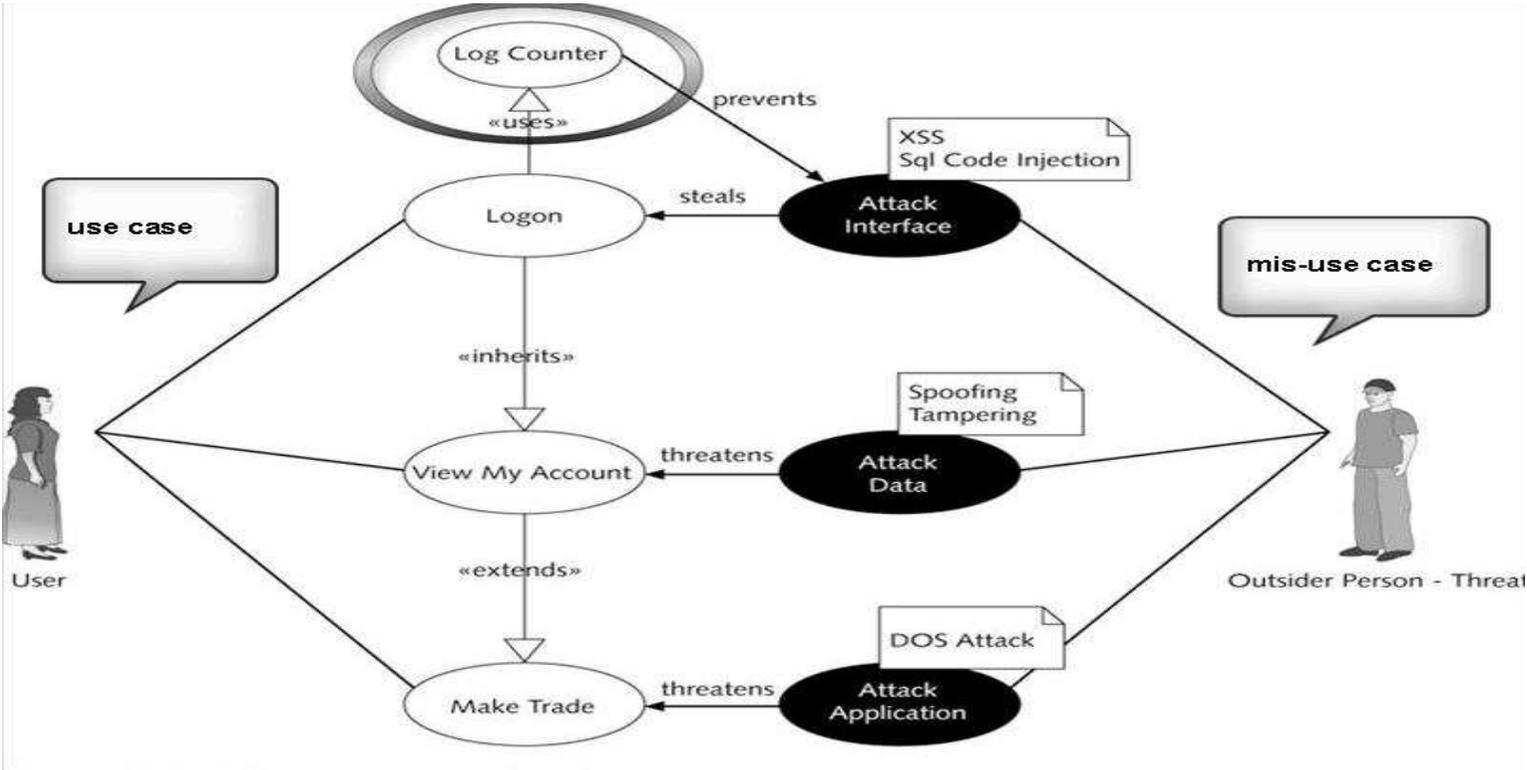
# Secure Methods

- Physical Attacks
  - Bringing down a system or blatantly stealing hardware so that attackers can hack into it at their own leisure is just as catastrophic as malicious code insertion
  - Flash drives and/or external hard drives are a convenient way to copy information and walk out without having to devise elaborate plans

# Secure Methods

- How to Counter Attack
  - Use a secure programming language such as Java, .NET
  - Make sure development team practices Software Diversity
  - Devise Use Cases
  - Code for Security

# Secure Methods



# Secure Methods

- Choosing the Right Countermeasure for the Misuse Case

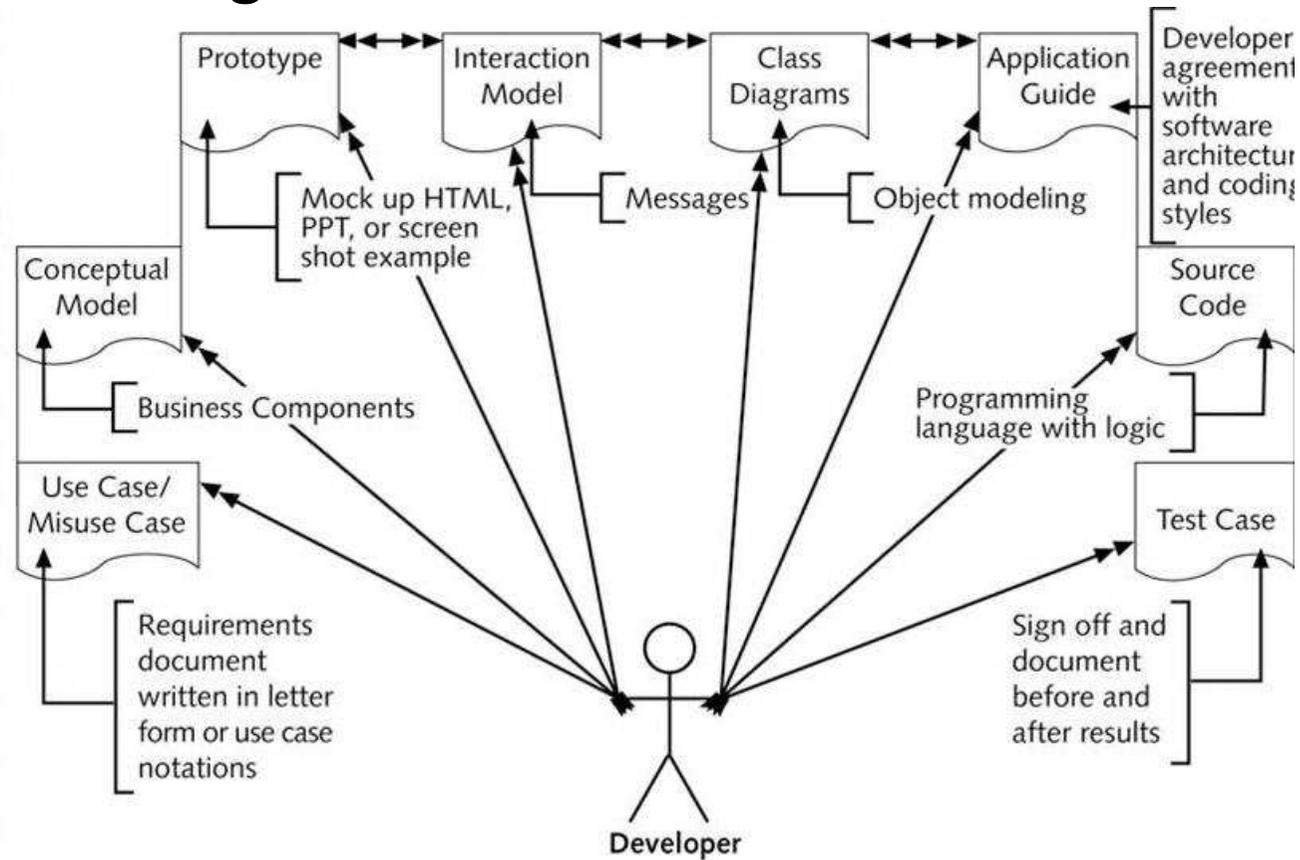
Misuse case	Design countermeasure	Principles
Man in the middle	<ul style="list-style-type: none"> <li>• Secure Sockets Layer (SSL)</li> <li>• Personal digital certificates</li> </ul>	<ul style="list-style-type: none"> <li>• Practice defense in depth</li> <li>• Promote privacy</li> <li>• Be reluctant to trust</li> </ul>
Tamper with input fields	<ul style="list-style-type: none"> <li>• POST methods</li> <li>• Web browser security</li> <li>• Secure Sockets Layer (SSL)</li> <li>• Personal digital certificates</li> </ul>	<ul style="list-style-type: none"> <li>• Promote privacy</li> <li>• Practice defense in depth</li> <li>• Be reluctant to trust</li> </ul>
Code tampering	<ul style="list-style-type: none"> <li>• Validating the input</li> <li>• Choosing a framework</li> <li>• Design patterns</li> <li>• Logging user navigation/requests</li> </ul>	<ul style="list-style-type: none"> <li>• Fail securely</li> <li>• Follow the principle of least privilege</li> <li>• Compartmentalize</li> <li>• Keep it simple</li> <li>• Remember that hiding secrets is hard</li> <li>• Be reluctant to trust</li> <li>• Use your community resources</li> </ul>
Steal username/password	<ul style="list-style-type: none"> <li>• Two-factor authentication (VPN)</li> </ul>	<ul style="list-style-type: none"> <li>• Secure the weakest link (users)</li> </ul>

# Secure Methods

- Whose responsible?
  - Management defines the Policies
  - Project Team carries out Policies through Access Rules
  - The Programmer works at the lowest level of details on how those Access Rules are carried out.

# Secure Methods

- The Programmer interacts with the most artifacts



# Secure Methods

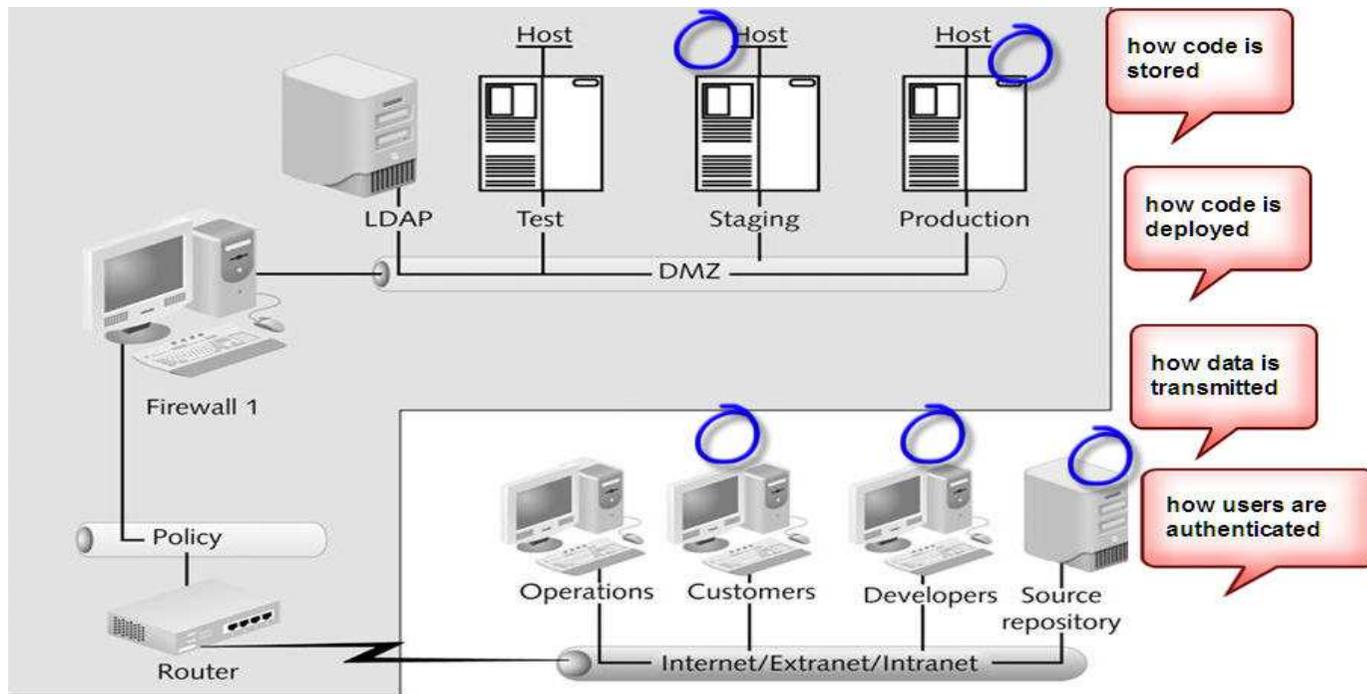
- Secure the Weakest Link
  - The weakest part of the system will most likely be attacked first
  - Analyzing and evaluating the weakest application layers and defensive layers is a principle that needs to be practiced at the beginning stages of software development (requirements) all the way through the maintenance stage

# Secure Methods

- Weak Links:
  - The data transmitted
  - The network
  - The host
  - The application

# Secure Methods

## – Weak Links



# Secure Methods

- Authenticating the User
  - Before wasting any more time on a request or taking the risk of allowing corrupted data into the application, make sure the user has the authority to use the services requested
  - Data authorization is a two-way street
    - First, the user's ID must be *authenticated* before the application responds to the user so that it knows what features to make available
    - Second, the user's request needs to be *authorized* again to make sure that data (the user's privileges) has not been changed or tampered with since it originally displayed on the screen

# Secure Methods

- Authentication consists of:
  - something you are
  - Something you know
  - Something you have



# Secure Methods

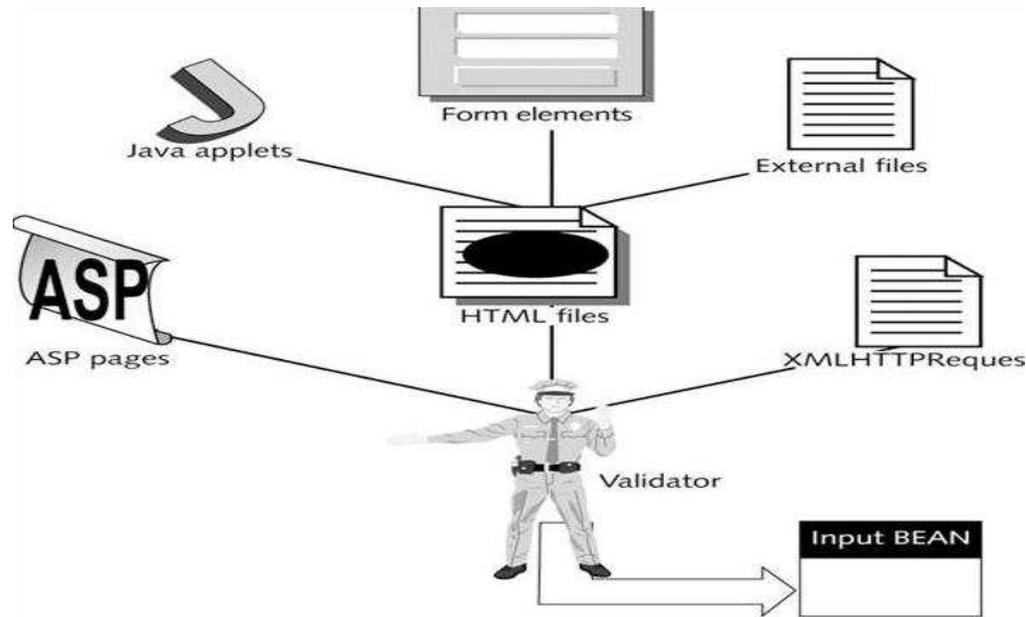
- Securing the Data Transmitted
  - The process in which your application picks up data for further processing needs to ensure:
    - Data is quarantined
    - Data is sanitized
    - Data is double-checked before used

# Secure Methods

- Cleansing the Request
  - Input cleansing is critical for securing the application
    - Whitelist values are predefined values that are acceptable for input
    - Take the **whitelist** values that were identified from the use case requirements and place them into a CONSTANTS or, better yet, into a database
    - Then code some utilities that reject HTML tags or **blacklist** values

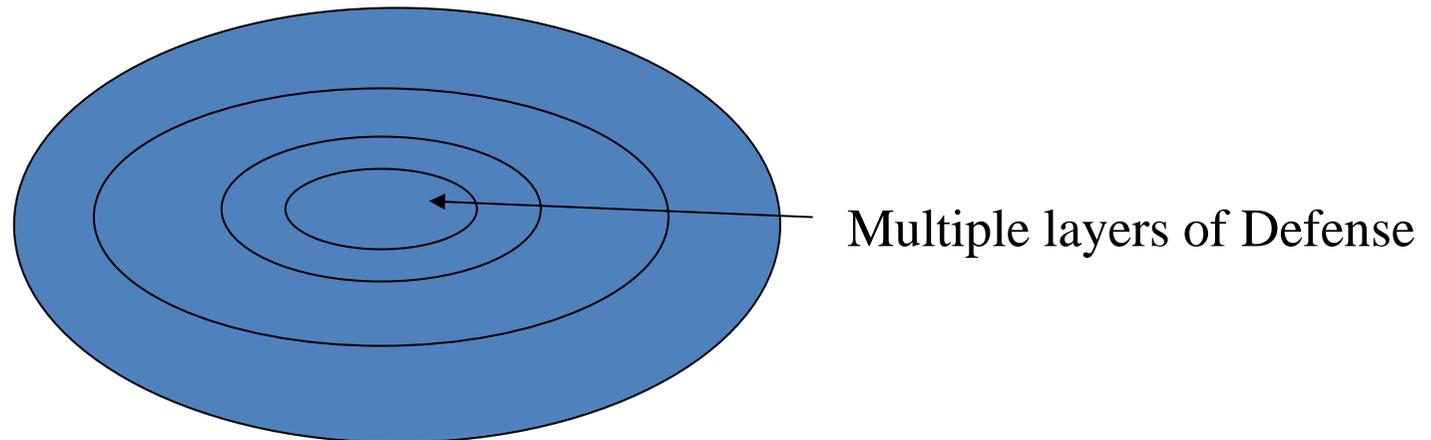
# Secure Methods

- Cleansing the Input Data
  - The first step in data collection is to run the data fields through a validation process and clean the data



# Secure Methods

- Defense in Depth
  - Defense in depth is designed on the principle that multiple layers of different types of protection provide substantially better protection
  - The goal is to limit access to certain features of the application



# Secure Methods

- Multi-Layer: Self-Monitoring Code
  - Self-monitoring code watches the user's activity and looks for unusual events
  - Some examples of self-monitoring code include the following:
    - *Clipping Levels*
    - *Alerting stakeholders* of invasion
    - *Unusual Attempts* at Assets

# Secure Methods

```
public void updateTempTableRegistry(String tableName)
{
    Connection con = null;
    PreparedStatement pstmt = null;
    ResultSet rs = null;
    try
    {
        con = dbConnMan.getConnection();
        String checkReportTempRegistry = "SELECT COUNT(*) TEMPCOUNT FROM REPORT_TEMP_REGISTRY WHERE TEMP_TABLE_NAME=?";
        pstmt = con.prepareStatement(checkReportTempRegistry);
        pstmt.setString(1, tableName);
        rs = pstmt.executeQuery();
        int tempCount;
        for(tempCount = 0; rs.next(); tempCount = rs.getInt("TEMPCOUNT"));
        if(tempCount == 0)
        {
            String sqlReportTempRegistry = "INSERT INTO REPORT_TEMP_REGISTRY VALUES(?,?)";
            pstmt = con.prepareStatement(sqlReportTempRegistry);
            pstmt.setString(1, tableName);
            pstmt.setLong(2, System.currentTimeMillis());
            pstmt.executeUpdate();
        } else
        {
            String sqlReportTempRegistry = "UPDATE REPORT_TEMP_REGISTRY SET LAST_ACCESS_TIME=? WHERE TEMP_TABLE_NAME=?";
            pstmt = con.prepareStatement(sqlReportTempRegistry);
            pstmt.setLong(1, System.currentTimeMillis());
            pstmt.setString(2, tableName);
            pstmt.executeUpdate();
        }
    }
    catch(Exception e)
    {
        e.printStackTrace();
    }
}
```

# Secure Methods

- Multi-Layer: Bounds Checking
  - Whitelist every Input
  - Count variable length
  - Check ArraySize

# Secure Methods

```
public class Constants {
    private static final Properties props = new Properties();
    private static final String CLASSNAME = "Constants";
    /** A handle to the unique Singleton instance. */
    static private Constants _instance = null;
    /** Creates a new instance of Constants */
    public Constants() {
        try {
            InputStream is = ((new Object()).getClass()).getResourceAsStream(Constants.PROPS_FILE_NAME);
            if(is == null) {
                throw new IOException("Properties file: " + Constants.PROPS_FILE_NAME + " NOT FOUND! ");
            }
            props.load(is);
            is.close();
        } catch (Exception e){
        }
    }
    static public Constants instance() {
        if(null == _instance) {
            _instance = new Constants();
        }
        return _instance;
    }
    public static final String SECRET_FORMULA = props.getProperty("secret.formula");
    public static final String FALSE_SECRET_FORMULA = props.getProperty("secret.formula.false");
    public static final String SECRET_FORMULA_GROUP_ID = props.getProperty("secret.formula.group_id");
    public static final String WHERE_AM_I = props.getProperty("where.am.i");
    public static final String DEST_REQ = props.getProperty("dest.required");
    public static final String ACTIVE = "A";
    public static final String DIS_ACTIVE = "A";
    public static final String WHITE_LIST_INPUT = " abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";
}
```

# Secure Methods

```
public void updateTempTableRegistry(String tableName)
{
    if (tableName.length() > 9) {
        return;
    }
    if (!Util.isValid(Constants.WHITE_LIST_INPUT, tableName)) {
        return;
    }

    Connection con = null;
    PreparedStatement pstmt = null;
    ResultSet rs = null;
    try
    {
        con = dbConnMan.getConnection();
        String checkReportTempRegistry = "SELECT COUNT(*) TEMPCOUNT FROM REPORT_TEMP_REGISTRY WHERE TEMP_TABLE_NAME=?";
        pstmt = con.prepareStatement(checkReportTempRegistry);
        pstmt.setString(1, tableName);
        rs = pstmt.executeQuery();
        int tempCount;
        for (tempCount = 0; rs.next(); tempCount = rs.getInt("TEMPCOUNT"));
        if (tempCount == 0)
        {
            String sqlReportTempRegistry = "INSERT INTO REPORT_TEMP_REGISTRY VALUES (?,?)";
            pstmt = con.prepareStatement(sqlReportTempRegistry);
            pstmt.setString(1, tableName);
            pstmt.setLong(2, System.currentTimeMillis());
            pstmt.executeUpdate();
        } else
        {
            String sqlReportTempRegistry = "UPDATE REPORT_TEMP_REGISTRY SET LAST_ACCESS_TIME=? WHERE TEMP_TABLE_NAME=?";
            pstmt = con.prepareStatement(sqlReportTempRegistry);
            pstmt.setLong(1, System.currentTimeMillis());
            pstmt.setString(2, tableName);
            pstmt.executeUpdate();
        }
    }
}
```

# Secure Methods

- Code for the Least Privilege
  - Give users the least amount of privilege required to perform the use case functionality
  - Applications that need access to other system resources; grant only what is needed

# Secure Methods

```
public class SCZoneManager extends ZoneManager {
    private static final Properties props = new Properties();
    private static final Logger log = Logger.getLogger(SCZoneManager.class);

    public SCZoneManager() {
        try {
            InputStream is = ((new Object()).getClass()).getResourceAsStream(Constants.PROPS_FILE_NAME);
            if(is == null) {
                throw new IOException("Properties file: " + Constants.PROPS_FILE_NAME + " NOT FOUND! ");
            }
            props.load(is);
            is.close();
        } catch (Exception e){
        }
    }

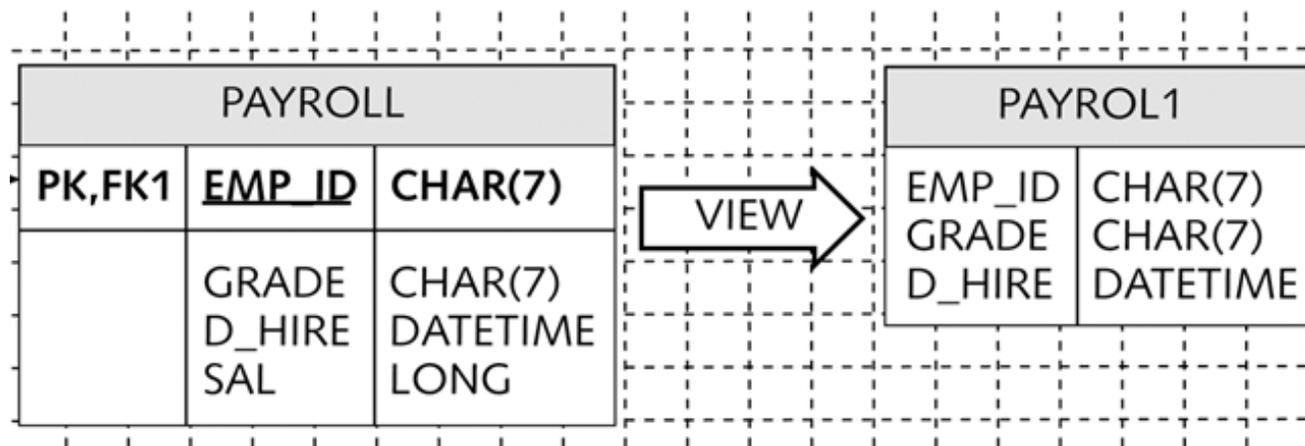
    protected String getIpAddresses()
    {
        return props.getProperty("secure.ip.var" );
    }

    public boolean isValidIP(HttpServletRequest request) {
        String ip = request.getHeader("PASSED_ID");

        if (ip == getIpAddresses())
        {
            return true;
        }
        return false;
    }
}
```

# Secure Methods

- Multi-Layer: Database Views
  - A database **view** provides a way to expose table columns to certain access paths while hiding others



# Secure Methods

- Multi-Layer: Stored Procedure

```
EXEC SQL
  SELECT COUNT(A.REGION_ID), A.NAME
  FROM REGION A, EMPLOYEE B, SALES C
  WHERE
  A.REGION_ID = '1234'
  AND A.REGION_ID = B.REGION_ID
  AND B.EMPLOYEE_ID = C.EMPLOYEE_ID
  AND C.SALE_AMOUNT > 50.00
  GROUP BY A.NAME
  ORDER BY 2, 1
END-EXEC
```

Static SQL is hard-coded  
into the program.

# Secure Methods

## – Fail Securely

- Fail securely is simply what happens when the system goes down
- Power failure, server crashes, and other network problems are reasons beyond the control of the application programmers
  - Address error-handling issues appropriately
  - Degrade gracefully

# Secure Methods

## – Fail Securely examples:

- Cron jobs that executes programs that wipes disk space
- Jobs that run in parrell with the application that check the status of the application. If it senses failure; alerts will be sent to Operations, any vulnerable asset can be locked down until freed by corrective measures.

# Secure Methods

- Summary
  - Define Secure Coding
    - Know the Types of Attacks
    - How to Counter Attack
    - Mis-Use Case
    - Right Countermeasure for the Misuse Case
  - Securing the Weakest Link
    - The data transmitted
    - The application
    - The host
    - The network

# Secure Methods

- Summary
- Coding for Defense in Depth
  - Layer Countermeasures
  - Clipping Levels
  - Alters
  - Monitors
  - Bounds Checking
  - Size
  - Whitelisting

# Secure Methods

- Summary
  - Code for the Least Privilege
    - Database View
- Fail Securely
  - Cron Jobs that Clean up
  - Jobs that lock assets