


Module 6:

SQL Server Security Threats and Countermeasures

Contents

Module Overview	1
Lesson 1: Threats from Authorized Users.....	2
Is it User Roll Enough for Security?	3
Examples of Threats	4
Countermeasures	6
Practice: Limiting Threats from an Authorized User	8
Lesson 2: Physically Stealing Data.....	9
Weak Points inside SQL Server.....	10
Weak Points Outside of SQL Server	11
Countermeasures	12
Practice: Protecting Database Back-up	13
Lesson 3: Data Transfer Sniffing.....	15
Client/Server Communications	16
Why a Firewall is not enough?	17
Countermeasures	18
Practice: Encrypt the Connection.....	19
Lesson 4: SQL Code Injection	20
What is SQL Code Injection?	21
Demonstration of SQL Server Injection.....	22
Countermeasures	23
Practice: Protecting from SQL Server Injection	24
Summary	27

Module Overview

- 
- Threats from Authorized Users
 - Physically Stealing Data
 - Data Transfer Sniffing
 - SQL Code Injection

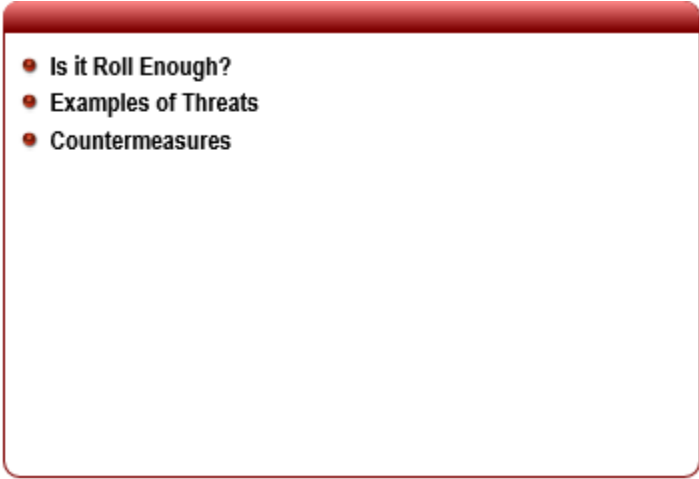
In this last module of the course, you will learn how to recognize security threats and apply correct and effective countermeasures to mitigate those threats. This module applies the cumulative knowledge that you have acquired throughout this course.

Objectives

After completing this module, you will be able to:

- Identify threats from authorized users
- Understand the risk from physically stealing data
- Prevent data transfer sniffing
- Avoid SQL code injection

Lesson 1: Threats from Authorized Users

- 
- Is it Roll Enough?
 - Examples of Threats
 - Countermeasures

In this lesson, you will learn that authorized users pose threats in some situations which cannot be predicted in the early stages of the software development process. In the first part of this lesson, we will discuss the “Is it user roll enough?” question. Later, we will explore some examples which will cast away any doubts about this dilemma.

Objectives

After completing this lesson, you will be able to:

- Answer the “Is it user roll enough?” question
- Identify threats
- Implement corresponding countermeasures

Is it User Roll Enough for Security?

- **Inner threats are more dangerous because we have a false sense of security**
 - false sense of security
 - we trust our users
- **User roll, as an authorization part in our environment, is not enough to ensure security and privacy elements**

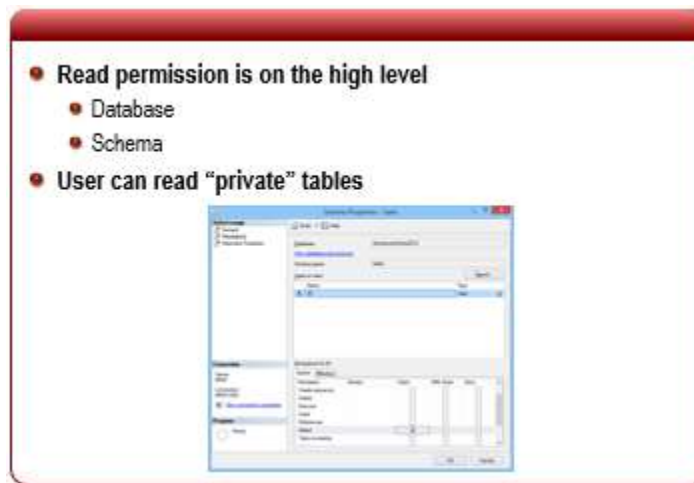
If we look at this question from an ideal world perspective, the answer would be yes. Unfortunately, this is not the case. We live in a world where the “inner” (within our working environment) and “outer” (outside working environment) threats should be consider as the same. In my opinion, inner threats are more dangerous because we tend to have a false sense of security. We trust our users, resources and processes, but we should be more cautious.

An authorized user is a person (or process) who is past the process of authentication. In the authorization part, the system will check the user policy and grant access to the corresponding assets.

So where is the threat? Users, like the system, can check the user policy and grant access to corresponding assets. Furthermore, an authorized user can decide to abuse his or her access to some assets. In this situation, the damage can be greater than that caused by outside threats.

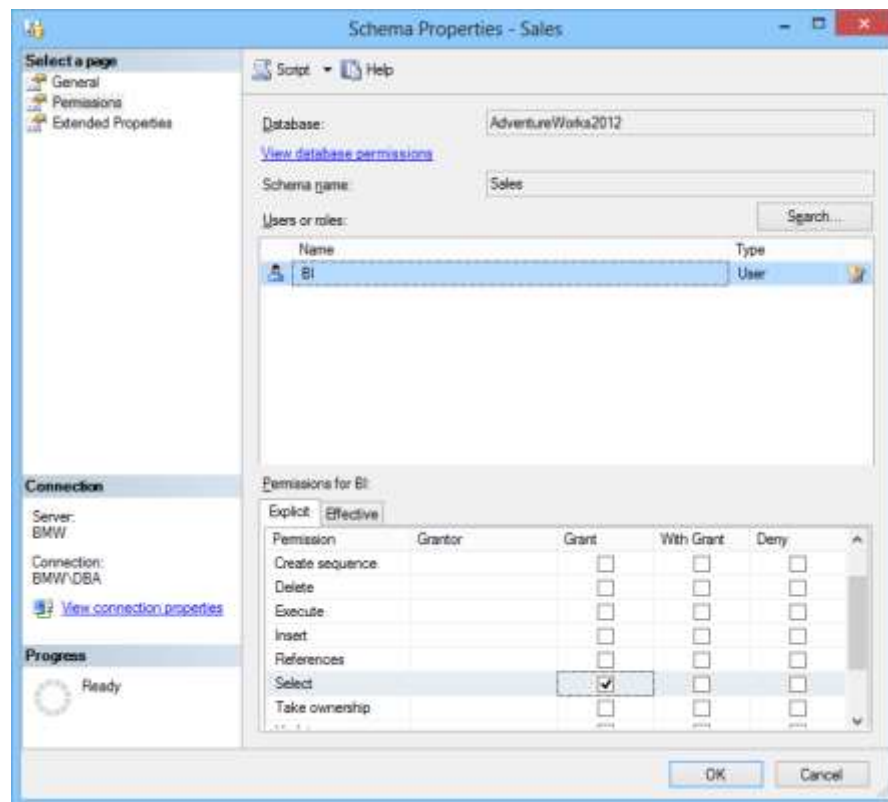
The final and correct answer to our initial question is no. User roll, as an authorization part in our environment, is not enough to ensure security and privacy elements.

Examples of Threats



Scenario 1:

Your business intelligence (BI) department requests access to the Sales schema in the AdventureWorks2012 database. They need it for the purpose of creating a report on a business analysis. The BI department has 15 information workers who are all under the same login mapped from the domain name DomainName\BI. Your task is to grant access to the database and schema.



To accomplish this task, you must:

- Map the domain login as a database user; and
- Grant a db_datareader fixed database role.

From this point, all 15 information workers can access all objects, within that schema, with SELECT permission.

Consequences: Sales schema contains the following table: Sales.CreditCard. This table is full of secure/private and personal information, such as credit card numbers, expiry dates, etc. From this point, there is no limitation and we are relaying on their sense of morality and ethics.

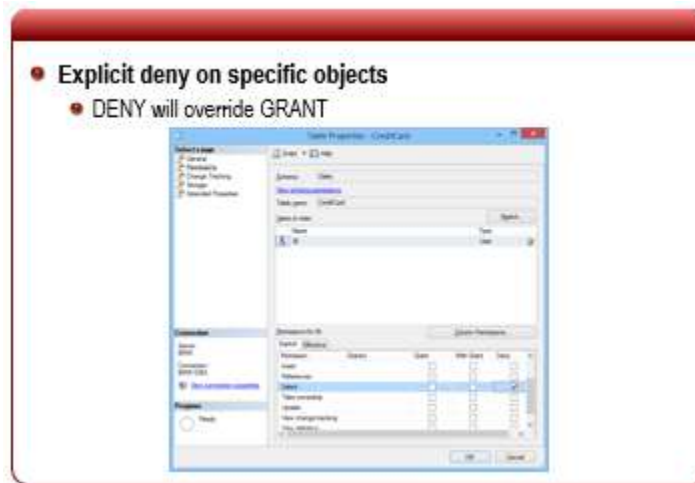
Scenario 2:

After examining a patient, a physician gives a diagnosis and prescribes therapy which may or may not incorporate medications. The prescription is then entered into the Hospital Information System (HIS). Mistakes in this process can produce serious consequences and may even result in the death of a patient. Regardless of the outcome, medical personnel are highly “motivated” to falsify data in the information system.

Consequences: From this point, there is no limitation and we are relaying on the medical personnel’s sense of morality and ethics. In this case, it is not ethical.

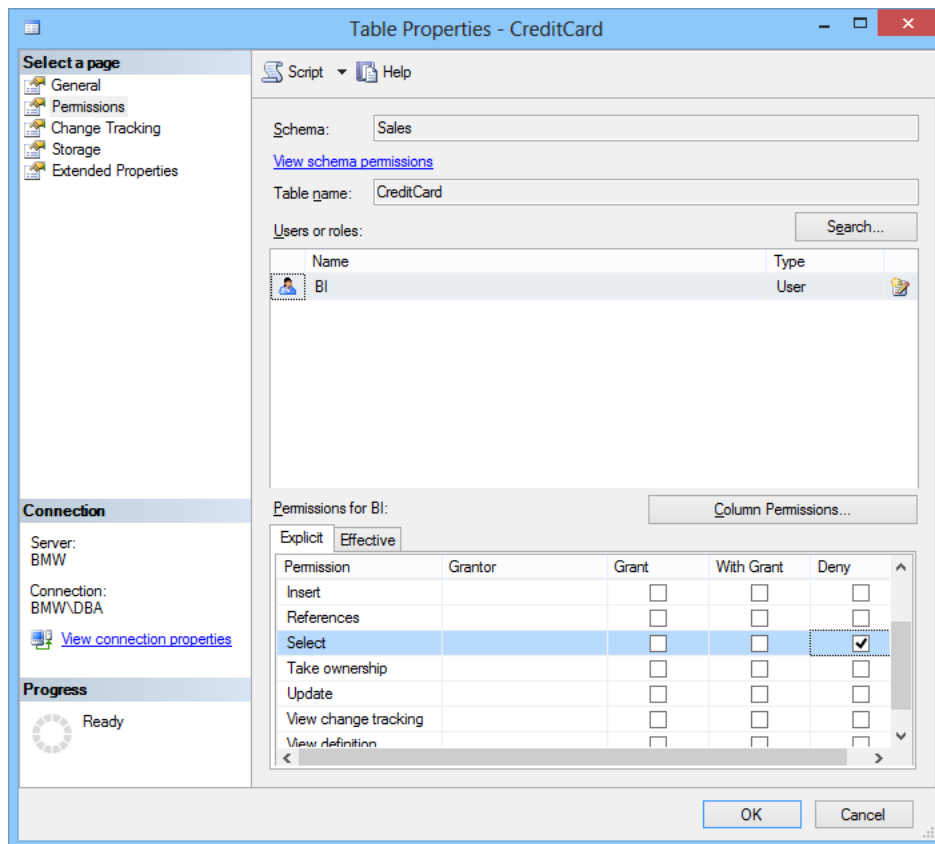
These two examples illustrate the kind of threats that can be posed by authorized users.

Countermeasures



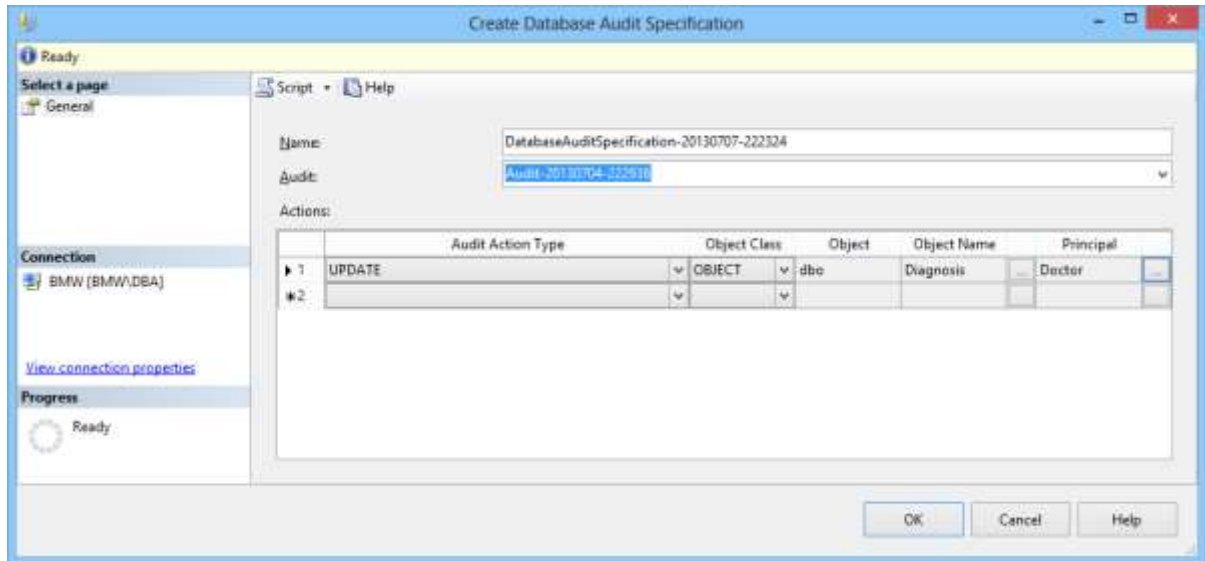
Scenario 1 Countermeasures:

This situation will be fixed by selecting the DENY check box on the Sales.CreditCard table. Yes, the information workers have SELECT on the whole schema, but DENY is always evaluated first.

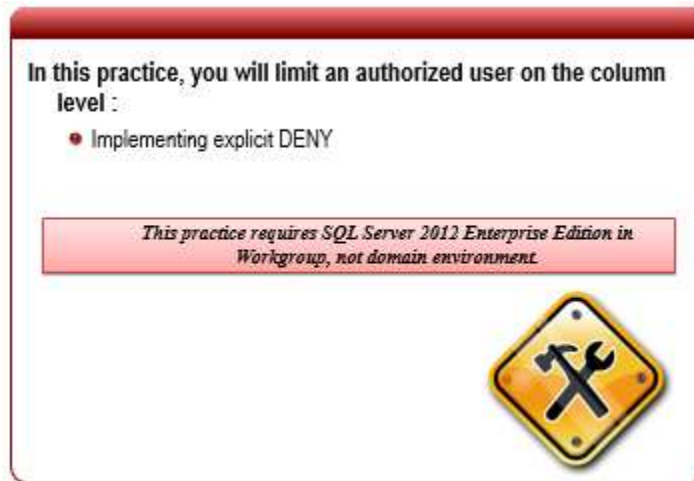


Scenario 2 Countermeasures:

This situation can be fixed with a trigger-based auditing environment or by using the new SQL Server Audit feature. In both cases, you should consider keeping your digital evidence in more than one location. One of the locations should be remote and protected by encryption elements.



Practice: Limiting Threats from an Authorized User



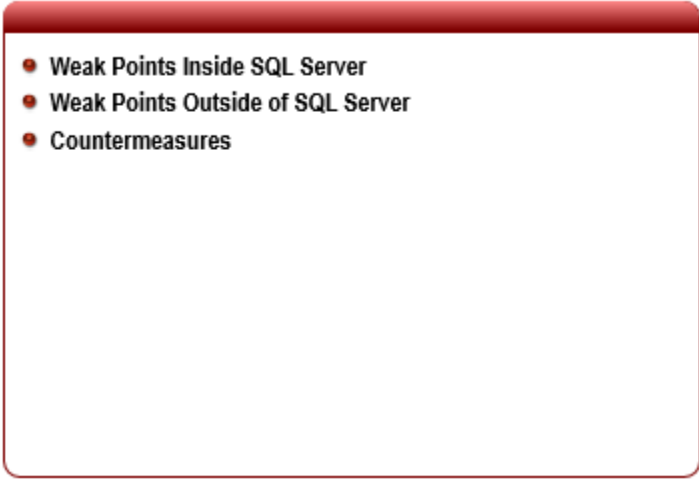
In this practice, you will limit an authorized user on the column level.

This practice requires SQL Server 2012 Enterprise Edition in Workgroup, not domain environment.

Exercise 1: Implementing explicit DENY

1. In **AdventureWorks2012**, create a **BI** user without login.
2. Grant **SELECT** permission on **Sales** schema.
3. Right-click the **CreditCard** table, click **Properties**, and then click **Permissions**.
4. Search and add **BI** user,
5. In the **SELECT** row, select **Deny** under **Column permissions**.
6. Under **Credit Number**, select **Deny**, and click **OK**.
7. Now the BI user can access the table, but the column named **CreditCards** is out of reach.

Lesson 2: Physically Stealing Data

- 
- Weak Points Inside SQL Server
 - Weak Points Outside of SQL Server
 - Countermeasures

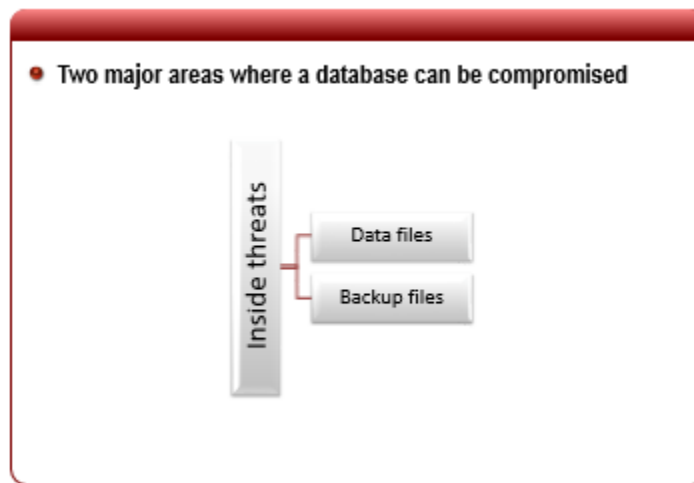
Protecting data in a working environment is an important thing to do. But protecting data from physical theft is also imperative. This is because all your security measures are almost completely useless when someone physically steals your database.

Objectives

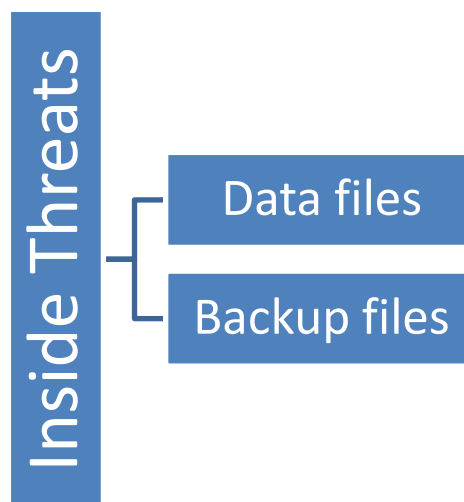
After completing this lesson, you will be able to:

- Identify weak points inside SQL Server
- Identify weak points outside SQL Server
- Implement corresponding countermeasures

Weak Points inside SQL Server

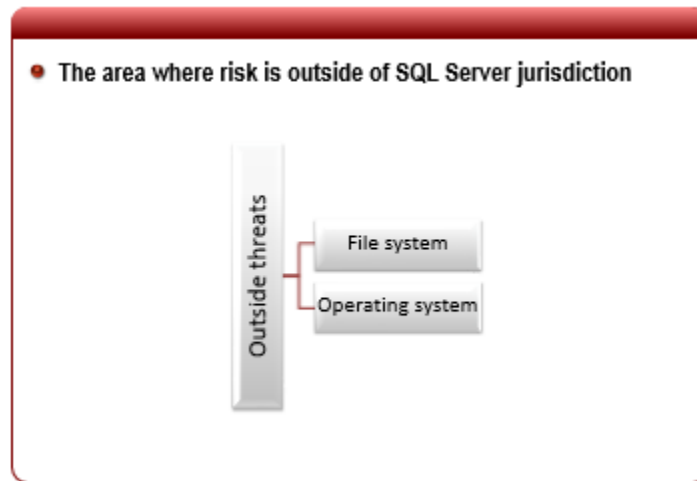


When we look at this problem, you will notice two major areas where a database can be compromised.

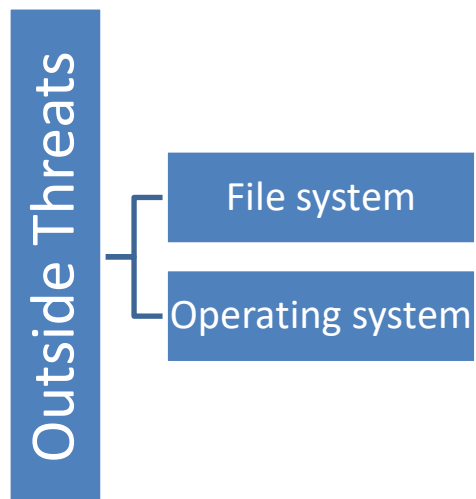


Data and backup files are the weak points inside a database because a user with sufficient access right can physically copy data files on removable media or s/he can copy back-up files. In both cases, this is a major security breach.

Weak Points Outside of SQL Server

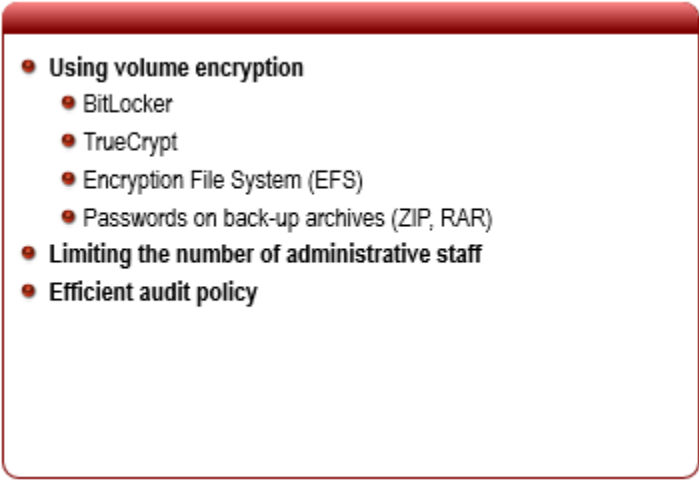


Outside threats refer to the area where risk is outside of SQL Server jurisdiction. But these threats require the attention of database administrators or IT security staff.



File and operating systems are connected and SQL Server cannot control access from this point in an efficient way because SQL Server depends on the operating system. Administrators and power users present greater risk because of their high access rights.

Countermeasures

- 
- Using volume encryption
 - BitLocker
 - TrueCrypt
 - Encryption File System (EFS)
 - Passwords on back-up archives (ZIP, RAR)
 - Limiting the number of administrative staff
 - Efficient audit policy

Weak Points inside SQL Server Solutions

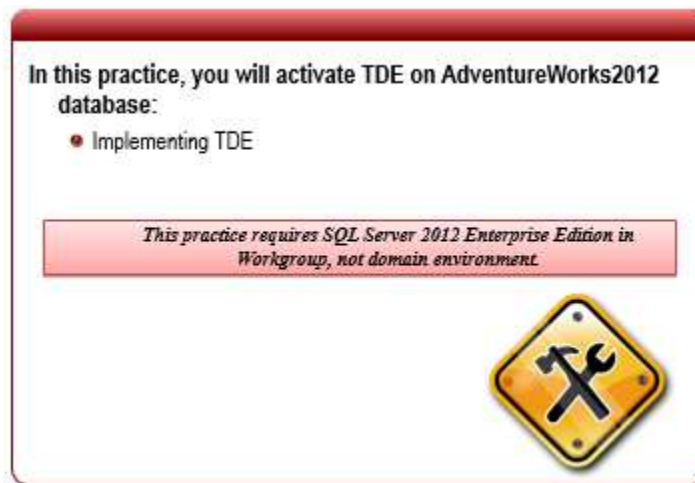
In this case, SQL Server can respond with Transparent Data Encryption (TDE), a feature explained in Module 4, Lesson 3. Encryption is the only way to protect your data from theft. However, not just any encryption will protect your data from theft; only TDE can prevent this. Certainly, symmetric and asymmetric keys can be used to protect data in the tables, but that is usually not sufficient.

Weak Points outside SQL Server Solutions

All potential threats should be identified by using best practices and relying on a good threat modeling phase. The following can be used to mitigate outside threats:

- Using volume encryption
 - BitLocker
 - TrueCrypt
 - Encryption File System (EFS)
 - Passwords on back-up archives (ZIP, RAR)
- Limiting the number of administrative staff
- Efficient audit policy

Practice: Protecting Database Back-up



In this practice, you will activate TDE on AdventureWorks2012 database.

This practice requires SQL Server 2012 Enterprise Edition in Workgroup, not domain environment.

Exercise 1: Implementing TDE

In this exercise, you will pass all necessary steps for activating TDE on AdventureWorks2012 sample database, create a back-up and restore in another location.

1. Create **Master Encryption Key**.

```
USE master;  
GO  
CREATE MASTER KEY  
ENCRYPTION BY PASSWORD = 'Some3xtr4Passw00rd';  
GO
```

2. Create a certificate and verify it.

```
USE master  
GO  
CREATE CERTIFICATE TDE WITH SUBJECT = 'My TDE Certificate';  
GO  
  
SELECT * FROM sys.certificates
```

3. Create a demo database named **TDEdb** and **Database Encryption Key**. The key is encrypted with the certificate created from step 2.

```
CREATE DATABASE TDEdb  
GO
```

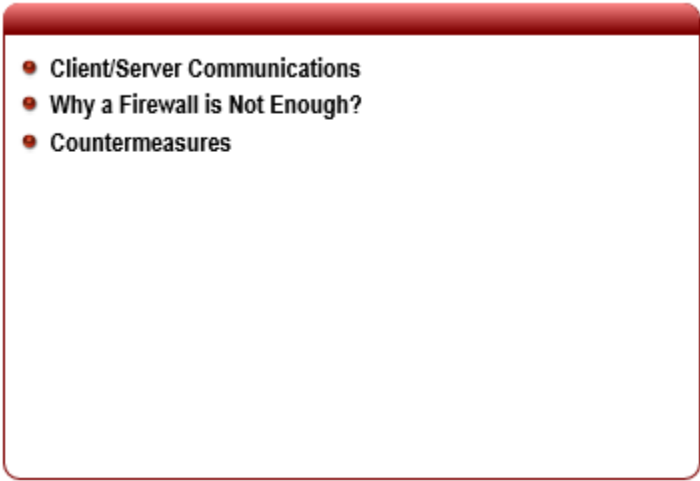
```
USE TDEdb
GO
CREATE DATABASE ENCRYPTION KEY
WITH ALGORITHM = AES_256
ENCRYPTION BY SERVER CERTIFICATE TDE;
GO
```

4. Turn on the **TDE** feature.

```
ALTER DATABASE TDEdb
SET ENCRYPTION ON;
GO
```

5. Create a back-up of the **TDEdb** database.
6. Copy the back-up file on another instance or another machine with SQL Server 2012.
7. Try the **RESTORE** operation.
8. Explain the error message.

Lesson 3: Data Transfer Sniffing

- 
- Client/Server Communications
 - Why a Firewall is Not Enough?
 - Countermeasures

In this lesson, you will learn what happens when information goes through communication channels between the client and the server. Data transfer is one of the biggest security issues in the modern IT world.

Objectives

After completing this lesson, you will be able to:

- Understand SQL Server's role in the client/server architecture
- Identify the firewall's role in the whole process
- Implement corresponding countermeasures

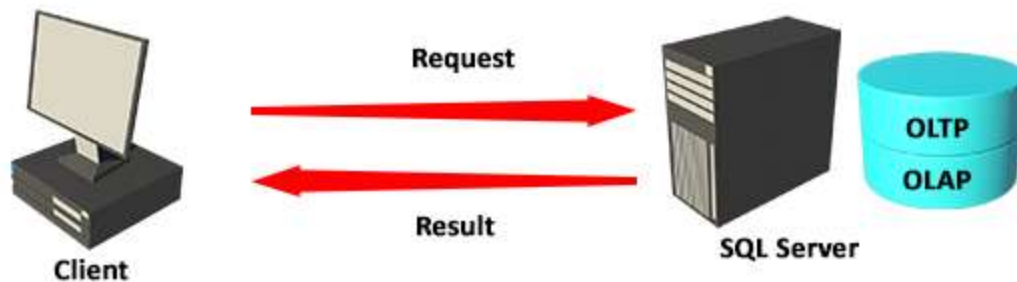
Client/Server Communications

- SQL Server uses classic client/server communication
- Anything can happen when network packets go outside of the SQL Server environment:
 - Communication monitoring
 - Data sniffing
 - Data tampering

SQL Server uses classic client/server communication. The client can be: another SQL Server, custom applications, a mobile device, etc. Regardless of the client, anything can happen when network packets go outside of the SQL Server environment.

- Communication monitoring
- Data sniffing
- Data tampering

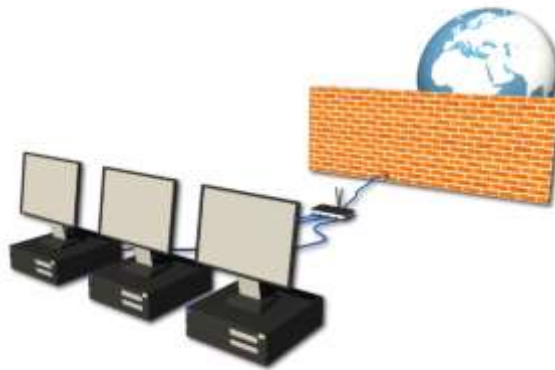
All of these threats can produce severe consequences.



Why a Firewall is not enough?

- Firewall is a necessary but not a sufficient security condition.
- Firewall will NOT help in these situations:
 - Poorly written application
 - Bad data access layer
 - Input validation
 - Etc.

It is often believed that since you have a firewall, you must therefore be secure. This belief is a classic misconception. The purpose of a firewall is to block/allow TCP/IP ports on the client/server side. As an example, we will open port 80 on a web server and close all other ports because they are not needed. Closing ports only minimize the area of potential attack, but we still have one door wide open and that is port 80.



A firewall is a necessary but not a sufficient security condition. A firewall will NOT help in the following situations:

- Poorly written application
- Bad data access layer
- Input validation
- Etc.

Your job requires you to prevent and/or fix any of those situations if they should happen to occur. These threats should be recognized and modeled in the threat modeling phase, and then later implemented in all other phases.

Countermeasures

- Server can use SSL to encrypt data transfer
- SSL encryption with a self-signed certificate is possible
- Encryption level used by SSL is 40-bit or 128-bit
- SSL encryption does slow performance
- SSL connections using a self-signed certificate do not provide strong security

Secure Sockets Layer (SSL)

SQL Server can use Secure Sockets Layer (SSL) to encrypt data in the transfer process across a network between an instance of SQL Server and a client application.

SSL encryption with a self-signed certificate is possible, but a self-signed certificate offers only limited protection.

Encryption level used by SSL is 40-bit or 128-bit depending on the version of the Windows operating system.

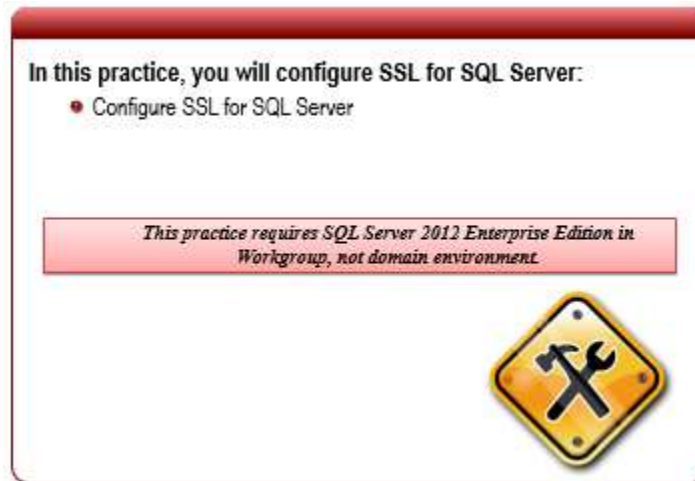
SSL encryption does slow performance.



SSL connections using a self-signed certificate do not provide strong security. They are easy targets to man-in-the-middle attacks. You should avoid using SSL self-signed certificates in a production.



Practice: Encrypt the Connection



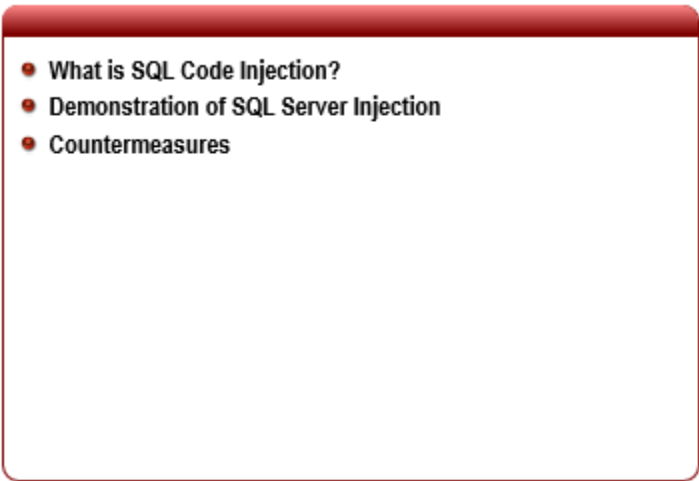
In this practice, you will configure SSL for SQL Server.

This practice requires SQL Server 2012 Enterprise Edition in Workgroup, not domain environment.

Exercise 1: The following procedure describes how to configure SSL for SQL Server.

1. Install a certificate in the Windows certificate store of the server computer.
2. Click **Start**, in the **Microsoft SQL Server** program group, point to **Configuration Tools**, and then click **SQL Server Configuration Manager**.
3. Expand **SQL Server Network Configuration**, right-click the protocols for the server you want, and then click **Properties**.
4. On the **Certificate** tab, configure the Database Engine to use the certificate.
5. On the **Flags** tab, view or specify the protocol encryption option. The login packet will always be encrypted.
 - a. When the **ForceEncryption** option for the Database Engine is set to **Yes**, all client/server communication is encrypted and clients that cannot support encryption are denied access.
 - b. When the **ForceEncryption** option for the Database Engine is set to **No**, encryption can be requested by the client application, but is not required.
 - c. SQL Server must be restarted after you change the **ForceEncryption** setting.

Lesson 4: SQL Code Injection

- 
- What is SQL Code Injection?
 - Demonstration of SQL Server Injection
 - Countermeasures

Input validation is a big security issue. An attacker can discover that your application does not correctly handle the type, length, format, or range of input data. The attacker can then supply carefully created input that can compromise your application.

In a case where the network and host-level of a system are fully secured, the public interface of your application is the only area of attack. The input to your application is a way to execute an attacker's code. Bad input validation is one of the dangerous issues surrounding SQL Injection.

Objectives

After completing this lesson, you will be able to:

- Understand SQL Injection
- Implement and test SQL Injection
- Implement corresponding countermeasures

What is SQL Code Injection?

- SQL injection attack exploits vulnerabilities in input validation
- Occur when your application uses input to construct dynamic SQL statements to access the database
- Using the SQL injection attack, the attacker can execute custom commands in the database

A SQL injection attack exploits vulnerabilities in input validation to run arbitrary commands in the database. It can occur when your application uses input to construct dynamic SQL statements to access the database.

It can also happen if your code uses stored procedures that are passed strings with unfiltered user input. Using the SQL injection attack, the attacker can execute custom commands in the database.

The issue can be serious if the application uses a high-privileged account to connect to the database. In this instance, it is possible to use the database server to run operating system commands and potentially compromise other servers.

Demonstration of SQL Server Injection

```
SqlDataAdapter myCommand = new SqlDataAdapter(
    "SELECT * FROM Users
    WHERE UserName='" + txtuid.Text + "'", conn);

; DROP TABLE Customers --
SELECT * FROM Users WHERE UserName='';
DROP TABLE Customers --'
```

Your application may be susceptible to SQL injection attacks when you incorporate invalidated user input into database queries. Particularly susceptible is code that constructs dynamic SQL statements with unfiltered user input.

```
SqlDataAdapter myCommand = new SqlDataAdapter(
    "SELECT * FROM Users
    WHERE UserName='" + txtuid.Text + "'", conn);
```

Attackers can inject SQL code by terminating the intended SQL statement with the single quote character followed by a semicolon character to begin a new command, and then executing the command of their choice.

```
' ; DROP TABLE Customers --
```

This results in a statement submitted to the database for execution.

```
SELECT * FROM Users WHERE UserName='';
DROP TABLE Customers --'
```

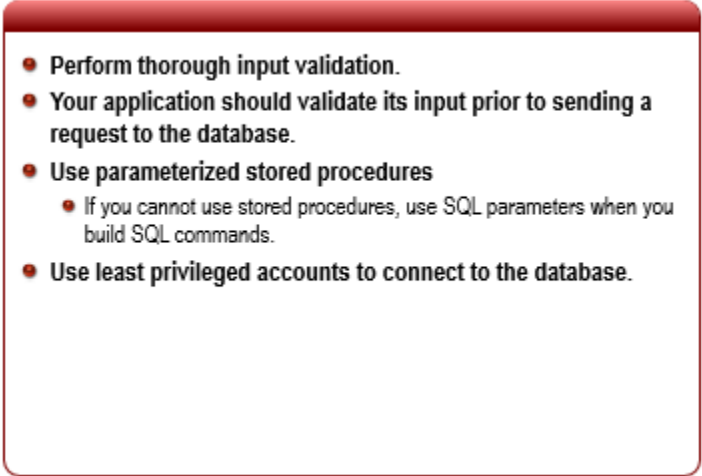
This deletes the “Customers” table, assuming that the application's login has sufficient permission. The double dash (--) denotes a SQL comment and is used to comment out any other characters added by the programmer, such as the trailing quote.

Or you can enter input to the **txtuid** field:

```
' OR 1=1 -
/*Construct following SQL statement:*/
SELECT * FROM Users WHERE UserName='' OR 1=1 --

/*
Because 1=1 is always true, the attacker retrieves every row
of data from the Users table.*/
```

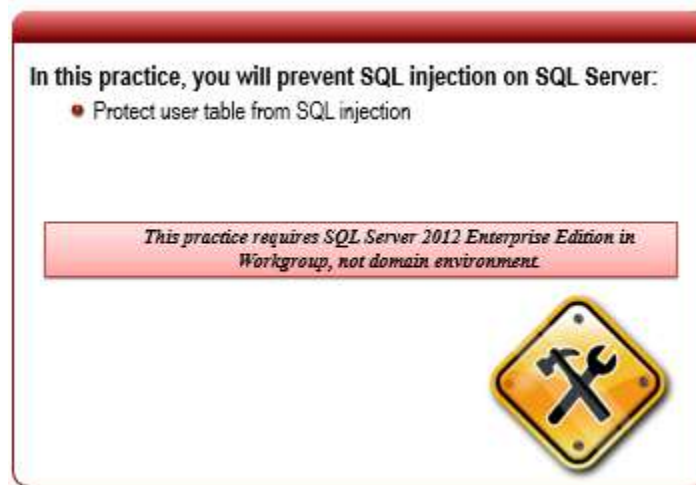

Countermeasures

- 
- Perform thorough input validation.
 - Your application should validate its input prior to sending a request to the database.
 - Use parameterized stored procedures
 - If you cannot use stored procedures, use SQL parameters when you build SQL commands.
 - Use least privileged accounts to connect to the database.

Countermeasures to prevent SQL injection include:

- Perform thorough input validation. Your application should validate its input prior to sending a request to the database.
- Use parameterized stored procedures for database access to ensure that input strings are not treated as executable statements.
- If you cannot use stored procedures, use SQL parameters when you build SQL commands.
- Use least privileged accounts to connect to the database.

Practice: Protecting from SQL Server Injection



In this practice, you will prevent SQL injection on SQL Server.

This practice requires SQL Server 2012 Enterprise Edition in Workgroup, not domain environment.

Exercise 1: The following procedure describes how to protect SQL injection on a user table.

1. Open **AdventureWorks2012** sample database.
2. We will create a **CRUD** (Create, Read, Update and Delete) procedure for the **HumanResources.Department** table
3. In the **New Query Windows**, type the following T-SQL code:

```
USE [AdventureWorks2012] ;
GO

CREATE PROC [HumanResources].[usp_DepartmentSelect]
    @DepartmentID SMALLINT
AS

    BEGIN TRAN

        SELECT [DepartmentID], [Name], [GroupName],
            [ModifiedDate]
        FROM [HumanResources].[Department]
        WHERE ([DepartmentID] = @DepartmentID
OR @DepartmentID IS NULL)

    COMMIT

GO

CREATE PROC [HumanResources].[usp_DepartmentInsert]
```

```
@Name name,
@GroupName name,
@ModifiedDate datetime
AS

BEGIN TRAN

INSERT INTO [HumanResources].[Department] ([Name],
[GroupName], [ModifiedDate])
SELECT @Name, @GroupName, @ModifiedDate

SELECT [DepartmentID], [Name], [GroupName],
[ModifiedDate]
FROM [HumanResources].[Department]
WHERE [DepartmentID] = SCOPE_IDENTITY()

COMMIT

GO

CREATE PROC [HumanResources].[usp_DepartmentUpdate]
@DepartmentID smallint,
@Name name,
@GroupName name,
@ModifiedDate datetime
AS

BEGIN TRAN

UPDATE [HumanResources].[Department]
SET [Name] = @Name, [GroupName] = @GroupName,
[ModifiedDate] = @ModifiedDate
WHERE [DepartmentID] = @DepartmentID

SELECT [DepartmentID], [Name], [GroupName],
[ModifiedDate]
FROM [HumanResources].[Department]
WHERE [DepartmentID] = @DepartmentID

GO

CREATE PROC [HumanResources].[usp_DepartmentDelete]
```

```
        @DepartmentID smallint
AS

        BEGIN TRAN

        DELETE
        FROM    [HumanResources].[Department]
        WHERE   [DepartmentID] = @DepartmentID

        COMMIT

GO
```

Summary

In this module, you learned how to:

- Identify threats from authorized users
- Minimize the risk from physical data theft
- Prevent data transfer sniffing
- Avoid SQL code injection

In this final module, you have learned that security, as a concept, is not enough to provide data security and privacy. When we look at SQL Server objects, database and data itself from outside SQL Server, we have noticed that we need more than just authorization/authentication. Also, protected data and back-up files are insecure when security is breached.

Encrypted data in a database is useless when a client requests some data, specifically because the requested data needs to be decrypted and transmitted over insecure network channels.

And finally, the connection between the development team and the SQL Server administration team is a critical point. SQL injection is one those examples where communication between them is not at an appropriate level.

Remember, security is an ongoing process, not a final state.

Objectives

After completing this module, you learned:

- How to identify threats from authorized users
- How to minimize the risk from physical data theft
- How to prevent data transfer sniffing
- How to avoid SQL code injection