Covers SQL Server 2016

THE EXPERT'S VOICE® IN SQL

Securing SQL Server

DBAs Defending the Database

Peter A. Carter

Apress®

Securing SQL Server

DBAs Defending the Database

Peter A. Carter

Apress®

Securing SQL Server: DBAs Defending the Database

Peter A. Carter
Botley, United Kingdom

ISBN-13 (pbk): 978-1-4842-2264-5 ISBN-13 (electronic): 978-1-4842-2265-2
DOI 10.1007/978-1-4842-2265-2

Library of Congress Control Number: 2016956880

Managing Director: Welmoed Spahr
Lead Editor: Jonathan Gennick
Development Editor: Laura Berendson
Technical Reviewer: Bradley Beard
Editorial Board: Steve Anglin, Pramila Balan, Laura Berendson, Aaron Black, Louise Corrigan, Jonathan Gennick, Todd Green, Robert Hutchinson, Celestin Suresh John, Nikhil Karkal, James Markham, Susan McDermott, Matthew Moodie, Natalie Pao, Gwenan Spearing
Coordinating Editor: Jill Balzano
Copy Editor: Kim Burton-Weisman
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springer.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a Delaware corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at www.apress.com/bulk-sales.

Any source code or other supplementary materials referenced by the author in this text are available to readers at www.apress.com. For detailed information about how to locate your book's source code, go to www.apress.com/source-code/. Readers can also access source code at SpringerLink in the Supplementary Material section for each chapter.

Printed on acid-free paper

This book is dedicated to my loving wife, Danielle.

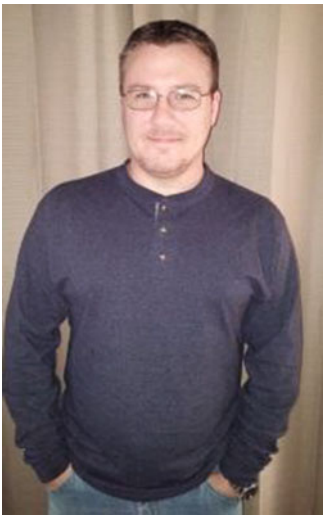
Contents at a Glance

Contents

About the Author

Peter Carter is a SQL Server expert with over a decade of experience in developing, administering, and architecting SQL Server platforms, data-tier applications, and ETL solutions. Peter has a passion for SQL Server and hopes that his enthusiasm for this technology helps or inspires others.

About the Technical Reviewer



Bradley Beard is a software engineer with more than 15 years' experience writing dynamic, interactive websites using ColdFusion and SQL Server. He graduated from Florida Institute of Technology in 2007 with a Master of Science in Computer Information Systems, and studied for his undergraduate degrees in CIS and Technology Management at Herzing University. In 2013, he earned the MCSA: SQL Server 2012 certification from Microsoft, and in 2016, he earned the MCSE: Business Intelligence certification. His continual quest for learning has earned him shelves full of books at home and at work, most of which are about SQL Server, ColdFusion, or general web architectures or frameworks.

He lives in Palm Bay, Florida, with his wife, Jessica, and children, Josh, Kaylee, Matthew, and Emma. He also apparently runs an animal shelter made up of his dogs—Lady and Bella, and his cats—Spice, Simba, Mercury, and Dobby. In his free time, he enjoys fishing and spending time with his wife and kids.

Bradley is available for consultation and third-shift remote employment on ColdFusion and SQL Server by contacting `bradley.beard@gmail.com`.

Acknowledgments

I would like to thank SplashData (www.splashdata.com) for kindly allowing me to use their “Worst password list of 2015” within this book.

Introduction

With repeated, high-profile, data security breaches hitting the headlines, security is moving increasingly to the forefront of the minds of data professionals.

SQL Server provides a broad and deep set of security features that allow you to reduce the attack surface of your SQL Server instance with defense-in-depth and principle of least privilege strategies.

The attack surface of SQL Server refers to the set of features and windows services that attackers can (and will) attempt to exploit to either steal data or reduce the availability of the data.

Defense-in-depth is a strategy used across the IT industry in which multiple layers of security are put in place. The idea is that if one layer of security is breached, then another layer will stop the attacker in their tracks.

In order to fully protect data against attack, SQL Server DBAs, developers, and architects alike must all understand how and when to implement each of the security features that SQL Server offers. This book attempts to address these topics.

Some of the examples in this book use the AdventureWorks2016 database. This database can be downloaded from `www.microsoft.com/en-us/download/details.aspx?id=49502`.

Some chapters also refer to CarterSecureSafe. This is a fictional company and product, which are purely designed to illustrate points made within this book.

CHAPTER 1

Threat Analysis

We live in an age where high-profile attacks on data are almost commonplace. Attacks can come from a variety of sources, ranging from cyberterrorism and modern warfare to industrial espionage, the "geek" factor, organized crime, and even disgruntled employees or former employees. Because of this, security is at the forefront of every good DBA's minds.

All RDBMS have the potential to be exploited with SQL injection attacks, as well as vulnerabilities that are unique to each product. For example, attackers often attempt to gain elevated access to Oracle by attempting to use default user passwords. While this risk can be mitigated with due diligence, with around 600 default user/passwords, it can be hard for Oracle DBAs to ensure that no stone is left unturned.

In SQL Server, a common attack is to attempt to brute-force attack the `sa` account, on port 1433. While the `sa` account can be disabled or have its name changed, the majority of SQL Server DBAs do not do this, and in many cases, there are poorly written client applications that require an `sa` account to be present.

Understanding Threat Modelling

Because every database management platform is vulnerable to many potential threats, it is important to undergo a process of *threat modelling* in order to mitigate the risks. Threat modelling is the process of identifying threats to a data-tier application (or in some instances, the entire enterprise) and then classify and rate the threats that have been discovered in order to determine the most critical to address. You are then in a position to determine the correct countermeasures in order to mitigate the risks.

In an ideal world, threat modelling should be carried out during the design phase of a project, and at the very least, at the testing stage. There will already be standards and policies in place for the enterprise as a whole, and you can ensure that the platform you are constructing meets these standards.

In the real world, however, this often does not happen due to time or budgetary constraints. Often, there are also no enterprise standards, specifically for database platforms, against which you can baseline your data tier. Unfortunately, just like comprehensive backup strategies, many companies and individuals do not put an emphasis on security until it is too late.

Electronic supplementary material The online version of this chapter (doi:10.1007/978-1-4842-2265-2_1) contains supplementary material, which is available to authorized users.

P. A. Carter, *Securing SQL Server*, DOI 10.1007/978-1-4842-2265-2_1

Even in companies that have rigorous security management policies, the focus tends to be avoiding external attacks (attacks from sources external to the company); whereas it is estimated that 70 percent of security breaches are internal (attacks originating from sources within the company network). This is due to employees with malicious intent, employees who unintentionally misuse systems, and also from the theft of employees' laptops or other devices. Therefore, it is important that companies focus on identifying the risks of attacks from inside their network, as well as outside.

Threat modelling consists of six sequential steps: identifying assets, creating an architecture overview, building a security profile, identifying the threats, documenting the threats, and finally, rating and prioritizing the threats.

■ **Tip** Threat modelling allows you to design and build countermeasures. Building the countermeasures is not part of the threat modelling process, however. Instead, the countermeasures are implemented as a separate process.

The following sections discuss how to perform threat analysis by using a fictional application called CarterSecureSafe, which belongs to the fictional company, CarterSecurityTools.com. The simple web application allows customers to shop for security software. The back end of the web application is a database hosted in a SQL Server instance.

Identifying Assets

The first step in the threat modelling process is to identify valuable assets. From the perspective of the DBA, identifying the valuable assets that must be protected consists of identifying the company's confidential information, which would have a commercial impact if it were lost (unavailable) or stolen. For example, a high-profile attack against an entertainment company reportedly saw the theft of roughly 76 million user accounts, leading to a cost of around $176 million.

DBAs should look to ensure that customer data, financial data, and sales data are especially secure. Remember that financial repercussions could occur, not just in tangible ways, such as through fines from regulators or in compensation to customers, but also in intangible ways, such as the loss of business reputation, reduced staff morale, or customers moving their business to a rival.

Creating an Architecture Overview

Creating an architecture overview consists of defining the logical architecture of the application and expressing it in diagrammatic form, along with the technology stack that will be (or has already been) used, to implement the application. This helps you identify areas of the end-to-end application that are potentially vulnerable, as well as identify any technology-specific vulnerabilities.

Creating the Infrastructure Components

In the case of CarterSecureSafe, the application consists of a web server, an application server, and a database server. You should also note how this architecture interacts with the underlying infrastructure. The diagram in Figure 1-1 shows how an architecture diagram for the CarterSecureSafe application might look.

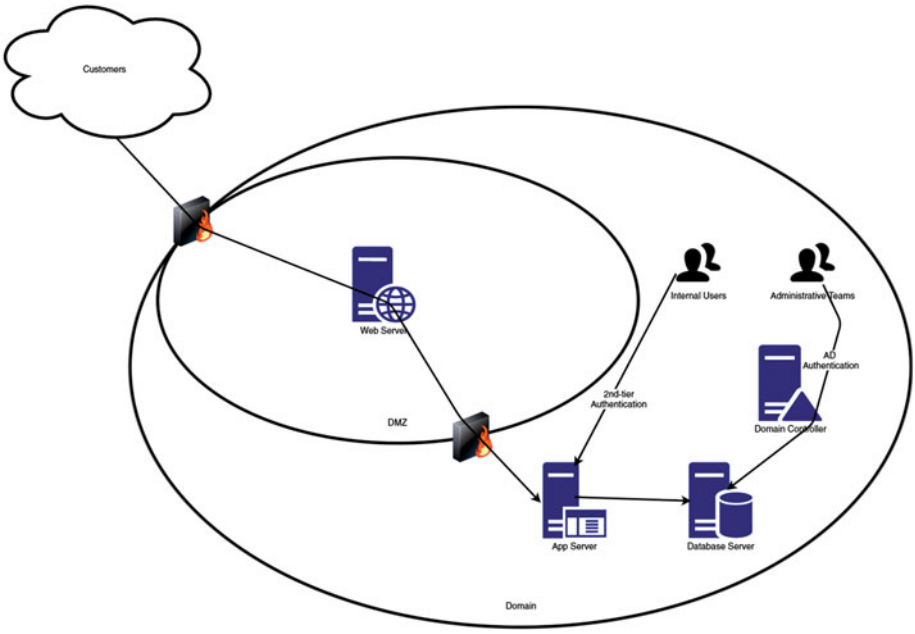


Figure 1-1. *Architecture diagram*

■ **Tip** In a real architecture diagram, you label servers with their name and IP address, rather than with a description of their usage.

The diagram shows that the application is accessed by both internal and external users. Internal users authenticate to the application server through Active Directory, while external users authenticate through a web server located in the company's DMZ (demilitarized zone).

■ **Note** As well as indicating the servers that are directly used by the application (web server, application server, and database server), infrastructure touch points are also included; namely, the corporate firewalls that traffic passes through, the DC (domain controller) used to authenticate internal users, and the isolated DMZ within the domain.

Identifying the Technology Stack

Let's now list out the technology used for each area of the topology. Table 1-1 demonstrates how this should look for the CarterSecureSafe application.

Table 1-1. *Technology Implementations*

Technology	Topology Component	Details
Active Directory	Authentication	Authenticates internal users and administrative teams.
IIS	Web Server	Authenticates external users and pass traffic to the application server.
.NET	Core Application & Authentication	The core web application built using the .NET framework. It also provides Forms authentication for internal users.
SQL Server 2016	Database Tier	The databases that drive the application are stored and managed on a SQL Server 2016 instance.
IPsec	Cryptography	Data is encrypted in transit, between the application and database server, using IPsec.
HTTPS	Protocol	External users access the web application via HTTPS.

For a DBA, it is very easy and natural to focus entirely on the SQL Server instance and its direct connections; but it is also important to understand the holistic application and platform in order to secure and test the data-tier application appropriately.

Creating a Security Profile

When creating a security profile, you begin to identify data flows, which, in turn, allow you to define trust boundaries and entry points. The CarterSecureSafe application is a simple solution that has two distinct flows of data.

The first of these flows happens when an Internet user orders an item from the store. The second flow of data comes from internal users, who need to update the status of customer's orders and perform other administrative sales and management tasks, such as reporting on sales trends.

Therefore, there are two clear data paths. The first is from the Internet via the web server and then through the application server to the SQL Server instance. The second is via the application server into the SQL Server instance, but originating from within the internal network.

■ **Tip** The CarterSecureSafe application is very simple, but for more complex data paths, you probably want to create data path diagrams to simplify the process and ensure that there are no gaps. The data path diagram also serves as documentation that is useful on an ongoing basis, such as when new team members are getting up to speed, or when the application is due to be upgraded or migrated.

The entry points that align to data paths can be identified as the web server (for Internet users) and the application server (for internal users). It is important to remember that there is a third entry point that is easy to overlook: internal users authenticating directly to the SQL Server instance.

Of course, this final entry point is intended for the use of DBAs to manage the instance and its databases, but it is important to remember that around 70 percent of security breaches are from internal sources.

The trust boundaries for the CarterSecureSafe application map to the firewalls. The data path from Internet users crosses both the perimeter and internal firewalls; whereas the internal data path remains within the internal trust boundary.

Now that the application has been decomposed, you can begin to build a security profile. From the DBA perspective, this will involve focusing on the elements that directly interface with the database. This profile can then be fed into the overall security profile of the application. Table 1-2 provides an example of how a security profile may look for the CarterSecureSafe application.

Table 1-2. *Security Profile*

Profile Element	Considerations
Input Validation	The application runs ad hoc T-SQL, as opposed to calling stored procedures. Therefore, the input cannot easily be validated at the SQL Server level. *
	As the main entry point is the web server, trust boundaries are crossed and the input cannot be trusted.
Authentication	Users authenticate to the database engine via second tier authentication. No domain authentication is required to access the databases.
	Penetration testing to ensure that the `sa` account has been either disabled or renamed, has not been carried out on the instance.
	The application server resolves user credentials. The application server uses a single user to authenticate to the database engine.

(*continued*)

Table 1-2. (*continued*)

Profile Element	Considerations
Cryptography	Data is encrypted in transit using IPsec.
	Databases are not encrypted using TDE (Transparent Data Encryption)
	No column level encryption is used.
Auditing	SQL Audit has not been configured, however, the default trace is running, which will capture a limited subset of activity, such as creating new objects.

**There may be (and should be) input validation on the application side, but the DBA is unlikely to have visibility of this*

Identifying Threats

Now that a security profile is in place, you can work to identify potential threats in the application. This usually involves performing a *penetration test*.

■ **Tip** A penetration test, also known as a *pen test*, involves scanning a solution (or in some cases, an enterprise) in an attempt to find vulnerabilities that could be exploited by attackers.

Understanding STRIDE

There are many penetration testing tools available, including Qualys, which can be obtained from `www.qualys.com` and Metasploit, which can be obtained from `www.metasploit.com`. This book uses Kali Linux, which can be downloaded from `https://www.kali.org/downloads/`.

The threats that are revealed by the penetration test can then be categorized using STRIDE methodology. STRIDE stands for

- **S**poofing identity
- **T**ampering with data
- **R**epudiation
- **I**nformation disclosure
- **D**enial of service (DoS)
- **E**levation of privileges

Spoofing identity refers to stealing another user's identity and using this identity to authenticate rather than using your own identity. The CarterSecureSafe application is particularly susceptible to this because the application server uses a single user to authenticate to the instance and because inputs cannot feasibly be validated at the database tier.

Tampering with data refers to the practice of maliciously modifying data. In the context of the overall application, this could refer to attacks such as cross-site scripting and manipulating HTTP headers. From the DBA perspective, however, it refers to maliciously modifying data stored within the database. For example, in the case of the CarterSecureSafe application, a malicious user may attempt to amend their account balance to zero.

Repudiation describes a malicious user's ability to hide or deny their activity. This is critical, because if repudiation is possible, you may not be aware that an attack has even taken place. If you are aware that security has been breached, it may be impossible to prove. Repudiation is an issue for the CarterSecureSafe application, because SQL Audit has not been implemented. This means that the only actions that are captured are those by the default trace, such as new object creation.

Information disclosure is the classification of threat that springs to most people's minds when they think of hacking. It refers to data being "stolen." Data theft occurs when an attacker forces a system to reveal more data than they have the permissions to see. As with spoofing identities and tampering with data, the CarterSecureSafe application is susceptible to this form of attack because the database layer does not validate inputs.

Denial-of-service (DoS) attacks occur when an attacker attempts to flood a system with so many requests that they either take down the system or make the system appear to be down due to its inability to deal with the volume of requests being received. DoS are among the most common forms of attacks and they are becoming increasingly sophisticated. This means that you should always take them into account during every threat modelling exercise.

Elevation of privileges refers to the act of exploiting a system to gain more permissions than you were intended to have. The fact that the security profile has revealed that penetration testing has not taken place around the `sa` account means that the CarterSecureSafe application is susceptible to this kind of attack.

As with all relational database management systems, SQL Server has known vulnerabilities, which can be exploited. These should be addressed wherever possible, usually through patching the system. If no patching is currently available, then at a minimum, you should consider implementing auditing and alerting, specifically tailored to the vulnerability.

Using STRIDE

You should document the potential threats against the application. I recommend using a table similar to Table 1-3.

Table 1-3. *STRIDE Classification*

Risk	Category	Example
SQL injection	S, T, I	Attacker types `' OR 1=1--` in password field of the web site to spoof the first user identity stored in the users table.
DoS	D	Attacker uses robots to simultaneously flood the database with resource intensive requests.
Stealing `sa` account credentials	E	An attacker suspects that the `sa` account has not been disabled or renamed. Therefore, an attack is launched against the password of the `sa` account.
DBA performs malicious action	R	A privileged user performs a malicious action and the attack cannot be proven due to lack of auditing.
SQL Server Remote Code Execution Vulnerability*	S, T	An attacker runs a malicious query to exploit a vulnerability in SQL Server, where the use of uninitialized memory in some virtual functions is permitted.

**At the time of writing, Microsoft has not released any security bulletins relating to SQL Server 2016. The vulnerability used as an example applies to SQL Server versions 2008–2014.*

■ **Note** While this type of attack sounds a little farfetched, it is more common than you may think. I am aware of two separate companies that have fallen foul of this in recent times. In one instance, on a DBA's last day, he dropped a key database. In the other instance, a SQL Server DBA obfuscated all stored procedures before leaving the company.

Rating Threats

Once threats have been identified and classified, you should begin the process of rating these threats based upon the probability of the attack occurring and compared to the damage that could be inflicted if the threat was realized. There are various methodologies used for rating threats.

Understanding Threat Rating Methodologies

The simplest method for rating threats is a straight High/Medium/Low system. With this system, each threat is given a rating based on your opinion. There are two issues with this approach, however. First, it makes the rating system subjective, as opinions are opinions only and are not necessarily correct. Second, opinions often differ, therefore it can be hard to gain a consensus on the priority in which the threats should be addressed.

A slightly more scientific approach is to use a Critical/Important/Moderate/Low system. This system offers more categories, which can aid prioritization where there are a large number of threats. A *critical* threat is usually defined as a threat that allows an attacker to penetrate a system without any alerts or warnings being fired and where there is precedence of this attack being performed.

An *important* threat is usually regarded as a threat where data could be compromised by an attacker and it would be easy for an attacker to exploit the vulnerability if it was discovered. With threats in this category, there is often precedence for similar vulnerabilities being exploited.

A *moderate* threat is categorized as a threat where it is possible for an attacker to exploit the vulnerability; however, the risk is mitigated by factors such as integrated authentication, which would be difficult for an attacker to exploit the weakness.

A *low* threat is normally regarded as one where the likelihood of the vulnerability being exploited is very low due to existing infrastructure or countermeasures that are in place. Often, threats that are categorized as low are not addressed, as it has been decided that the cost of addressing them outweighs the potential costs of the attack being exploited.

■ **Caution** While pragmatically ignoring a threat with a low rating is sensible, because we all understand that budgets and timescales are always important factors, I do like to remind management and budget holders of the analogy involving the Fukushima nuclear disaster in 2011. The risk analysis when building this plant reportedly factored in protection against an earthquake and protection against a tsunami. The risk of two earthquakes and a tsunami occurring at the same time, however, was regarded unlikely to require consideration. The first earthquake was within the designed tolerance of the reactors, but following the second earthquake and tsunami, the Fukushima plant largely melted within three days.

A damage potential × probability formula is another common system for determining threat ratings. Using this technique, you rate the damage potential of each threat using a scale of 1 to 10, where 1 means that an attack exploiting this particular vulnerability would cause only minimal damage and 10 indicates that an attack exploiting the particular vulnerability would be a catastrophe.

You then rate the likelihood of the threat being realized on a scale of 1 to 10. Here, 1 indicates that there is very little chance of the threat being realized and 10 means that it is almost certain. Once the two ratings for each threat have been established, you multiply the damage potential rating by the probability rating for each threat. This gives your threats a priority score on a scale of 1 to 100.

Understanding DREAD Methodology

My preference for rating threats is to use a methodology known as DREAD. Although it is not often used in recent times, with many favoring the simpler methodologies, I find it the best and most comprehensive fit for data-tier applications.

DREAD stands for

- **D**amage potential
- **R**eproducibility
- **E**xploitability
- **A**ffected users
- **D**iscoverability

Damage potential rates the damage potential of each threat using a scale of 1 to 10, where 1 means that an attack exploiting this particular vulnerability would cause only minimal damage and 10 indicates that an attack exploiting the particular vulnerability would be a catastrophe.

Reproducibility rates how easy it would be for an attacker to repeatedly reproduce the attack on a scale of 1 to 10. 1 indicates that is would be almost impossible to reproduce and 10 means that it would be very easy to reproduce the attack. The easier it is to reproduce an attack, the greater the likelihood there is for automated attacks (using bots) that systematically exploit the vulnerability.

Exploitability rates the ease in which an attack could exploit the vulnerability, using a scale of 1 to 10. 1 indicates that the vulnerability would be extremely difficult to exploit due to factors such as domain authentication being required. A rating of 10 indicates that an attacker could exploit the vulnerability with ease.

Affected users rates the number of users that would be affected by the threat on a scale of 1 to 10. To calculate the rating, you should take the percentage of users that would be affected, divide this number by 10, and then round to the nearest whole number. For example, if 80 percent of users would be affected, then the rating would be 8. If only 25 percent of users would be affected, then the rating would be 3.

Discoverability rates how easily an attacker to discover the vulnerability on a scale of 1 to 10. A rating of 1 means that the vulnerability is obscure and an attacker would be unlikely to stumble across it or realize its potential. A rating of 10 would indicate that the vulnerability can easily be discovered. For example, it may be a well-known, documented attack strategy, such as SQL Injection.

Using DREAD Methodology

Once each threat has been given a rating in each of the DREAD categories, the ratings should be summed and then divided by the number of threats before being rounded to the nearest whole number. This gives you the overall DREAD rating for each threat. Let's use the threats identified earlier and rate them using DREAD. The risks in Table 1-4 have been ordered by their DREAD rating.

Table 1-4. *DREAD Ratings*

Risk	Category (STRIDE)	D	R	E	A	D	Threat Rating
SQL injection	S, T, I	10	10	9	10	10	10
DoS	D	10	10	10	10	10	10
Stealing sa account credentials	E	6	10	8	10	10	9
DBA performs malicious action	R	10	1	1	10	10	6
SQL Server Remote Code Execution Vulnerability	S, T	8	5	5	6	1	5

You can see that the risk of SQL injection attacks, stealing the sa account password, and denial-of-service attacks should be addressed immediately. The risk of DBAs performing malicious actions and the SQL Server Remote Code Execution Vulnerability being exploited should still be address, but with a lower priority.

Creating Countermeasures

The security modelling is now complete and you should start to consider what countermeasures you can put in place for each of the risks that you have identified, starting with the threats that have the highest DREAD rating.

Mitigating the risk of SQL injection involves validating the inputs received. This should be performed at the application, but it is important to remember that the DBA is the last line of defense against attacks. Therefore, you should review how the application is interacting with the database. You identified that the application is running ad hoc queries and you could reduce the risk of SQL injection attacks by introducing a hosting standard that requires applications to access data within the database using stored procedures, as opposed to ad hoc SQL. You can then ensure that the stored procedure is validating the values passed to its parameters.

Although this causes rework, both on the application tier and the database tier, the code that is currently being executed by the application can be reused inside stored procedures. This approach may also give other advantages, such as increased reuse of execution plans.

The risk of an attacker gaining elevated privileges by attacking the password of the sa account can be mitigated by disabling or renaming the sa account. If the application is legacy or third-party, however, then it may have a hard requirement to use the sa account. If this is the case, then you should, at a minimum, introduce SQL Server Audit to increase reputability, and preferably, a combination of triggers and policy-based management to protect against some common malicious actions.

■ **Tip** SQL Server Audit is discussed in Chapter 3. A full discussion around policy-based management is beyond the scope of this book. Full details of implementing the technology can be found in my book *Pro SQL Server Administration* (Apress, 2015).

Denial-of-service attacks are one of the most difficult to protect against. This goes part way in explaining why they are one of the most common forms of attack. One way to reduce the risk is to ensure that the database server is not placed in the DMZ and is therefore not directly exposed to the Internet. This security best practice is already in place for the CarterSecureSafe application.

You could further reduce the impact of a DoS attack by implementing Resource Governor. This allows you to limit the resources that were consumed by attacker's requests. If the application tier were written using Java EE, then you could also use WebLogic server to reduce network traffic. As the application layer is .NET, however, this approach is not feasible in this case.

■ **Tip** A full discussion of Resource Governor is beyond the scope of this book. A full discussion of the technology and how to implement it can be found in my book *Pro SQL Server Administration* (Apress, 2015).

If a rouge DBA decided to attack the database, then there is very little that you can do to stop it. What you must do, however, is ensure that you have a reputability strategy in place. This involves using SQL Server Audit (discussed in Chapter 3) to ensure that malicious actions are traceable. This serves two purposes. The obvious reason is that you can prove what happened and take appropriate action. Less obvious is the fact that you can prove and take action against a malicious DBA will potentially act as a deterrent. This is known as a *soft security measure*. Processes should also be reviewed to ensure that sensible best practices are being followed, such as disabling user accounts when a staff member leaves. Currently, the CarterSecureSafe application does not have SQL Server Audit implemented. You should consider implementing fine-grain auditing to ensure reputability.

Because the DREAD rating for the SQL Server Remote Code Execution Vulnerability is low, and specifically due to the obscurity of the vulnerability, you (or management) will likely decide that you should not take immediate action to mitigate the risk, as appropriate countermeasures will likely prove to be cost prohibitive compared to the likelihood of the vulnerability being exploited. You should keep the risk logged in your project's risk register and patch the instance as soon as a patch becomes available.

Summary

Threat modelling is not a terribly glamourous task, but it is absolutely essential to creating and maintaining a secure environment. It provides a mechanism for identifying threats and prioritizing the efforts to create countermeasures.

Idealistically, you want to tackle all threats as quickly as possible; however, a pragmatic approach is required. Due to time and/or budgetary constraints, you may need to record some threats in the project's risk register, instead of implementing rigorous countermeasures. In some instances, there may also be other reasons for exceptions, such as the requirements or a legacy application.

STRIDE methodology is an approach for categorizing threats. STRIDE is the acronym for

- Spoofing identity
- Tampering with data
- Repudiation
- Information disclosure
- Denial of service (DoS)
- Elevation of privileges

There are many methodologies that can be used to rating threats. I recommend the use of DREAD methodology. DREAD is an acronym for

- Damage potential
- Reproducibility
- Exploitability
- Affected users
- Discoverability

CHAPTER 2

SQL Server Security Model

An early step in implementing countermeasures to potential threats in SQL Server is to ensure that you have a full understanding of the security model. SQL Server 2016 provides a rich security framework with overlapping layers of security that help database administrators (DBAs) counter risks and threats in a manageable way.

It is important for DBAs to understand the SQL Server security model so that they can implement the technologies in the way that best fits the needs of their organization and applications, while minimizing the amount of security administration that is required. This chapter discusses the implementation of the security hierarchy in SQL Server 2016.

Active security refers to the practice of limiting users' access to data and structures through the use of permissions. When working with the SQL Server security model, the three entities to ensure that you understand are principals, securable, and permissions. The definition of each of these entities is found in Table 2-1.

Table 2-1. *Security Model Definitions*

Entity	Definition
Principal	A security principal is an entity, such as a user.
Securable	A securable is data, an artifact, or metadata.
Permission	Permissions are rights that are granted or denied to a security principal to define the principal's permitted access to a securable.

Tip Passive security refers to auditing activity. SQL Server Audit is discussed in Chapter 3.

Security Principal Hierarchy

Security principals are organized into a hierarchy, which allows administrators to assign permissions to a group of users. This has obvious benefits, allowing you to implement a security based on a user's role within an organization. For example, all salespersons can easily be assigned the same permissions if a preconfigured sales role that already has all required permissions assigned to it exists. Figure 2-1 defines the complete hierarchy

P. A. Carter, *Securing SQL Server*, DOI 10.1007/978-1-4842-2265-2_2

of security principals that can access data or data structures within SQL Server. The hierarchy begins at the domain level and passes through to the local server layer, the SQL Server instance layer, and finally, the database layer.

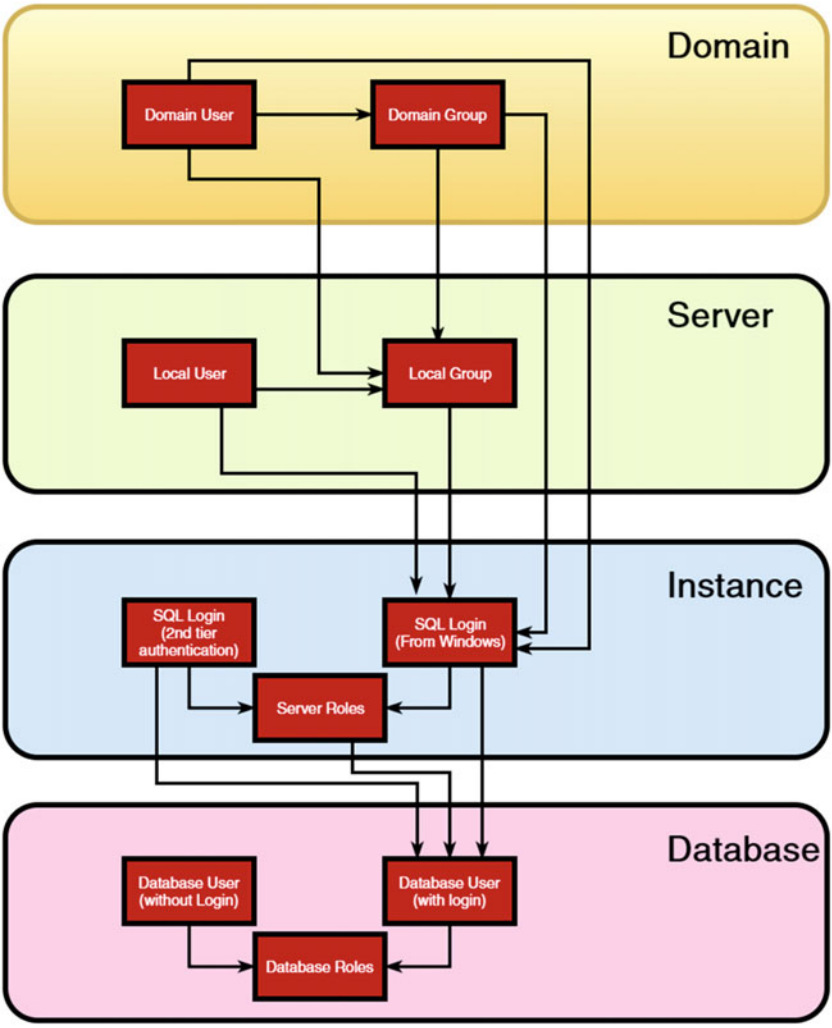


Figure 2-1. *Security hierarchy*

The diagram in Figure 2-1 shows that a login, created within the SQL Server instance, can be mapped to a local Windows user or group or to a domain user or group. Usually, in an enterprise environment, this is a domain user or group. (A *group* is a collection of users that are granted permissions as a unit.) This eases the administration of security. Imagine (as described earlier), that a new starter joins the sales team. When he is added

to the domain group called SalesTeam–which already has all of the required permissions to file system locations, SQL Server databases, and so on—he immediately inherits all required permissions to perform his role.

The diagram also illustrates how local server accounts or domain accounts and groups can be mapped to a user at the database level (a database user without login). This is part of the functionality of contained databases. This technology was introduced in SQL Server 2012 to support high availability with AlwaysOn Availability Groups. Contained database authentication is discussed later in this chapter.

SQL Server logins, which are mapped to local server or domain-level users or groups, are created at the SQL Server instance level. At the instance level, it is also possible to create SQL Server logins, which use second-tier authentication (if you are using mixed-mode authentication).

Both of these login types can be added to fixed server roles and user-defined server roles at the instance level. Doing this allows you to grant the principal a common set of permissions to instance-level objects, such as linked servers and endpoints. You can also map logins to database users, which in turn can be granted permissions to database level objects.

Database users reside at the database level of the hierarchy. You can grant them permissions directly to schemas and objects within the database, or you can add them to database roles. Database roles are similar to server roles, except they are used to grant a common set of permissions at the database layer instead of the instance layer. Database layer securables include schemas, tables, views, and stored procedures, and so forth.

Instance Level Security

Implementing instance level security involves creating and managing logins, credentials, and server roles. Securables at the instance level include databases, endpoints, and AlwaysOn Availability Groups. The following sections discuss logins, server roles, and credentials.

■ **Tip** Cryptographic Providers and SQL Server Audits are also administered at the instance level. SQL Server Audit is discussed in Chapter 3 and Cryptographic Providers are discussed in Chapter 5.

Logins

Since SQL Server 2012, it has been possible for users to authenticate directly to a database as part of contained database functionality. Generally, however, database engine users need to authenticate at the instance level. SQL Server supports two authentication modes at the instance level: Windows authentication and mixed mode authentication.

When an instance is in Windows authentication mode, users must authenticate to either the local server or to the domain before they can access the SQL Server instance. A login is created within the SQL Server instance, which maps to either their Windows user or a Windows group that contains their Windows user. The SID (security identifier) of the Windows principal is stored in the Master database of the instance.

■ **Tip** In addition to creating a login mapped to a Windows user or group, you can also map a login to a certificate or an asymmetric key. Doing so does not allow a user to authenticate to the instance by using a certificate, but it does allow code signing so that permissions to procedures can be abstracted, rather than granted directly to a login. This helps when you are using dynamic SQL, which breaks the ownership chain. In this scenario, when you run the procedure, SQL Server combines the permissions from the user who called the procedure and the user who maps to the certificate.

When an instance is configured to use mixed mode authentication, it is still possible to create a login that maps to a Windows user or Windows group, but it is also possible to create second-tier logins, known as *SQL logins*. These logins have their name, password, and SID stored in the Master database of the instance. These details are verified when a user connects to the instance. A user can then authenticate to the instance using this username and password, without the need for prior authentication to the server or domain.

Mixed mode authentication is less secure than authentication, because it is possible to attack the instance without first authenticating to the domain. Therefore, it is best practice us use Windows authentication. It is often necessary to use mixed mode authentication, however, for the following reasons:

- Legacy applications that require a second-tier login
- Access from outside of the domain (such as a Linux server)
- Environments where security is implemented in the application tier and a single SQL login connects to the database engine

Creating a Login

A login can be created via T-SQL by using the `CREATE LOGIN` statement. When creating a login from a Windows user or group, the syntax is very straightforward, because there is no password management involved. Table 2-2 describes the `WITH` options that are valid when creating a login from a Windows security principal.

Table 2-2. *CREATE LOGIN Options for Windows Security Principal*

Option	Description
`DEFAULT_DATABASE`	Specifies a "landing" database for the login. This is the database to which their context is scoped when they initially authenticate to the instance. This scope can be overwritten in the connection string.
`DEFAULT_LANGUAGE`	Specifies the default language that is assigned to the login. If omitted, then this is configured to be the default language of the instance.

Listing 2-1 demonstrates how to create a login for the Windows user Pete in the CarterSecureSafe.com domain.

Listing 2-1. Create a Login from a Windows Security Principal

```
USE master
GO

CREATE LOGIN [cartersecuresafe\Pete]
FROM WINDOWS
WITH DEFAULT_DATABASE=master, DEFAULT_LANGUAGE=British ;
GO
```

When creating a second tier, SQL login, however, there are many more WITH options that can be used. These options are described in Table 2-3.

Table 2-3. *CREATE LOGIN Options for SQL Login*

Option	Description
PASSWORD	Specifies the initial password used by the login; in clear text.
PASSWORD HASHED	Specifies a hashed representation of the initial password used by the login.*
SID	Specifies the SID of the login.*
DEFAULT_DATABASE	Specifies a "landing" database for the login. This is the database to which their context id scoped when they initially authenticate to the instance. This scope can be overwritten in the connection string.
DEFAULT_LANGUAGE	Specifies the default language assigned to the login. If omitted, then this is configured to be the default language of the instance.
CHECK_POLICY	Specifies that the login's password must meet the same requirements, such as length and complexity, as Windows users, as enforced by Group Policy or Local Security Policy.
CHECK_EXPIRATION	Specifies that the login's password will expire, in line with password expiration policy configured for Windows users, as enforced by Group Policy or Local Security Policy. This option is only valid if CHECK_POLICY is also specified.
MUST_CHANGE	Specifies that the user must change their password the first time that they login to the instance. This option can only be used if CHECK_EXPIRATION is also specified.

**Discussed in the "Migrating Logins Between Instances" section*

Listing 2-2 demonstrates how to create a SQL login called Danni. The password for the login must meet the complexity requirements specified by Group Policy, but the password will not expire. This means that it is not possible to force the password to be changed the first time that the user logs in.

Listing 2-2. Create a SQL Login

```
USE master
GO

CREATE LOGIN Danni
WITH PASSWORD='COmplexPa$$wOrd',
DEFAULT_DATABASE=master, CHECK_EXPIRATION=OFF, CHECK_POLICY=ON ;
GO
```

Migrating Logins Between Instances

There will be occasions when you need the same login to exist on multiple servers. This may be because of a server migration, or it may be because you are implementing DR and you need users to be able to reconnect to the DR instance transparently.

For logins that are created from Windows security principals, this poses no problem what so ever. As discussed earlier in this chapter, the SID of the Windows security principal is stored in the Master database, but the principal itself, is managed by Windows. Therefore, you can simply create a login on the second instance and map it to the same Windows user or group.

If you are working with SQL logins, however, this scenario is more challenging to deal with, because the SQL Server instance hosts and manages the SID and the password of the login. This means that if you create a login with the same name and password on a different instance, then they will still be completely different, isolated principals. Once you move or failover your databases to the second instance, the database users become orphaned. This means that they no longer map to a login. This is because the login on the second instance has a different SID, despite having the same name and password.

You can manage this issue at point of failover by using an `ALTER USER` statement `WITH LOGIN` option.

■ **Tip** The `ALTER USER WITH LOGIN` syntax replaces the functionality of the deprecated `sp_change_users_logins` system stored procedure.

Listing 2-3 demonstrates how to remap a database user called Danni in the AdventureWork2016 database to a login called Danni on a new instance.

Listing 2-3. Remap a Database User to a Login

```
USE AdventureWorks2016
GO

ALTER USER Danni WITH LOGIN = Danni ;
```

■ **Tip** Database users are discussed in the “Database-Level Security” section of this chapter.

Naturally, it is more efficient if users are already mapped to their correct login at the point that the databases are moved or failed over. This can be achieved by creating the login and manually assigning it the correct SID.

This is where the `WITH SID` option mentioned in Table 2-1 comes into play. If you script out the SQL login on an instance and include the SID, then you can use the script to pre-create the login on the DR instance with the correct SID, meaning that the database users automatically map to the correct login without the need to alter them. Listing 2-4 demonstrates how this can be achieved using a SQLCMD script, which replicates a login called Danni from `ProdInstance1` to `DRInstance1`.

■ **Tip** The script must be running in `SQLCMD` mode; otherwise, it will not execute.

Listing 2-4. Migrate a Login to a New Instance

```
:CONNECT CARTERSECURESAFE\ProdInstance1

DECLARE @SQL NVARCHAR(MAX) ;

SET @SQL = (SELECT 'CREATE LOGIN '
                        + name
                        + ' WITH PASSWORD = ''COmplexPasswOrd'', SID = 0x'
                        + CONVERT(NVARCHAR(64), SID, 2)
                FROM sys.sql_logins
                WHERE Name = 'Danni') ;

:CONNECT CARTERSECURESAFE\DRInstance1

EXEC(@SQL) ;
```

■ **Caution** The preceding method works perfectly well in a `SQLCMD` script. Although it is worthy to note that depending on your server and network configuration, you may be sending the password across the wire in plain text. If this is the case, then you should follow the approach discussed next to hash the password.

An issue arises with the approach that we have discussed; it's a common scenario, in which you want to script out the logins for an environment and keep them under source control so that you can apply them to other environments, as required. In this scenario, you store passwords in plain text. This poses an obvious security risk and it should be avoided. Instead, you should script out the SQL logins, so that the passwords are stored in the same encrypted format that SQL Server stores them in.

This approach can be achieved by scripting the logins with passwords based on the `password_hash` column of the `sys.sql_logins` view. Instead of this, however, we script the logins using the `HASHBYTES()()` function to generate the password hash for the login. This technique is demonstrated in Listing 2-5 and gives you an insight into how SQL

Server hashes the passwords. The script generates the DDL (Data Definition Language) statements required to script out all enabled SQL logins from an instance with hashed passwords. The script can subsequently be placed in source control and run on other environments as required.

■ **Tip** In normal operations, you should take the approach of taking the password hash from the `sys.sql_logins` view. This is preferable because it does not require knowledge of the login's plain text password.

The `HASHBYTES()` function returns a hash of its input. It accepts the parameters detailed in Table 2-4.

Table 2-4. *HASHBYTES() Parameters*

Parameter	Description
Algorithm	The algorithm that the function uses to hash the data. The following are acceptable values: • MD2 • MD4 • MD5 • SHA • SHA1 • SHA2_256 • SHA2_512
Input	The value hashed by the function.

Notice that the script uses a function called `CRYPT_GEN_RANDOM()`. This function uses the Windows CAPI (Crypto API) to generate a cryptographic random number, which it returns as a hexadecimal number. The function accepts the parameters detailed in Table 2-5.

Table 2-5. *CRYP_GEN_RANDOM Parameters*

Parameter	Description
Length	The length of the number to be generated.
Seed	An optional randomization seed.

Listing 2-5. Script out Logins with Hashed Passwords

```
DECLARE @password NVARCHAR(MAX) = 'COmplexPa$$wOrd' ;
DECLARE @salt VARBINARY(4) = CRYPT_GEN_RANDOM(4) ;
DECLARE @hash VARBINARY(1000) ;
DECLARE @SQL NVARCHAR(MAX) ;
```

```
SET @hash = (SELECT 0x0200 + @salt + HASHBYTES('SHA2_512', CAST(@password
AS VARBINARY(MAX)) + @salt)) ;

SET @SQL = (SELECT 'CREATE LOGIN '
                                + Name
                                + ' WITH PASSWORD = '
                                + CONVERT(NVARCHAR(1000), @hash, 1)
                                + ' HASHED, SID = 0x'
                                + CONVERT(NVARCHAR(64), SID, 2)
                        FROM sys.sql_logins
                        WHERE is_disabled = 0
                        FOR XML PATH('')) ;

SELECT @SQL ;
```

Server Roles

SQL Server provides a set of built-in server roles. These roles allow you to assign instance-level permissions to logins that have common requirements. They are called *fixed server roles*. It is not possible to change the permissions that are granted to them; you can only add and remove logins. Table 2-6 describes each fixed server role.

Table 2-6. *Fixed Server Roles*

Role	Description
`sysadmin`	The `sysadmin` role gives administrative permissions to the entire instance. A member of the sysadmin role can perform any action within the instance of the SQL Server relational engine.
`bulkadmin`	In conjunction with the `INSERT` permission on the target table within a database, the `bulkadmin` role allows a user to import data from a file using the `BULK INSERT` statement. This role is normally given to service accounts that run ETL processes.
`dbcreator`	The `dbcreator` role allows its members to create new databases within the instance. Once a login creates a database, that login is automatically the owner of that database and is able to perform any action inside it.
`diskadmin`	The `diskadmin` role gives its members the permissions to manage backup devices within SQL Server.
`processadmin`	Members of the `processadmin` role are able to stop the instance from T-SQL or SSMS (SQL Server Management Studio). They are also able to kill running processes.

(*continued*)

Table 2-6. (*continued*)

Role	Description
public	All logins are added to the public role. Although you can assign permissions to the public role, this does not fit with the principle of least privilege. This role is normally only used for internal SQL Server operations, such as authentication to TempDB.
securityadmin	Members of the securityadmin role are able to manage logins at the instance level. For example, members may add a login to a server role (except sysadmin) or assign permissions to an instance-level resource, such as an endpoint. However, they cannot assign permissions within a database to database users.
serveradmin	serveradmin combines the diskadmin and processadmin roles. In addition to starting or stopping the instance, however, members of this role can also shut down the instance using the SHUTDOWN T-SQL statement. The subtle difference here is that the SHUTDOWN command gives you the option of not running a CHECKPOINT in each database if you use it with the NOWAIT option. Additionally, members of this role can alter endpoints and view all instance metadata.
setupadmin	Members of the setupadmin role are able to create and manage linked servers.

You can also create custom server roles, which allow you to grant a custom set of permissions to a group of logins. For example, if you implemented AlwaysOn Availability Groups, then you may wish to create a server role called AvailabilityRole and grant this group the following permissions:

- Alter any availability group
- Alter any endpoint
- Create an availability group
- Create endpoint

You can then add to this role the junior DBAs who are not authorized to be made members of the sysadmin fixed server role, but who need to manage the high availability and disaster recovery of the instance. The script in Listing 2-6 demonstrates how to create the server role and grant it the relevant permissions.

Notice that the script uses the GRANT statement to assign permissions to the role. When assigning permissions to a server role or login, the three following assignments can be made:

- GRANT
- DENY
- REVOKE

GRANT provides a principal with the permissions to access a securable. You can use the WITH option in a GRANT statement to additionally provide a principal with the ability to GRANT the same permission to other principals.

DENY specifically denies a principal's permissions to access a securable; DENY always overrules GRANT. Therefore, if a login is a member of a server role (or roles) that gives the login permissions to alter an endpoint, but the principal itself, has explicitly been denied permissions to alter the same endpoint, then the principal is not able to manage the endpoint.

REVOKE removes a permission assignment to a securable. This includes DENY associations as well as GRANT associations. If a login has been assigned permissions through a server role, however, then revoking the permissions to that securable, directly against the login itself, has no effect. In order to have an effect, you need to either use the DENY permission assignment against the login or server role, or you need to REVOKE the permissions from the server role.

Listing 2-6. Create Server Role and Grant Permissions

```
CREATE SERVER ROLE AVAILABILITYROLE AUTHORIZATION [CarterSecureSafe\
SQLAdmin] ;
GO

GRANT ALTER ANY AVAILABILITYROLE GROUP TO AVAILABILITYROLE ;
GRANT ALTER ANY ENDPOINT TO AVAILABILITYROLE ;
GRANT CREATE AVAILABILITYROLE GROUP TO AVAILABILITYROLE ;
GRANT CREATE ENDPOINT TO AVAILABILITYROLE ;
GO
```

You could add the Danni login to this server role by using the code shown in Listing 2-7.

Listing 2-7. Add Login to Server Role

```
ALTER SERVER ROLE AvailabilityRole ADD MEMBER Danni ;
GO
```

Credentials

Credentials are used to provide the ability to access resources that are external to the SQL Server instance. SQL logins can use credentials to access operating system–level resources. SQL Server Agent Proxy Accounts use credentials to access SQL Server Agent Subsystems, such as PowerShell or CmdExec. Credentials are also used when taking backups to Azure.

When used to access operating system–level resources, a credential usually records the identity and password of a Windows security principal. If used for backups to Azure, however, then the credential records the name and private key of the Azure storage account. Listing 2-8 demonstrates how to create a credential to use for backups to Azure; the storage account is called CarterSecureSafeStorageAcc.

Listing 2-8. Create a Credential for Backups to Azure

```
CREATE CREDENTIAL URLBackupCredential
WITH IDENTITY = 'CarterSecureSafeStorageAcc'
              ,SECRET ='\Ydfg\
SGdTgJNpVFl992sBv7Bp1gyL61I33wNrTMHGBDdtVcx97F5f6SC5uDi59FeY2/
IjxyqsuLU2xrkrNAGT==' ;
```

Database-Level Security

At the database level, security is implemented by assigning permissions to security principals; namely users and database roles. The following sections discuss both of these principal types.

Users

A database user is typically created from a login at the instance level. This means that the same instance level security principal can be granted permissions on resources in multiple databases. Since SQL Server 2012, however, it has also been possible to create a user without a login. The following sections discuss each of these database user types.

Users with a Login

Users can be created by using the `CREATE USER` T-SQL statement. Users that are created with an association to a login have a limited set of options that can be configured. These options are described in Table 2-7.

Table 2-7. *Options When Creating a User from a Login*

Option	Description
`DEFAULT_SCHEMA`	Specifies the default schema for a user. Schemas are discussed in Chapter 4
`ALLOW_ENCRYPTED_VALUE_MODIFICATIONS`	Specifies that users are allowed to bulk copy encrypted data, without first decrypting it. Encryption is discussed in Chapter 5.

You can create a user in the AdventureWorks2016 database, which is associated with the login called Danni, by using the script in Listing 2-9.

Listing 2-9. Create a User from a Login

```
USE AdventureWorks2016
GO

CREATE USER Danni FOR LOGIN Danni
        WITH DEFAULT_SCHEMA = Sales ;
```

The script creates a user that has the same name as the login. Whilst this is not mandatory, it is sensible, because it aids the administration of the security principals and makes the hierarchy of principals transparent to new DBAs joining your team. The script sets the user's default schema to `Sales`. This means that the Danni user is able to reference objects in the `Sales` schema using one-part names. If no default schema is specified for a user, then their default schema is `dbo`.

■ **Note** Schemas are discussed in further detail in Chapter 4.

Users Without a Login

When creating a user that is not associated with a login, the user can either be mapped to a Windows security principal or it can be created using SQL Server authentication.

■ **Note** Users can only be created by using SQL Server authentication if the database is configured with a partial containment level. Contained databases are beyond the scope of this book, but you can learn more about them at `https://msdn.microsoft.com/en-us/library/ff929071.aspx`.

Table 2-8 details the properties that can be configured when creating a user without a login.

Table 2-8. *Options When Creating a User Without a Login*

Option	Description
`DEFAULT_SCHEMA`	Specifies the default schema for a user. Schemas are discussed in Chapter 4
`ALLOW_ENCRYPTED_VALUE_MODIFICATIONS`	Specifies that users are allowed to bulk copy encrypted data, without first decrypting it. Encryption is discussed in Chapter 5.
`DEFAULT_LANGUAGE`	Specify the default language for the user. This options can be expressed as an `lcid`, a language name, or a language alias.
`SID`	Specify the SID associated with the user (SQL Server authentication only). This creates users in multiple databases that share the same SID. It helps with high availability and disaster recovery techniques by using AlwaysOn Availability Groups.

Listing 2-10 demonstrates how to create a user called Phil from a Windows security principal called Phil, which exists in the `cartersecuresafe` domain.

Listing 2-10. Create a User from a Windows Security Principal

```
USE AdventureWorks2016
GO

CREATE USER [cartersecuresafe\phil]
        WITH DEFAULT_SCHEMA=dbo ;
```

As previously mentioned, creating users with SQL authentication—also known as users with a password—is only possible in contained databases. If you attempt to create a user with SQL authentication in a database that is not contained, you receive the error shown in Figure 2-2.

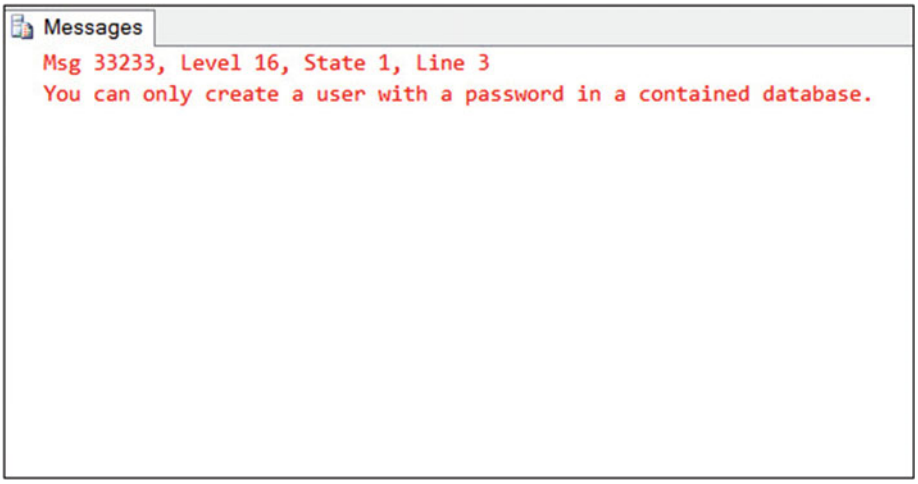


Figure 2-2. *Error when attempting to create a user with password in a non-contained database*

The script in Listing 2-11 configures the instance to allow contained database authentication before configuring the AdventureWorks2016 database to support partial containment. Finally, the script creates a user, named Pete, with a password.

Listing 2-11. Create a User with Password in a Contained Database

```
USE master
GO

EXEC sp_configure 'show advanced options', 1 ;
GO
```

```
RECONFIGURE ;
GO

EXEC sp_configure 'contained database authentication', '1' ;
GO

RECONFIGURE WITH OVERRIDE ;

ALTER DATABASE AdventureWorks2016
        SET CONTAINMENT = PARTIAL
        WITH NO_WAIT ;
GO

USE AdventureWorks2016
GO

CREATE USER Pete
        WITH PASSWORD = 'Pa$$wOrd123' ;
GO
```

When you use contained database users, you need to take a number of additional security considerations into account. First, some applications may require a user have permissions to multiple databases. If the user is mapped to a Windows user or group, then this is straightforward because the SID that is being authenticated is that of the Windows object. If the database user is using second-tier authentication, however, then you need to duplicate the user's SID from the first database. To do this, you need to adhere to the following steps:

1. Create a user with password in the first database.
2. Retrieve the user's SID from `sys.database_principals`.
3. Create the user in additional databases, specifically supplying the SID that you have recovered from the metadata.

The `sys.database_principals` catalog view exposes the columns detailed in Table 2-9.

Table 2-9. *sys.database_principals Columns*

Column	Description
`name`	The name of the security principal.
`principal_id`	The id of the security principal. This id is only unique within the database.
`type`	A single-character abbreviation of the type description.

(*continued*)

Table 2-9. (*continued*)

Column	Description
`type_desc`	A description of the type of security principal. The following are possible values: • APPLICATION_ROLE • CERTIFICATE_MAPPED_USER • EXTERNAL_USER • WINDOWS_GROUP • ASYMMETRIC_KEY_MAPPED_USER • DATABASE_ROLE • SQL_USER • WINDOWS_USER • EXTERNAL_GROUPS
`default_schema_name`	The name of the principals default schema.
`create_date`	The date and time that the principal was created.
`modify_date`	The date and time that the principal was last modified.
`owning_principal_id`	The id of the security principal that is marked as the owner of the principal. This is 1, which is the `dbo` id for all principals except database roles.
`sid`	The SID of the security principal.
`is_fixed_role`	Indicates if the principal is a fixed database role. • 1 indicates that the principal is a fixed database role • 0 indicates that the principal is not a fixed database role
`authentication_type`	Indicates how the principal authenticates to the database. The following are the possible values: • 0 indicates No authentication • 1 indicates Instance authentication • 2 indicates Database authentication • 3 indicates Windows authentication
`authentication_type_desc`	A textual description of the authentication type. The following are the possible values: • NONE • INSTANCE • DATABASE • WINDOWS
`default_language_name`	The name of the default language assigned to the principal
`default_language_lcid`	The `lcid` of the default language assigned to the principal

Listing 2-12 demonstrates how to generate a list of usernames and SID for users with a password in the AdventureWorks2016 database.

Listing 2-12. Retrieve SIDs for Users with a Password

```
USE AdventureWorks2016
GO

SELECT
        name
        ,sid
FROM sys.database_principals
WHERE authentication_type = 2 ;
```

Database Roles

SQL Server provides a set of built-in database roles. These roles allow you to assign database-level permissions to users that have common requirements. They are called *fixed database roles*. It is not possible to change the permissions that are granted to them; you can only add and remove users to and from the roles. Table 2-10 describes each of these roles.

Table 2-10. *Fixed Database Roles*

Role	Description
db_accessadmin	Members of this role can add and remove database users from the database.
db_backupoperator	The db_backupoperator role gives users the permissions they need to back up the database, natively. It may not work for third-party backup tools, such as CommVault or Backup Exec, since these tools often demand sysadmin rights.
db_datareader	Members of the db_datareader role can run SELECT statements against any table in the database. It is possible to override this for specific tables by explicitly denying a user, the permissions to read those tables. DENY always overrides GRANT.
db_datawriter	Members of the db_datawriter role can perform DML (data manipulation language) statements against any table in the database. It is possible to override this for specific tables by specifically denying a user the permissions to write to a table. DENY always overrides GRANT.
db_denydatareader	The db_denydatareader role denies the SELECT permission against every table in the database.

(*continued*)

Table 2-10. (*continued*)

Role	Description
db_denydatawriter	The db_denydatawriter role denies its members the permissions to perform DLM statements against every table in the database.
db_ddladmin	Members of this role are given the ability to run CREATE, ALTER, and DROP statements against any object in the database. This role is rarely used, but I have seen a couple of examples or poorly written applications that create database objects on the fly. If you are responsible for administering an application such as this, then the ddl_admin role may be useful.
db_owner	Members of the db_owner role can perform any action within the database that has not been specifically denied.
db_securityadmin	Members of this role can GRANT, DENY, and REVOKE a user's permissions to securables. They can also modify role memberships, with the exception of the db_owner role.

In addition to the fixed roles, it is also possible to create your own user-defined database roles. This simplifies administration by allowing DBAs to create roles that map to requirements of business teams that use a specific database. For example, an administrator of the AdventureWorks2016 database may create a role for salespeople, a role for the procurement department, and a role for the manufacturing department.

The script in Listing 2-13 demonstrates how to create the role for the sales team. The role is granted SELECT, INSERT, and UPDATE permissions on the Sales schema of the AdventureWorks2016 database. The Danni user is a member of the role. The role is called SalesRole and owned by dbo.

Listing 2-13. Create SalesRole

```
USE AdventureWorks2016
GO

--Create the role

CREATE ROLE SalesRole AUTHORIZATION dbo ;
GO

--Grant permissions to the role

GRANT DELETE ON SCHEMA::Sales TO SalesRole ;

GRANT INSERT ON SCHEMA::Sales TO SalesRole ;

GRANT SELECT ON SCHEMA::Sales TO SalesRole ;
```

```
GRANT UPDATE ON SCHEMA::Sales TO SalesRole ;

--Add user to the role

ALTER ROLE SalesRole ADD MEMBER Danni ;
```

Summary

SQL Server provides a flexible hierarchy to implement security. Role-based security is available out of the box. Database administrators should embrace this ability to simplify the administration of security.

The database engine supports two methods of authentication: Windows authentication and SQL Server authentication. The latter is SQL Server's implementation of second-tier authentication. Windows authentication should be used—unless there is a good reason not to—because it is more secure.

Database engine users typically authenticate to the instance via a login. This login then maps to users within the databases that they require access to. It is also possible for users to authenticate directly to a database. This is known as a *user without a login*. When a user without a login is implemented, they can authenticate via Windows authentication, or if the database is contained, then SQL Server authentication can alternatively be used.

CHAPTER 3

SQL Server Audit

Passive security refers to the practice of logging user activity to avoid the threat of non-repudiation. This is important because if an attack is launched by a privileged user, it allows appropriate disciplinary or even legal action to be taken. SQL Server provides SQL Server Audit to assist with implementing passive security.

Understanding SQL Server Audit

SQL Server Audit offers DBAs the ability to capture granular information about activity at both the instance level and the database level. Audit logs can be saved to a file, the Windows Security log, or the Windows Application log. The location that the audit logs are saved to is known as the *target*. There is exactly one target associated with each audit.

The SQL Server Audit resides at the instance level and defines the properties of the audit and the target. It is possible to create multiple server audits in each instance. This is useful if you have to audit many events in a busy environment, as it allows you to distribute the IO by using file targets and placing each target file on a separate volume.

Choosing the correct target is an important security consideration. If you choose the Windows Application log as a target, then any Windows user who is authenticated to the server is able to access it. The Security log is a lot more secure than the Application log but also more complex to configure as a target for SQL Server Audit logs.

When using the Security log as a target, the service account that is running the SQL Server service requires the Generate Security Audits user rights assignment. This can be assigned from the server's local security policy, but ideally, it is configured at a group policy level to avoid the risk of a GPO change overriding the setting. Application-generated auditing also needs to be enabled.

Size is another consideration for the target. If you decide to use the Application log or Security log, then it is important that you consider and potentially increase the size of these logs before you begin using them for your audit. Also, work with your Windows administration team to decide on how the log is cycled when full and if you are archiving the log by backing it up to tape.

P. A. Carter, *Securing SQL Server*, DOI 10.1007/978-1-4842-2265-2_3

The SQL Server Audit can be associated with one or more server audit specifications and database audit specifications. Specifications define the activity that is captured by the audit at the instance level and the database level, respectively. It is helpful to have multiple server and/or database audit specifications if you are planning to log many actions, because you can categorize them to make administration easier, while still associating them with the same server audit. Each database within the instance must have its own database audit specification if you plan to audit activity across multiple databases.

SQL Server Audit Actions and Action Groups

SQL Server Audit events are based on the SQL Server event classes. Related actions are grouped into audit action groups. These audit action groups map to SQL Server event class categories. When creating a server audit specification or database audit specification, you configure the audit specification to capture audit action groups, which contain the events that you wish to capture.

Audit groups are available at three distinct layers: server (meaning instance), database, and audit. Providing the ability to audit the changes to audits avoids the threat of non-repudiation caused by a privileged user launching an attack and attempting to cover his tracks by changing the audit information that is logged.

Table 3-1 describes the action groups that are available at the server level. Notice that some groups are nested.

Table 3-1. *Server-Level Action Groups*

Action Group	Description	Actions Contained
APPLICATION_ROLE_CHANGE_PASSWORD_GROUP	The event is raised when an application role's password is changed.	APPLICATION_ROLE_CHANGE_PASSWORD_GROUP
AUDIT_CHANGE_GROUP	The event is raised when Audit is created, dropped, or altered.	CREATE ALTER DROP AUDIT SHUTDOWN ON FAILURE CREATE ALTER DROP AUDIT_CHANGE_GROUP
BACKUP_RESTORE_GROUP	The event is raised when a `BACKUP` command or `RESTORE` command is issued.	RESTORE BACKUP_RESTORE_GROUP BACKUP BACKUP LOG
BROKER_LOGIN_GROUP	The event is raised when Service Broker security events occur.	BROKER LOGIN BROKER_LOGIN_GROUP
DATABASE_CHANGE_GROUP	The event is raised when a database is created, dropped, or altered.	DATABASE_CHANGE_GROUP CREATE ALTER DROP
DATABASE_LOGOUT_GROUP	The event is raised when a user without a login logs out of a database	DATABASE LOGOUT DATABASE_LOGOUT_GROUP
DATABASE_MIRRORING_LOGIN_GROUP	The event is raised when Database Mirroring related security events occur	DATABASE MIRRORING LOGIN DATABASE_MIRRORING_LOGIN_GROUP
DATABASE_OBJECT_ACCESS_GROUP	The event is raised when non-schema bound database objects are accessed.	DATABASE_OBJECT_ACCESS_GROUP
DATABASE_OBJECT_CHANGE_GROUP	The event is raised when non-schema bound database objects are created, dropped, or altered.	DATABASE_OBJECT_CHANGE_GROUP

(*continued*)

Table 3-1. (*continued*)

Action Group	Description	Actions Contained
DATABASE_OBJECT_OWNERSHIP_CHANGE_GROUP	The event is raised when the owner of a database object is changed.	DATABASE_OBJECT_OWNERSHIP_CHANGE_GROUP
DATABASE_OBJECT_PERMISSION_CHANGE_GROUP	The event is raised when permissions are assigned or revoked from a database object.	DATABASE_OBJECT_PERMISSION_CHANGE_GROUP
DATABASE_OPERATION_GROUP	The event is raised when SQL Server background operational tasks, such as a `CHECKPOINT`, occurs.	VIEW DATABASE STATE CONNECT DATABASE_OPERATION_GROUP CHECKPOINT SUBSCRIBE QUERY NOTIFICATION AUTHENTICATE SHOW PLAN
DATABASE_OWNERSHIP_CHANGE_GROUP	The event is raised when the owner of a database is changed.	TAKE OWNERSHIP DATABASE_OWNERSHIP_CHANGE_GROUP
DATABASE_PERMISSION_CHANGE_GROUP	The event is raised when permissions are assigned or revoked to a principal within a database.	DATABASE_PERMISSION_CHANGE_GROUP REVOKE DENY GRANT GRANT WITH GRANT REVOKE WITH GRANT REVOKE WITH CASCADE DENY WITH CASCADE
DATABASE_PRINCIPAL_CHANGE_GROUP	The event is fired when a principal is created, dropped, or altered, within a database.	DATABASE_PRINCIPAL_CHANGE_GROUP
DATABASE_PRINCIPAL_IMPERSONATION_GROUP	The event is fired when the impersonation of a database-level principal occurs.	DATABASE_PRINCIPAL_IMPERSONATION_GROUP
DATABASE_ROLE_MEMBER_CHANGE_GROUP	The event is raised when the membership of a database role is changed.	DATABASE_ROLE_MEMBER_CHANGE_GROUP
DBCC_GROUP	The event is raised when a `DBCC` statement is run.	DBCC DBCC_GROUP

(*continued*)

Table 3-1. (*continued*)

Action Group	Description	Actions Contained
FAILED_DATABASE_AUTHENTICATION_GROUP	The event is raised when a user attempts to authenticate to a contained database, but the authentication fails.	FAILED_DATABASE_AUTHENTICATION_GROUP DATABASE AUTHENTICATION FAILED
FAILED_LOGIN_GROUP	The event is raised when an attempt to authenticate to the instance fails.	LOGIN FAILED
FULLTEXT_GROUP	The event is raised when full-text events occur.	FULLTEXT FULLTEXT_GROUP
LOGIN_CHANGE_PASSWORD_GROUP	The event is raised when a login's password is changed	RESET PASSWORD RESET OWN PASSWORD CHANGE OWN PASSWORD CHANGE PASSWORD UNLOCK ACCOUNT MUST CHANGE PASSWORD
LOGOUT_GROUP	The event is raised when a principal logs out of the instance.	LOGOUT
SCHEMA_OBJECT_ACCESS_GROUP	The event is raised when an object permission is used for a schema.	SELECT INSERT UPDATE DELETE REFERENCES EXECUTE RECEIVE VIEW CHANGETRACKING SCHEMA_OBJECT_ACCESS_GROUP
SCHEMA_OBJECT_CHANGE_GROUP	The event is raised when a schema is created, dropped, or altered.	SCHEMA_OBJECT_CHANGE_GROUP
SCHEMA_OBJECT_OWNERSHIP_CHANGE_GROUP	The event is raised when the owner of a schema-bound object is changed.	SCHEMA_OBJECT_OWNERSHIP_CHANGE_GROUP

(*continued*)

Table 3-1. (*continued*)

Action Group	Description	Actions Contained
SCHEMA_OBJECT_ PERMISSION_ CHANGE_GROUP	The event is raised when permissions are assigned or revoked to a schema-bound object.	SCHEMA_OBJECT_ PERMISSION_CHANGE_ GROUP
SERVER_OBJECT_ CHANGE_GROUP	The event is raised when an instance-level object is created, dropped, or altered.	ALTER BACKUP CREATE CREDENTIAL MAP TO LOGIN DROP NO CREDENTIAL MAP TO LOGIN RESTORE
SERVER_OBJECT_ OWNERSHIP_CHANGE_ GROUP	The event is raised when the owner of an instance-level object is changed.	TAKE OWNERSHIP
SERVER_OBJECT_ PERMISSION_CHANGE_ GROUP	The event is raised when permissions are assigned or revoked to an instance-level object.	DENY DENY WITH CASCADE GRANT GRANT WITH GRANT REVOKE REVOKE WITH CASCADE REVOKE WITH GRANT
SERVER_OPERATION_ GROUP	The event is raised when instance configuration changes are made.	ALTER ALTER RESOURCES CREATE DROP
SERVER_PERMISSION_ CHANGE_GROUP	The event is raised when permissions are assigned or revoked to instance-level permissions.	DENY DENY WITH CASCADE GRANT GRANT WITH GRANT REVOKE REVOKE WITH CASCADE REVOKE WITH GRANT SERVER_PERMISSION_ CHANGE_GROUP

(*continued*)

Table 3-1. (*continued*)

Action Group	Description	Actions Contained
SERVER_PRINCIPAL_ CHANGE_GROUP	The event is fired when instance-level principals are created, dropped, or altered.	ALTER CHANGE DEFAULT DATABASE CHANGE DEFAULT LANGUAGE CHANGE LOGIN CREDENTIAL CREATE DISABLE DROP ENABLE NAME CHANGE PASSWORD EXPIRATION PASSWORD POLICY
SERVER_PRINCIPAL_ IMPERSONATION_ GROUP	The event is raised when impersonation of an instance-level principal occurs.	IMPERSONATE
SERVER_ROLE_MEMBER_ CHANGE_GROUP	The event is raised when the membership of a server role is changed.	ADD MEMBER DROP MEMBER
SERVER_STATE_ CHANGE_GROUP	The event is raised when the state of the instance is modified.	SERVER CONTINUE SERVER PAUSED SERVER SHUTDOWN SERVER STARTED SERVER_STATE_CHANGE_ GROUP
SUCCESSFUL_ DATABASE_ AUTHENTICATION_ GROUP	The event is raised when a principal successfully authenticates to a contained database.	DATABASE AUTHENTICATION SUCCEEDED SUCCESSFUL_DATABASE_ AUTHENTICATION_ GROUP
SUCCESSFUL_LOGIN_ GROUP	The event is raised when a principal successfully authenticates to the instance.	LOGIN SUCCEEDED

(*continued*)

Table 3-1. (*continued*)

Action Group	Description	Actions Contained
TRACE_CHANGE_GROUP	The event is raised if a trace is modified.	ALTER TRACE TRACE AUDIT C2OFF TRACE AUDIT C2ON TRACE AUDIT START TRACE AUDIT STOP TRACE_CHANGE_GROUP
TRANSACTION_GROUP	The event is raised when a transaction begins, commits, or rolls back.	STATEMENT_ROLLBACK_ GROUP TRANSACTION_BEGIN_ GROUP TRANSACTION_COMMIT_ GROUP TRANSACTION_GROUP TRANSACTION_ ROLLBACK_GROUP
USER_CHANGE_ PASSWORD_GROUP	The event is raised when a user with password's password is changed.	USER_CHANGE_ PASSWORD_GROUP
USER_DEFINED_AUDIT_ GROUP	The event is triggered when the `sp_audit_write` procedure if executed.	USER DEFINED AUDIT USER_DEFINED_AUDIT_ GROUP

Table 3-2 describes the action groups that are available at the database level. Notice that many of the groups are the same as the server-level groups. The difference is that groups at the database-level apply only to the database with which they are associated. The server-level groups apply to all databases on the instance.

Table 3-2. *Database-Level Audit Action Groups*

Action Group	Description	Actions Contained
APPLICATION_ROLE_CHANGE_PASSWORD_GROUP	The event is triggered when an application role's password is changed.	APPLICATION_ROLE_CHANGE_PASSWORD_GROUP
AUDIT_CHANGE_GROUP	The event is raised when an audit is created, dropped, or altered.	CREATE ALTER DROP AUDIT SHUTDOWN ON FAILURE CREATE ALTER DROP AUDIT_CHANGE_GROUP
BACKUP_RESTORE_GROUP	The event is triggered when a database is backed up or restored.	RESTORE BACKUP_RESTORE_GROUP BACKUP BACKUP LOG
DATABASE_CHANGE_GROUP	The event is raised when a database is created, dropped, or altered.	DATABASE_CHANGE_GROUP CREATE ALTER DROP
DATABASE_LOGOUT_GROUP	The event is raised when a user without a login logs out of a database.	DATABASE LOGOUT DATABASE_LOGOUT_GROUP
DATABASE_OBJECT_ACCESS_GROUP	The event is raised when non-schema bound database objects are accessed.	DATABASE_OBJECT_ACCESS_GROUP
DATABASE_OBJECT_CHANGE_GROUP	The event is raised when non-schema bound database objects are created, dropped, or altered.	DATABASE_OBJECT_CHANGE_GROUP
DATABASE_OBJECT_OWNERSHIP_CHANGE_GROUP	The event is raised when the owner of a database is changed.	DATABASE_OBJECT_OWNERSHIP_CHANGE_GROUP

(*continued*)

Table 3-2. (*continued*)

Action Group	Description	Actions Contained
DATABASE_OBJECT_PERMISSION_CHANGE_GROUP	The event is raised when permissions are assigned or revoked to a principal within a database.	DATABASE_OBJECT_PERMISSION_CHANGE_GROUP
DATABASE_OPERATION_GROUP	The event is raised when SQL Server background operational tasks, such as a `CHECKPOINT`, occurs.	VIEW DATABASE STATE CONNECT DATABASE_OPERATION_GROUP CHECKPOINT SUBSCRIBE QUERY NOTIFICATION AUTHENTICATE SHOW PLAN
DATABASE_OWNERSHIP_CHANGE_GROUP	The event is raised when the owner of a database is changed.	TAKE OWNERSHIP DATABASE_OWNERSHIP_CHANGE_GROUP
DATABASE_PERMISSION_CHANGE_GROUP	The event is raised when permissions are assigned or revoked to a principal within a database.	DATABASE_PERMISSION_CHANGE_GROUP REVOKE DENY GRANT GRANT WITH GRANT REVOKE WITH GRANT REVOKE WITH CASCADE DENY WITH CASCADE
DATABASE_PRINCIPAL_CHANGE_GROUP	The event is fired when a principal is created, dropped, or altered within a database	DATABASE_PRINCIPAL_CHANGE_GROUP
DATABASE_PRINCIPAL_IMPERSONATION_GROUP	The event is fired when the impersonation of a database-level principal occurs.	DATABASE_PRINCIPAL_IMPERSONATION_GROUP
DATABASE_ROLE_MEMBER_CHANGE_GROUP	The event is raised when the membership of a database role is changed.	DATABASE_ROLE_MEMBER_CHANGE_GROUP
DBCC_GROUP	The event is raised when a `DBCC` statement is run.	DBCC DBCC_GROUP

(*continued*)

Table 3-2. (*continued*)

Action Group	Description	Actions Contained
FAILED_DATABASE_AUTHENTICATION_GROUP	The event is raised when a user attempts to authenticate to a contained database, but the attempt fails.	FAILED_DATABASE_AUTHENTICATION_GROUP DATABASE AUTHENTICATION FAILED
SCHEMA_OBJECT_ACCESS_GROUP	The event is raised when an object permission is used for a schema.	SELECT INSERT UPDATE DELETE REFERENCES EXECUTE RECEIVE VIEW CHANGETRACKING SCHEMA_OBJECT_ACCESS_GROUP
SCHEMA_OBJECT_CHANGE_GROUP	The event is raised when a schema is created, dropped, or altered.	SCHEMA_OBJECT_CHANGE_GROUP
SCHEMA_OBJECT_OWNERSHIP_CHANGE_GROUP	The event is raised when the owner of a schema-bound object is changed.	SCHEMA_OBJECT_OWNERSHIP_CHANGE_GROUP
SCHEMA_OBJECT_PERMISSION_CHANGE_GROUP	The event is raised when permissions are assigned or revoked to a schema-bound object.	SCHEMA_OBJECT_PERMISSION_CHANGE_GROUP
SUCCESSFUL_DATABASE_AUTHENTICATION_GROUP	The event is raised when a principal successfully authenticates to a contained database.	DATABASE AUTHENTICATION SUCCEEDED SUCCESSFUL_DATABASE_AUTHENTICATION_GROUP
USER_CHANGE_PASSWORD_GROUP	The event is raised when a user with password's password is changed.	USER_CHANGE_PASSWORD_GROUP
USER_DEFINED_AUDIT_GROUP	The event is triggered when the `sp_audit_write` procedure if executed.	USER DEFINED AUDIT USER_DEFINED_AUDIT_GROUP

Table 3-3 describes the audit action group available at the audit level.

Table 3-3. *Audit-Level Audit Action Groups*

Action Group	Description	Actions Contained
AUDIT_ CHANGE_ GROUP	The event is fired when a SQL Server Audit artifact is created, dropped, or altered.	CREATE SERVER AUDIT ALTER SERVER AUDIT DROP SERVER AUDIT CREATE SERVER AUDIT SPECIFICATION ALTER SERVER AUDIT SPECIFICATION DROP SERVER AUDIT SPECIFICATION CREATE DATABASE AUDIT SPECIFICATION ALTER DATABASE AUDIT SPECIFICATION DROP DATABASE AUDIT SPECIFICATION

Implementing SQL Server Audit

The following sections discuss how to create a server audit, a server audit specification, and a database audit specification.

Creating a Server Audit

A server audit can be created using the `CREATE SERVER AUDIT` DDL (Data Definition Language) statement. Table 3-4 describes the options that are available when creating a server audit.

Table 3-4. *Server Audit Options*

Option	Description
FILEPATH	Specifies the filepath where the audit logs is generated. Only applies if you choose a file target.
MAXSIZE	Specifies the largest size that the audit file can grow to. The minimum size you can specify for this is 2MB. Only applies if you choose a file target.
MAX_ROLLOVER_FILES	When the audit file becomes full, you can either cycle that file or generate a new file. The MAX_ROLLOVER_FILES setting controls how many new files can be generated before they begin to cycle. The default value is UNLIMITED, but specifying a number caps the number of files to this limit. If you set it to 0, then there will only ever be one file, which will cycle every time it becomes full. Any value above 0 indicates the number of rollover files that is permitted. So for example, if you specify 5, then there is a maximum of six files in total. Only applies if you choose a file target.
MAX_FILES	As an alternative to MAX_ROLLOVER_FILES, the MAX_FILES setting specifies a limit for the number of audit files that can be generated, but when this number is reached, the logs will not cycle. Instead, the audit fails and events that cause an audit action to occur are handled based on the setting for ON_FAILURE. Only applies if you choose a file target.
RESERVE_DISK_SPACE	Pre-allocate space on the volume equal to the value set in MAXSIZE, as opposed to allowing the audit log to grow as required. Only applies if you choose a file target.
QUEUE_DELAY	Specifies if audit events are written synchronously or asynchronously. If set to 0, events are written to the log synchronously. Otherwise, specify the duration in milliseconds that can elapse before events are forced to write. The default value is 1000 (1 second), which is also the minimum value.
ON_FAILURE	Specifies what should happen if events that cause an audit action fail to be audited to the log. Acceptable values are CONTINUE, SHUTDOWN, or FAIL_OPERATION. When CONTINUE is specified, the operation is allowed to continue. This can lead to unaudited activity occurring. FAIL_OPERATION causes auditable events to fail, but allows other actions to continue. SHUTDOWN forces the instance to stop if auditable events cannot be written to the log.
AUDIT_GUID	Because server and database audit specifications link to the server audit through a GUID, there are occasions when an audit specification can become orphaned. These include when you attach a database to an instance, or when you implement technologies such as AlwaysOn Availability Groups. This option allows you to specify a specific GUID for the server audit, as opposed to having SQL Server generate a new one.

It is also possible to create a filter on the server audit. This is useful when your Audit Specification captures activity against an entire class of object, but you are only interested in a subset of this information. For example, you may configure a server audit specification to log members added to or removed from server roles; but really, you are only interested in members being added to or removed from the `sysadmin` server role. In this scenario, you can filter on the `sysadmin` role and reduce the amount of "noise" recorded in the audit log.

■ **Note** Please refer to Chapter 2 for further information on server roles.

The script in Listing 3-1 demonstrates how to create a server audit. The audit uses a file target and the target may consist of an unlimited number of files, although each file is limited in size to 256MB. The audit is configured so that if the audit fails to log an operation, that operation will fail. There is also a filter placed on the audit so that only activity, where the `object_name` property is equal to `sysadmin`, is logged. This allows you to create a server audit specification, which checks for members being added to or removed from a server role, as discussed earlier.

■ **Tip** If you are following along with the demonstrations, then you should change the filepath to match your own configuration.

Listing 3-1. Create a Server Audit

```
USE Master
GO

CREATE SERVER AUDIT [Audit-CarterSecureSafe]
TO FILE
(
        FILEPATH = 'c:\audit_files\audit'
        ,MAXSIZE = 256 MB
        ,MAX_ROLLOVER_FILES = 2147483647
        ,RESERVE_DISK_SPACE = OFF
)
WITH
(
        QUEUE_DELAY = 1000
        ,ON_FAILURE = CONTINUE
)
WHERE object_name = 'sysadmin' ;
```

An audit can be enabled by altering the audit. This is demonstrated in Listing 3-2.

Listing 3-2. Enabling an Audit

```
ALTER SERVER AUDIT [Audit-CarterSecureSafe]
WITH (STATE = ON) ;
```

Create a Server Audit Specification

A server audit specification can be created using the `CREATE SERVER AUDIT SPECIFICATION` DDL statement. Table 3-5 describes the options that can be set when creating a server audit specification.

Table 3-5. *Server Audit Specification Options*

Argument	Description
audit_specification_name	The name to be assigned to the server audit specification.
audit_name	The name of the server audit to which the specification is associated.
audit_action_group_name	The name of a group of related auditable actions at the instance level.
`STATE`	Specifies if the server audit specification should be started on creation.

Listing 3-3 demonstrates how to create a server audit specification, which captures changes to the membership of server roles.

Listing 3-3. Create a Server Audit Specification

```
CREATE SERVER AUDIT SPECIFICATION [ServerAuditSpecification-
CarterSecureSafe]
FOR SERVER AUDIT [Audit-CarterSecureSafe]
ADD (SERVER_ROLE_MEMBER_CHANGE_GROUP) ;
```

Create a Database Audit Specification

Creating a database audit specification is similar to creating a server audit specification but provides more flexibility because you can specify filters, such as the securable or principal to be audited.

You can create a database audit specification by using the `CREATE DATABASE AUDIT SPECIFICATION` DDL statement. Table 3-6 describes the options that are available when creating a database audit specification.

Table 3-6. *Database Audit Specification Options*

Argument	Description
audit_specification_name	The name to be assigned to the database audit specification.
audit_name	The name of the server audit to which the specification is associated.
action	The granular action to be audited.
audit_action_group_name	The name of a group of related auditable actions at the database level.
class	The class name of the securable.
securable	The name of the securable to be audited.
principal	The name of the principal to be audited.
STATE	Specifies if the server audit specification should be started on creation.

Listing 3-4 demonstrates how to create a database audit specification that is associated with the `Audit-CarterSecureSafe` audit, and captures `DELETE` statements made against the `Person.Person` table in the AdventureWorks2016 database by any user.

Listing 3-4. Create a Database Audit Specification

```
CREATE DATABASE AUDIT SPECIFICATION [DatabaseAuditSpecification-
AdventureWorks2016]
FOR SERVER AUDIT [Audit-CarterSecureSafe]
ADD (DELETE ON OBJECT::Person.Person BY public) ;
```

Server audit specifications and database audit specifications can be enabled on creation or by altering the specification. This is demonstrated in Listing 3-5.

Listing 3-5. Enabling an Audit Specification

```
ALTER SERVER AUDIT SPECIFICATION [DatabaseAuditSpecification-
AdventureWorks2016]
WITH (STATE = ON) ;
```

Creating Custom Audit Events

There may be times when you want to use SQL Server Audit to capture very specific events that are not possible to do with the out-of-the-box functionality of SQL Server Audit. If this is the case, then you can create a server audit specification or a database audit specification that is configured to capture the `USER_DEFINED_AUDIT_GROUP` audit action group, and then manually fire the event in your application code. The following sections demonstrate how to create the server audit and database audit specification required to log sales orders in which more than five different items are ordered.

Creating the Server Audit and Database Audit Specification

The script in Listing 3-6 uses the techniques that you learned in this chapter to create a server audit and a database audit specification linked to the server audit and captures USER_DEFINED_AUDIT_GROUP. The database audit specification is created in the AdventureWorks2016 database in which the Person.Person table is hosted.

Listing 3-6. Create the Server Audit and Database Audit Specification

```
USE Master
GO

CREATE SERVER AUDIT [Audit-CarterSecureSafeCustom]
TO FILE
(
        FILEPATH = 'c:\audit_files\audit-custom'
        ,MAXSIZE = 256 MB
        ,MAX_ROLLOVER_FILES = 2147483647
        ,RESERVE_DISK_SPACE = OFF
)
WITH
(
        QUEUE_DELAY = 1000
        ,ON_FAILURE = CONTINUE
) ;
GO

CREATE SERVER AUDIT SPECIFICATION [ServerAuditSpecification-
CarterSecureSafeCustom]
FOR SERVER AUDIT [Audit-CarterSecureSafeCustom]
ADD (USER_DEFINED_AUDIT_GROUP) ;
GO

ALTER SERVER AUDIT [Audit-CarterSecureSafeCustom]
WITH (STATE = ON) ;

ALTER DATABASE AUDIT SPECIFICATION [ServerAuditSpecification-
CarterSecureSafeCustom]
WITH (STATE = ON) ;
```

Raising the Event

A custom event can be raised by using the `sp_audit_write` system stored procedures. Table 3-7 describes the parameters accepted by the `sp_audit_write` procedure. The values for all parameters are user-defined and are recorded in the audit log when the event is fired.

Table 3-7. *sp_audit_write Parameters*

Parameter	Description
`@user_defined_event_id`	Specifies the id of the user-defined event.
`@succeeded`	Specifies if the event was successful. • 0 indicates that the event failed • 1 indicates that the event succeeded
`@user_defined_information`	Specifies the description of the event.

The `sp_audit_write` procedure can be called from a code module, such as a stored procedure or trigger. In our scenario, the table is updated from ad hoc SQL within the Sales application, so you call the `sp_audit_write` procedure from inside a DML (data manipulation language) trigger. Listing 3-7 demonstrates how to create the trigger.

■ **Caution** DML triggers can cause a negative performance impact if they are created against a table that has many writes. They should be used with caution and performance characteristics should be assessed before being implemented in a production environment.

Listing 3-7. Create a Trigger to Fire the Event

```
CREATE TRIGGER FireCustomEvent
ON Person.Person
AFTER INSERT
AS
BEGIN
      IF (SELECT COUNT(*) FROM Inserted) > 5
      BEGIN
       EXEC sys.sp_audit_write 1, 1, 'More than 5 items order' ;
      END
END ;
```

Summary

SQL Server Audit provides DBAs with a flexible and lightweight auditing mechanism. This is important for avoiding issues of non-repudiation when privileged users perform unauthorized actions.

An audit object is used to configure the target. It is also used to specify the behaviors of the audit, such as what should happen if SQL Server Audit fails to write an event to the audit log. Multiple audits can exist on an instance.

Server audit specifications and database audit specification are used to define which events should be audited. Multiple server audit specifications and database audit specifications can be associated with a single audit.

SQL Server Audit is made extensible by the `USER_DEFINED_AUDIT_GROUP` audit action group. This action group enables custom events to be fired. Custom events are triggered by calling the `sp_audit_write` system stored procedure. This procedure can be called from a code module, such as a stored procedure or trigger; it also allows DBAs to capture events that are specific to their environments. They cannot capture through out-of-the-box functionality.

CHAPTER 4

Data-Level Security

Below the principal hierarchy, SQL Server provides a rich set of functionality for securing data. This chapter discusses the appropriate use of schemas, ownership chaining, impersonation, row-level security, and dynamic data masking.

Note Row-Level Security and dynamic data masking are new features of SQL Server 2016.

Schemas

Schemas provide a logical namespace for database objects. They also provide a layer of abstraction between objects and their owners. Every object within a database must be owned by a database user. In much older versions of SQL Server, this ownership was direct. In other words, a user named Luan could have owned ten individual tables. From SQL Server 2005 onward, however, this model has changed so that Luan now owns a schema, and the ten tables reside within the schema—meaning that Luan implicitly owns the ten tables.

This abstraction simplifies changing the ownership of database objects; in this example, to change the owner of the ten tables from Luan to Paul, you need to change the ownership of a single artifact (the schema) as opposed to changing the ownership of all ten tables.

Well-defined schemas can also help simplify the management of permissions, because you can grant a principal the permissions on a schema, as opposed to the individual objects within that schema. For example, assume that you have five sales-related tables: `SalesOrdersHeader`, `SalesOrderDetails`, `SalesPerson`, `Stores`, and `Customers`. If you put all five tables within a single schema named `Sales`, you would then be able to assign the `SELECT`, `UPDATE`, and `INSERT` permissions on the `Sales` schema to a database role, which contains the sales team's database users. Assigning permissions to an entire schema does not just affect tables, however. For example, granting `SELECT` on a schema also gives a user the permissions to run `SELECT` statements against all views within the schema. Granting the `EXECUTE` permission on a schema grants `EXECUTE` on all procedures and functions within the schema. For this reason, well-designed schemas group tables by business rules, as opposed to technical joins.

P. A. Carter, *Securing SQL Server*, DOI 10.1007/978-1-4842-2265-2_4

Consider the AdventureWorks2016 database, specifically the `SalesOrderHeader`, `SalesOrderDetail`, `SalesPerson`, `Employee`, and `Person` tables. Figure 4-1 is a partial database diagram of the AdventureWorks2016 database, which shows that these tables are physically joined with primary key and foreign key constraints. Even though the tables are physically joined, it would not be sensible to place the `SalesOrderHeader` or `SalesOrderDetails` tables in the same schema as the `Employee` or `Person` tables, because salespeople are unlikely to be authorized to see employee information. Instead, the only tables in the `Sales` schema should be the `SalesOrderHeader`, `SalesOrderDetail`, and `SalesPerson` tables. Indeed, this aligns with the actual design of the AdventureWorks2016 database.

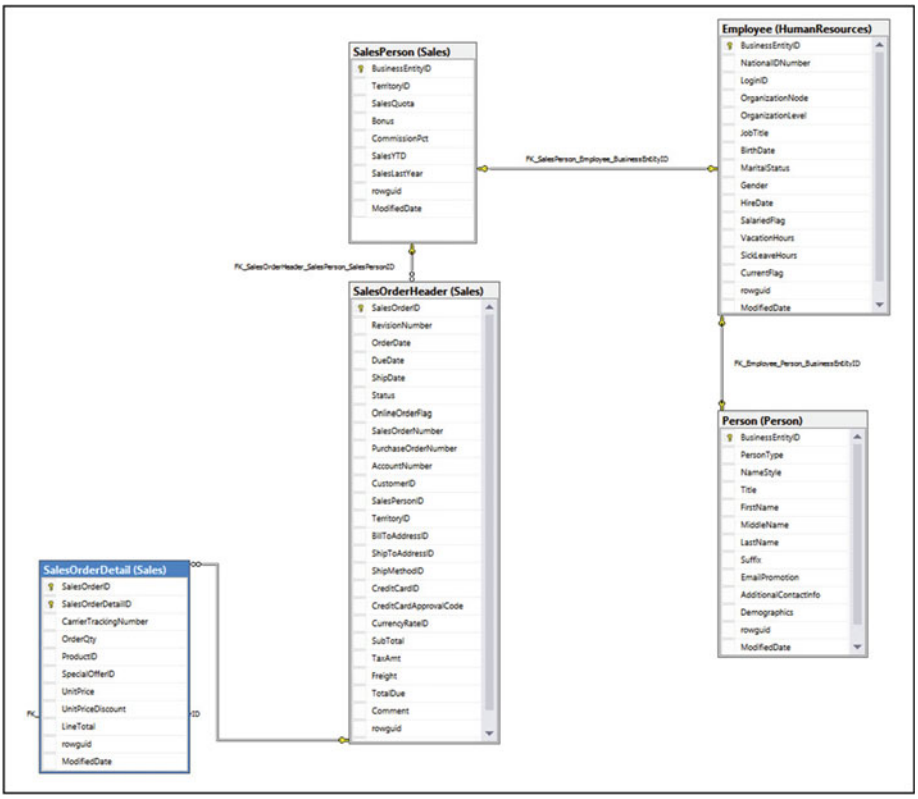


Figure 4-1. *Partial database diagram*

Listing 4-1 demonstrates how to create a schema called `Chapter4` in the AdventureWorks2016 database. It then assigns the user Danni, `SELECT` permission to the schema.

Listing 4-1. Create a New Schema

```
USE AdventureWorks2016
GO

CREATE SCHEMA Chapter4 ;
GO

GRANT SELECT ON SCHEMA::Chapter4 TO Danni ;
GO
```

To change a table's schema post creation, use the `ALTER SCHEMA TRANSFER` statement, as demonstrated in Listing 4-2. This script creates a table without specifying a schema, which means that it is automatically placed in the `dbo` schema. It is then moved to the `Chapter4` schema.

Listing 4-2. Transfer an Object to a Different Schema

```
USE AdventureWorks2016
GO

CREATE TABLE ChangeSchema
(
ID int
) ;
GO

ALTER SCHEMA Chapter4 TRANSFER dbo.ChangeSchema ;
GO
```

Ownership Chaining

SQL Server 2016 offers an implementation of row-level security, which is discussed in the "Row-Level Security" section of this chapter. In previous versions of SQL Server, however, row-level security could be rather tricky to implement. The standard way to implement row-level security was to use views or procedures, which limited the amount of data that was returned. Users can be granted permissions to the procedures and views, which form an abstraction layer, without granting the user permissions to the underlying tables.

This method works because of a concept called *ownership chaining*. When multiple objects are called sequentially by a query, SQL Server regards them as a chain. When you are chaining objects together, the permissions are evaluated differently, depending on the principal that owns the schema(s) in which the objects reside.

For example, imagine that you have a view named `View1`, which is based on two tables: `Table1` and `Table2`. If all three of these objects share the same owner, then when a `SELECT` statement is run against the view, the caller's permissions on the view are evaluated, but their permissions on the underlying tables are not.

This means that if you want to grant UserB the `SELECT` permissions on specific rows within `Table1`, then you can create a view that stores a query that returns the rows that this user is permitted to see. At this point, the user can run a `SELECT` statement from the view, as opposed to the base table. As long as he has `SELECT` permission on the view and the view shares an owner with the base table(s), then his permissions on the underlying table are not evaluated and the query succeeds. This is represented in Figure 4-2.

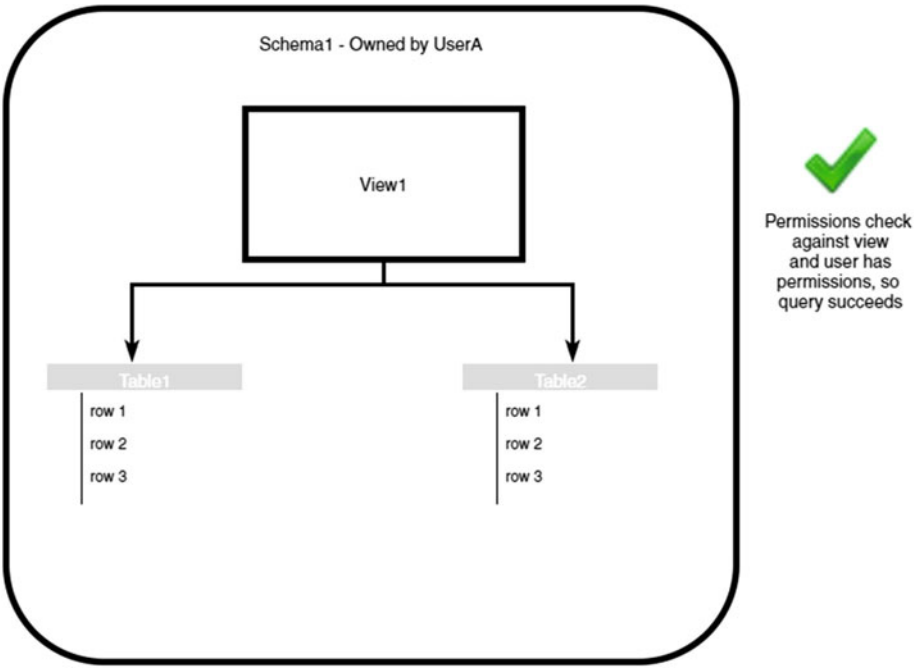


Figure 4-2. *Successful ownership chain*

The ownership chain is broken in the event that one of the objects that the view is based on does not have the same owner as the view. In this scenario, permissions on the underlying table are checked by SQL Server, and an error is returned if the user does not have appropriate permissions to the underlying table. This is illustrated in Figure 4-3.

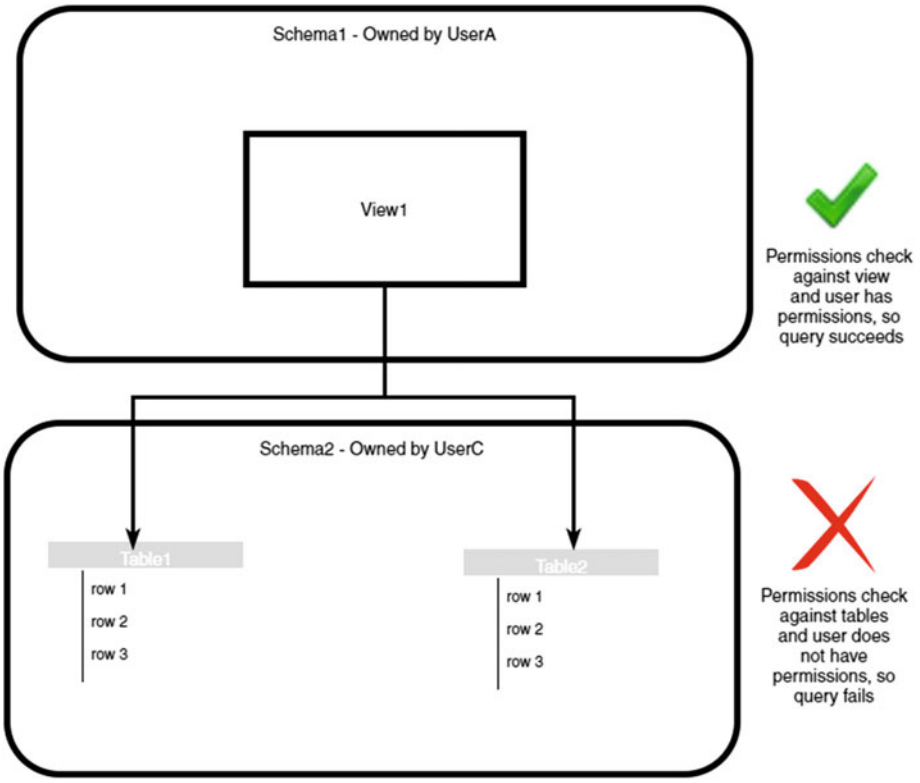


Figure 4-3. *Broken ownership chain*

■ **Caution** It is important to note that ownership chains lead to `DENY` assignments being bypassed. This is because neither the `GRANT` or `DENY` assignments of the user are evaluated.

Impersonation

Impersonation refers to the practice of executing T-SQL statements or code modules under the context of a different security principal. This helps you to enforce the principal of least privilege by assigning fewer permissions to users, but elevating those permissions at the point when a section of code is executed.

In SQL Server, impersonation can be implemented through the `EXECUTE AS` clause. The `EXECUTE AS` clause can be placed in the header of a stored procedure, function, or DML trigger. `EXECUTE AS` can also be used during a session to change the security context. Table 4-1 describes the context specifications that can be specified when using `EXECUTE AS`.

Table 4-1. *EXECUTE AS Context Specifications*

Usage	Context Specification
Session	• LOGIN • USER
Procedures, functions and DML triggers	• CALLER • SELF • OWNER • USER
Database-level DDL triggers	• CALLER • SELF • USER
Server-level DDL triggers	• CALLER • SELF • LOGIN
Queues	• CALLER • SELF • USER

Table 4-2 explains the usage of each of the context specifications.

Table 4-2. *Context Specification Usage*

Context Specification	Description
CALLER	The code executes under the original context. This is the default behavior for all modules, except queues.
SELF	The code executes under the context of the principal that created or last altered the module.
OWNER	The code executes under the context of the principal that owns the module or the schema in which the module resides.
USER	The code executes under the context of a specific database user.
LOGIN	The code runs under the context of a specific login.

The script in Listing 4-3 demonstrates the `EXECUTE AS` functionality by using a system function named `SUSER_SNAME()`. This function returns the name of a login from a SID that is passed a parameter. If no parameter is passed, then it returns the name of the login of the current security context.

Listing 4-3. Change Security Context

```
--Execute under current security context
SELECT SUSER_SNAME() ;

--Switch to the context of Danni
EXECUTE AS USER = 'Danni' ;

--Execute under Danni's security context
SELECT SUSER_NAME() ;
```

The results of this query are shown in Figure 4-4. As you can see, the first query ran under the context of my login. After using the `EXECUTE AS` statement, however, the security context changed to Danni.

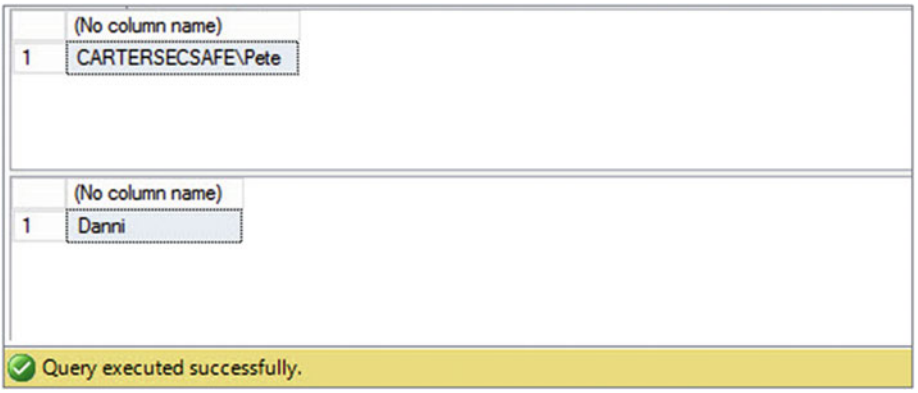


Figure 4-4. *Results of change security context query*

To revert back to the original security context, the code must use the `REVERT` statement. If no `REVERT` statement is supplied in the code, then the code continues to run under the modified context until the end of the session or code module.

■ **Caution** The user that creates the code module that contains the `EXECUTE AS` clause, or the user executing the ad hoc SQL within a session using the `EXECUTE AS` clause, must have the `IMPERSONATE` permission on the security context that the code runs under.

Row-Level Security

Implemented in SQL Server 2016, Row-Level Security (RLS) allows DBAs to simplify the management of fine-grain security by providing an out-of-the-box technology. In many cases, security is implemented in the middle tier and the application connects to SQL Server using a single login. RLS can also assist with improving architectural principals by pushing logic and security to the back end.

RLS is implemented through a security policy and security predicates. The following sections introduce each of these concepts before demonstrating how the technology could be implemented in the AdventureWorks2016 database.

Security Predicates

A *security predicate* is a function that is applied to a result set to determine which rows can be returned or modified by the user accessing the data. The functions are inline table-valued functions that must be created by the DBA. There are two types of security predicates that can be implemented: filter predicates and block predicates.

Filter predicates filter the rows that are returned to a user when they query a table or view. This type of predicate is silent, meaning that the user or application is given no indication that rows have been filtered from the result set. Filter predicates affect `SELECT`, `UPDATE`, and `DELETE` statements.

Unlike filter predicates, block predicates return an error if they are violated. This type of predicate explicitly blocks `INSERT`, `UPDATE`, and `DELETE` statements, which violate the predicate. For `UPDATE` statements, block predicates can be defined as `BEFORE` or `AFTER`. When defined as `BEFORE`, the predicate is applied based on the original value. When defined as `AFTER`, the predicate is applied based on the value of a tuple after the `UPDATE` statement has been applied. As you would expect, if the predicate is for an `INSERT` statement, then `AFTER` is the only option, and if the predicate is for a `DELETE` statement, `BEFORE` is the only option.

It is a good idea to create a new schema in which to place your security predicates. This is because any user should be able to access the functions, and placing the RLS objects in a separate schema makes it easy to manage these permissions.

When creating the function, it is also a good idea to use `SCHEMABINDING`. This is because any function calls or joins to other tables can be made without additional permission configuration. If you do not use `SCHEMABINDING`, then `SELECT` or `EXECUTE` permissions are required on the referenced objects by users calling the security predicate.

■ **Note** If you use `SCHEMABINDING`, then it is not possible to alter the columns in the table or view that are referenced by the security predicate.

Security Policies

A security policy binds the security predicate(s) to tables and views. It is the security policy that invokes the security predicate and specifies how the predicate should be used (filter, block before, block after).

A security policy can be created using the `CREATE SECURITY POLICY` statement. Table 4-3 describes the arguments that can be specified when creating a security policy.

Table 4-3. *CREATE SECURITY POLICY Arguments*

Argument	Description
`schema_name.security_policy_name`	The name assigned to the security policy and the schema in which it should be created.
`ADD`	Specifies if the predicate should be `FILTER` or `BLOCK`.
`PREDICATE`	The two-part name of the security predicate.
`column_name | expression`	The column name or expression used as the input parameter for the security predicate.
`table_schema.table_name`	The two-part name of the target table to which the security policy is applied.
`block_DML_operations`	If the `ADD` argument set to `BLOCK`, then the DML operations to block is defined.
`STATE`	Specifies whether the security policy is enabled on creation.
`SCHEMABINDING`	Specifies whether security predicates that are bound to the security policy must be created with `SCHEMABINDING`.
`NOT FOR REPLICATE`	Specifies that the security policy should not be executed when a replication agent modifies the target object.

Implementing RLS

This section discusses how RLS can be implemented. Imagine that you are required to allow managers to view information in the `HumanResources.Employee` table of the AdventureWorks2016 database. The challenge using traditional permission assignments, however, is that they should only be able to view the information on employees who report to them (either directly or indirectly).

You can achieve this using RLS, first by creating a security predicate and a security policy. The security predicate defines which rows are accessible by a user based on the `OrganizationNode` column of the table.

■ **Tip** The `OrganizationNode` column of the `HumanResources.Employee` table uses the `HierarchyID` data type. HierarchyID is a complex data type implemented through CLR; it was first implemented in SQL Server 2008. It exposes a variety of methods to assess a row's level of the hierarchy. A full method reference for the HierarchyID data type can be found at https://msdn.microsoft.com/en-us/library/bb677193.aspx.

The first step is to create the security predicate. Listing 4-4 is code used to write such a predicate function. Notice that before creating the predicate function, the script creates a new schema, called Security, in which the function resides. This is in line with the best practices described in the "Security Predicates" section.

Listing 4-4. Create a Security Predicate

```
USE AdventureWork2016
GO

CREATE SCHEMA Security ;
GO

ALTER FUNCTION Security.fn_securitypredicate(@OrganizationNode HIERARCHYID)
    RETURNS TABLE
WITH SCHEMABINDING
AS
RETURN SELECT 1 AS fn_securitypredicate_result
FROM HumanResources.Employee e1
WHERE @OrganizationNode.IsDescendantOf(OrganizationNode) = 1
AND LoginID = 'adventure-works\' + SUSER_SNAME() ;
GO
```

You now need to create the security policy. This is demonstrated in Listing 4-5.

Listing 4-5. Create a Security Policy

```
CREATE SECURITY POLICY Security.EmployeeSecurityPolicy
ADD FILTER PREDICATE Security.fn_securitypredicate(OrganizationNode) ON
HumanResources.Employee
WITH (STATE=ON, SCHEMABINDING=ON) ;
```

If you were now to run the script in Listing 4-6, only employees who report to Roberto are returned.

Listing 4-6. Test the RLS

```
EXECUTE AS USER = 'Roberto0'
  SELECT * FROM HumanResources.Employee ;
REVERT
```

The results of this query are illustrated in Figure 4-5.

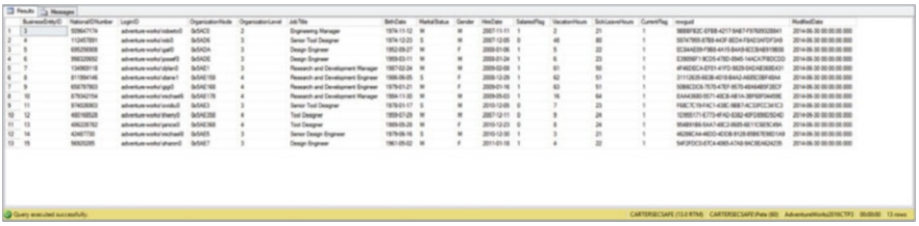


Figure 4-5. *RLS rest results*

Dynamic Data Masking

Dynamic data masking is a technology that was introduced in SQL Server 2016 that allows non-privileged users to see only a subset of an atomic value stored within a tuple. For example, imagine a call center for a credit card company. For data protection, the call center operatives are not authorized to see an entire credit card number. They need to identify the customer, however, and one of the questions used for the security checks is the last four numbers of the credit card number.

In this scenario, dynamic data masking could be used on the credit card number column so that all but the last four digits are obfuscated. This can help improve application architecture by pushing code from the middle tier to the back end, which improves reusability and reduces resource consumption in the middle tier.

Table 4-4 describes the dynamic masking functions that are available in SQL Server 2016.

Table 4-4. *Dynamic Data Masking Functions*

Function	Supported Data Types	Description
`default`	`char, nchar, varchar, nvarchar, text, ntext, bigint, bit, decimal, int, money, numeric, smallint, smallmoney, tinyint, float, real, date, datetime2, datetime, datetimeoffset, smalldatetime, time, binary, varbinary, image`	Fully masks a value. The type of masking depends on the data type of the value.
`partial`	`char, nchar, varchar, nvarchar`	Accepts a prefix, a masking value, and a suffix.
`email`	`char, nchar, varchar, nvarchar`	Reveals only the first letter of the e-mail address, the `@` symbol, and the domain suffix.
`random`	`bigint, decimal, int, numeric, smallint, smallmoney, tinyint, float, real`	Replaces a value with a random value from within a specified range.

Dynamic data masking can be implemented by using the `MASKED WITH` syntax in either a `CREATE TABLE` or an `ALTER COLUMN` statement. For example, the statement in Listing 4-7 adds a mask to the `Sales.CreditCard` table in the AdventureWorks2016 database so that users only see the last four digits of credit card numbers when the `CardNumber` column is queried.

Listing 4-7. Add a Data Mask

```
USE AdventureWorks2016
GO

ALTER TABLE Sales.CreditCard
ALTER COLUMN CardNumber ADD MASKED WITH (FUNCTION = 'partial(0,"XXXX-XXXX-
XXXX-",4)');
```

Let's take a look at dynamic data masking in action. Let's assume that the user brian3 has `SELECT` privileges to the `Sales.CreditCard` and runs the query shown in Listing 4-8.

Listing 4-8. Query the Sales.CreditCard Table

```
EXECUTE AS USER = 'brian3' ;
SELECT TOP 10
       CreditCardID
      ,CardType
      ,CardNumber
      ,ExpMonth
      ,ExpYear
      ,ModifiedDate
  FROM Sales.CreditCard ;
REVERT
```

The results of this query are shown in Figure 4-6.

Results | Messages

	CreditCardID	CardType	CardNumber	ExpMonth	ExpYear	ModifiedDate
1	1	SuperiorCard	XXXX-XXXX-XXXX-5310	11	2006	2013-07-29 00:00:00.000
2	2	Distinguish	XXXX-XXXX-XXXX-9722	8	2005	2013-12-05 00:00:00.000
3	3	ColonialVoice	XXXX-XXXX-XXXX-8353	7	2005	2014-01-14 00:00:00.000
4	4	ColonialVoice	XXXX-XXXX-XXXX-8248	7	2006	2013-05-20 00:00:00.000
5	5	Vista	XXXX-XXXX-XXXX-0042	4	2005	2013-02-01 00:00:00.000
6	6	Distinguish	XXXX-XXXX-XXXX-6181	9	2006	2014-04-10 00:00:00.000
7	7	Distinguish	XXXX-XXXX-XXXX-1028	6	2007	2013-02-01 00:00:00.000
8	8	SuperiorCard	XXXX-XXXX-XXXX-3101	7	2007	2013-06-30 00:00:00.000
9	9	Distinguish	XXXX-XXXX-XXXX-5901	2	2005	2013-09-23 00:00:00.000
10	10	SuperiorCard	XXXX-XXXX-XXXX-6493	8	2008	2011-08-31 00:00:00.000

Figure 4-6. *Results of masked query*

To reveal the full value, users must be granted the UNMASK permission. For example, imagine that user brian3 was granted the UNMASK permission, as demonstrated in Listing 4-9. The user is now able to see the whole value of the credit card number.

Listing 4-9. Grant the UNMASK Permission

```
GRANT UNMASK TO brian3 ;
```

Summary

SQL Server provides a rich suite of functionality for assisting in the management of data level security. Schemas provide a namespace for objects. When organized by business area (as opposed to technical relationship), they can simplify the administration of security, because it allows you to assign permissions based on business role.

Each new version of SQL Server introduces new security features—and SQL Server 2016 is no exception. Row-Level Security (RLS) introduces the ability to restrict the rows within a table based on a user's security attributes, such as username or session context.

Dynamic data masking allows non-privileged users to see a partially obfuscated value rather than the full value within a column. For example, a call center operative can see only the last four digits of a customer's credit card number, as opposed to the full number.

CHAPTER 5

Encryption in SQL Server

Encryption is a process of obfuscating data with the use of an algorithm that uses keys and certificates. This means that if security is bypassed and data is accessed or stolen by attackers, then it is useless, unless the keys that were used to encrypt it are also acquired. This adds an additional layer of security over and above access control, but it does not replace the requirement for an access control strategy. Encrypting data also has the potential to degrade performance and increase the size of data, so you should use it on the basis of need, as opposed to implementing it on all data as a matter of routine.

This chapter begins with an overview of encryption concepts. You then review the SQL Server encryption hierarchy before a demonstration on how to implement Transparent Data Encryption. The chapter also covers cell-level encryption and Always Encrypted, a technology introduced in SQL Server 2016 that helps isolate encryption keys from the data that they secure.

Generic Encryption Concepts

The following sections introduce the generic encryption concepts of symmetric keys, asymmetric key, certificates, and the Windows Data Protection API.

Defense-in-Depth

Defense-in-depth is a technique used across the IT landscape. It refers to implementing multiple layers of security. For example, a company likely has a perimeter firewall on the outskirts of the network. There may then be further firewalls inside the network, between data centers or network blocks. From the SQL Server perspective, defense-in-depth is achieved by using an encryption strategy to supplement the access control strategy. It does not replace the need for access control, but it does provide an additional layer of defense.

Symmetric Keys

A *symmetric key* is an algorithm to encrypt data. It is the weakest form of encryption because it uses the same algorithm for both encrypting and decrypting the data. Although it is the weakest form of encryption, it is also the method that has the least performance overhead. You can encrypt a symmetric key with a password, with another key or with a certificate.

P. A. Carter, *Securing SQL Server*, DOI 10.1007/978-1-4842-2265-2_5

Asymmetric Keys

Unlike a symmetric key, which uses the same algorithm to decrypt or encrypt data, an *asymmetric key* uses a pair of keys (algorithms). One of the keys is used only for encryption and the other is used only for decryption. The key that is used to encrypt the data is called the *private key* and the key that is used to decrypt the data is known as the *public key*.

Certificates

A *certificate* is issued by a trusted source, known as a *certificate authority* (CA). It uses an asymmetric key, but also provides a digitally signed statement that binds the public key to a principal or device, which holds the corresponding private key.

Self-Signed Certificates

A *self-signed certificate* is a certificate that has been signed by the same entity that its identity certifies. Self-signed certificates can be created by SQL Server.

Windows Data Protection API

The Windows Data Protection API (DPAPI) is a cryptographic application programming interface (API) that ships with the Windows operating system. It allows keys to be encrypted by using user secret information or domain secret information. DPAPI is used to encrypt the service master key, which is the top level of the SQL Server encryption hierarchy. The service master key is discussed in the "SQL Server Encryption Concepts" section of this chapter.

SQL Server Encryption Concepts

SQL Server's cryptography functionality relies on a hierarchy of keys and certificates. The root level of the hierarchy is the service master key. The following sections describe the use of master keys and EKM (Extensible Key Management), as well as SQL Server's encryption hierarchy.

Master Keys

The root level of the SQL Server encryption hierarchy is the service master key. The service master key is created automatically when the instance is built. It is used to encrypt database master keys, credentials, and the passwords for linked servers by using the DPAPI. The service master key is stored in the master database. There is always one service master key per instance. Since SQL Server 2012, the service master key has been a symmetric key generated using the AES 256 algorithm. Older versions of SQL Server used the Triple DES algorithm.

■ **Tip** When you upgrade an instance from SQL Server 2008 R2 or lower, it is good practice to regenerate the key due to the encryption algorithm used in SQL Server 2012 and higher.

If you need to regenerate the service master key, then all keys within the instance's encryption hierarchy must be decrypted and then re-encrypted. This means that every key and certificate that is encrypted directly or indirectly from the master key must be regenerated. This is a very resource-intensive process and should only be attempted during a maintenance window.

You can regenerate the service master key using the command in Listing 5-1. You should be aware that if the process fails to decrypt and re-encrypt any key that is below it in the hierarchy, then by default, the whole regeneration process fails. You can change this behavior by using the `FORCE` keyword, which forces the process to continue after errors.

■ **Caution** Be warned that using the `FORCE` keyword leaves any data that cannot be decrypted and re-encrypted unusable. There is no way to regain access to this data.

Listing 5-1. Regenerate the Service Master Key

```
ALTER SERVICE MASTER KEY REGENERATE ;
```

Because the service master key is crucial, it is very important to back it up after building a new instance and after the key is regenerated. You should then store the backup in a secure offsite location, so that it is available in disaster recovery scenarios. You can also restore the backup of this key when you migrate an instance to a different server to avoid issues with the encryption hierarchy. The script in Listing 5-2 demonstrates how to back up and restore the service master key. If the master key you restore is identical, then SQL Server lets you know and data does not need to be decrypted and re-encrypted.

■ **Tip** If your instance does not use any encryption features, then a backup of the service master key is not required.

Listing 5-2. Back up and Restore the Service Master Key

```
--Backup Service Master Key

BACKUP SERVICE MASTER KEY
TO FILE = 'c:\keys\service_master_key'
ENCRYPTION BY PASSWORD = 'Pa$$w0rd' ;

--Restore Service Master Key
```

```
RESTORE SERVICE MASTER KEY
FROM FILE = 'c:\keys\service_master_key'
DECRYPTION BY PASSWORD = 'Pa$$w0rd' ;
```

■ **Tip** `service_master_key` is the name of the key file, not a folder. By convention, it does not have an extension. If you are following along with the demonstrations, then remember to change the filepath to match your own configuration.

A *database master key* is a symmetric key, encrypted using the AES 256 algorithm. The service master key is used to encrypt private keys and certificates that are stored within a database. It is encrypted using a password as the secret; but a copy is created, which is encrypted using the service master key. This allows the service master key to be opened automatically when required. If this copy does not exist, then you need to open it manually.

If the copy does not exist or is corrupt, the key needs to be explicitly opened in order for you to use a key that is below it in the hierarchy (a key that has been encrypted, using the service master key). Copies of the service master key are stored within the database and the master database.

It is as important to back up a service master key as it is to back up a service master key, because losing the key results in data loss for any data that is below it in the encryption hierarchy. In some cases, this could be an entire database. The script in Listing 5-3 demonstrates how to create a service master key for the AdventureWorks2016 database. It then backs up the key and attempts to restore it. The `FORCE` keyword can be used for service master keys in the same way it can be used for a service master key. This keyword forces the decrypt and re-encrypt process to continue on error. There is a possibility of data loss, however.

■ **Tip** If you are following along with the examples, remember to change the filepath to match your own configuration.

Listing 5-3. Administering a Database Master Key

```
USE AdventureWorks2016
GO

CREATE MASTER KEY ENCRYPTION BY PASSWORD = 'Pa$$w0rd' ;

BACKUP MASTER KEY TO FILE = 'c:\keys\Chapter5_master_key'
ENCRYPTION BY PASSWORD = 'Pa$$w0rd';

RESTORE MASTER KEY
FROM FILE = 'c:\keys\Chapter5_master_key'
```

```
DECRYPTION BY PASSWORD = 'Pa$$w0rd' --The password in the backup file
ENCRYPTION BY PASSWORD = 'Pa$$w0rd' ; --The password it will be encrypted
within the
database
```

EKM and Key Stores

An Extensible Key Management (EKM) module allows you to generate and manage keys and certificates used to secure SQL Server data within a third-party hardware security module (HSM). The EKM module provides the interface with SQL Server by using the Microsoft Cryptographic API (MS-CAPI). This is more secure because the key is not being stored with the data. It also means that you can benefit from advanced features that may be offered by a third-party vendor, such as key rotation and secure key disposal. When using an HSM, you may also witness improved performance because the encryption and decryption of keys are hardware-based.

Key stores provide secure storage and a trusted source for keys and certificates. Windows Certificate Store provides the functionality within your own Windows Server enterprise. Azure Key Vault offers key storage within Windows Azure. There are also third-party and open source key store providers, such as Amazon Key Management Services (which is a service within the Amazon Web Services ecosystem), KeyWhiz, and Vault, to name but a few.

SQL Server Encryption Hierarchy

Figure 5-1 illustrates the encryption hierarchy in SQL Server.

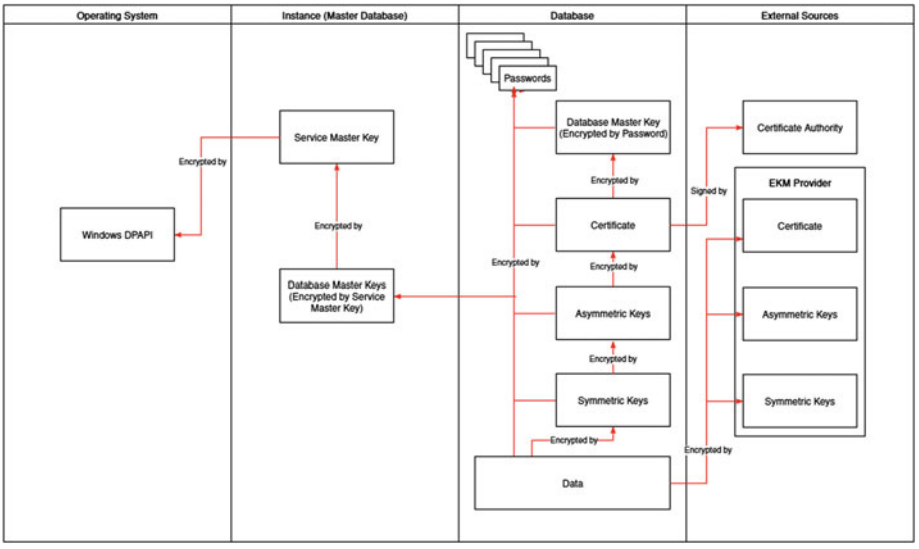


Figure 5-1. *SQL Server encryption hierarchy*

Encrypting Data

Data can be encrypted in SQL Server by using either a password or the encryption hierarchy. The following sections discuss each of these approaches.

Encrypting Data with a Password or a Passphrase

The most basic level of encrypting data in SQL Server is the `ENCRYPTBYPASSPHRASE()` function, which allows you to encrypt data by directly using a password or a passphrase, rather the SQL Server encryption hierarchy.

To illustrate this, let's look at the `Sales.CreditCard` table in the AdventureWorks2016 database. This table stores information on customers' credit card information in plain text, which is not a great idea from a security perspective and it may also be against regulatory requirements on data protection.

Imagine that the company's compliance department has noticed this issue during an audit and has tasked you with encrypting the credit card number column. To encrypt this column, you need to perform the following tasks:

- Create a new column of type VARBINARY
- Encrypt the values in the `CardNumber` column and insert them into the new column
- Drop the original column
- Update queries and ETL processes to use the new column

■ **Tip** If you have been following the examples in previous chapters, then you should remove dynamic data masking from the CardNumber column before continuing. This can be achieved with the script in Listing 5-4.

Listing 5-4. Drop Dynamic Data Mask

```
USE AdventureWorks2016
GO

ALTER TABLE Sales.CreditCard
ALTER COLUMN CardNumber DROP MASKED ;
```

The first task is to add a new column to the table. This can be achieved using the script in Listing 5-5. Because the column initially has no values and there is no `DEFAULT` constraint, you should allow `NULL` values. This can be changed once the column is populated.

Listing 5-5. Add a New Column to Hold the Encrypted Credit Card Numbers

```
USE AdventureWorks2016
GO

ALTER TABLE Sales.CreditCard ADD
        CardNumberEncrypted varbinary(8000) NULL ;
```

The next task is to populate the new column. To achieve this, encrypt the values of the `CardNumber` column, using the `ENCRYPTBYPASSPHRASE()` function. This function accepts the parameters described in Table 5-1.

Table 5-1. *ENCRYPTBYPASSPHRASE() Parameters*

Parameter	Description
`passphrase`	The password or phrase to generate a symmetric key.
`cleartext`	The value to be encrypted.
`add_authenticator`	Specifies if an authenticator should be used.
`authenticator`	The value to be used to derive an authenticator.

The script in Listing 5-6 demonstrates how the `CardNumberEncrypted` column can be populated.

Listing 5-6. Populate the Encrypted Column

```
UPDATE Sales.CreditCard
SET CardNumberEncrypted = ENCRYPTBYPASSPHRASE('Pa$$w0rd', CardNumber, 0) ;
```

You can now set the `CardNumberEncrypted` column to not allow `NULL` values and to drop the original column. This is demonstrated in Listing 5-7.

■ **Tip** Do not run the script in Listing 5-7 if you plan to follow further examples in this chapter, because you are reusing the `CardNumber` column.

Listing 5-7. Set Encrypted Column NOT NULL and Drop Original Column

```
--Set CardNumberEncrypted column to be NOT NULL
ALTER TABLE Sales.CreditCard
ALTER COLUMN CardNumberEncrypted VARBINARY(256) NOT NULL ;

--Do not run following section, if you plan to follow later examples
```

```
DROP INDEX AK_CreditCard_CardNumber ON Sales.CreditCard ;
GO

ALTER TABLE Sales.CreditCard
        DROP COLUMN CardNumber ;
```

Changing ETL processes and queries depends on how your database is being used, of course. The AdventureWorks2016 database is an OLTP database, so it is likely that credit card numbers are updated either by salespeople, or by customers directly, as opposed to via ETL process. There may be downstream ETL processes, however, which move the data into a data warehouse or archived database.

Let's assume that there is a stored procedure that was previously used to return customers' credit card information from a web portal. This fictional stored procedure is shown in Listing 5-8. Assume that the user's `BusinessEntityID` has been determined elsewhere in the front-end app based upon login information.

Listing 5-8. Return Credit Card Information

```
CREATE PROCEDURE ReturnCredCardInfo @BusinessEntityID INT
AS
BEGIN
        SELECT p.BusinessEntityID, p.firstName, p.LastName, cc.CardNumber,
cc.CardType, cc.ExpMonth, cc.ExpYear
        FROM Person.Person p
        INNER JOIN Sales.PersonCreditCard pcc
                ON p.BusinessEntityID = pcc.BusinessEntityID
                INNER JOIN Sales.CreditCard cc
                        ON pcc.CreditCardID = cc.CreditCardID
        WHERE p.BusinessEntityID = @BusinessEntityID ;
END
```

To work with the new, encrypted column, you need to modify the stored procedure to use the `DECRYPTBYPASSPHRASE()` function. This function accepts the parameters described in Table 5-2.

Table 5-2. *DECRYPTBYPASSPHRASE() Parameters*

Parameter	Description
`passphrase`	The password or phrase to decrypt the data.
`ciphertext`	The value to be decrypted.
`add_authenticator`	Specifies whether an authenticator is required to decrypt the data.
`authenticator`	The authenticator data.

The script in Listing 5-9 demonstrates how to rewrite the procedure. Note that in addition to decrypting the column, you must also convert the result back to an NVARCHAR value for meaningful results to be returned. You know that 25 characters is sufficient for the NVARCHAR value, because it is the length of the original CardNumber column.

Listing 5-9. Modify the Procedure to Work with the Encrypted Column

```
ALTER PROCEDURE ReturnCredCardInfo @BusinessEntityID INT
AS
BEGIN
        SELECT p.BusinessEntityID
                , p.firstName
                , p.LastName
                , CONVERT(NVARCHAR(25),
                        DECRYPTBYPASSPHRASE('Pa$$wOrd',cc.
CardNumberEncrypted, 0)
                  )
                ,  cc.CardType
                , cc.ExpMonth
                , cc.ExpYear
        FROM Person.Person p
        INNER JOIN Sales.PersonCreditCard pcc
                ON p.BusinessEntityID = pcc.BusinessEntityID
                INNER JOIN Sales.CreditCard cc
                        ON pcc.CreditCardID = cc.CreditCardID
        WHERE p.BusinessEntityID = @BusinessEntityID ;
END
```

This approach still leaves a security hole, however. The data is decrypted by the application, so although the user may not have the permissions to see some data, all of the data is decrypted. This is appropriate in some scenarios, such as when a sales team manages credit card information and customers do not have direct access to the application.

Imagine a scenario, however, where you want users to be able to manage their own credit card information. In this instance, you might want to ensure that all data remains encrypted, except for a user's own credit card number.

To implement this strategy, when a user inputs his credit card information, it is encrypted using the password that the customer uses to log in to the application. The front-end application can simply pass the credit card number and the user's password to a stored procedure, via parameters. Listing 5-10 describes two stored procedures. The first can be used by the front-end application to add a new credit card. The second returns the credit card number. Notice that the ENCRYPTBYPASSPHRASE() and DECRYPTBYPASSPHRASE() functions accept variables as parameters, as well as hard-coded strings.

Listing 5-10. Encrypt and Decrypt Data Based Upon a User's Password

```
USE AdventureWorks2016
GO
CREATE PROCEDURE dbo.AddCreditCard
                                    @BusinessEntityID         INT
                                    ,@
CreditCardNumber        NVARCHAR(25)
                                    ,@
CardType                NVARCHAR(50)
                                    ,@ExpMonth                TINYINT
                                    ,@ExpYear                 SMALLINT
                                    ,@
Password                NVARCHAR(128)
AS
BEGIN
        DECLARE @CreditCardID        INT ;
        BEGIN TRANSACTION
                INSERT INTO Sales.CreditCard
                                ( CardType ,
                                  ExpMonth ,
                                  ExpYear ,
                                  ModifiedDate,
                                  CardNumberEncrypted
                                )
                VALUES  ( @CardType,
                                  @ExpMonth,
                                  @ExpYear,
                                  SYSDATETIME(),
                                  ENCRYPTBYPASSPHRASE(@Password, @
CreditCardNumber, 0)
                                ) ;
                SET @CreditCardID = @@IDENTITY ;

                INSERT INTO Sales.PersonCreditCard
                                ( BusinessEntityID ,
                                  CreditCardID ,
                                  ModifiedDate
                                )
                VALUES  ( @BusinessEntityID,
                                  @CreditCardID,
                                  SYSDATETIME()
                                ) ;
        COMMIT
END
GO
```

```
CREATE PROCEDURE ReturnCredCardInfo
                                        @BusinessEntityID       INT
                                        ,@
Password                NVARCHAR(128)
AS
BEGIN
        SELECT
                CONVERT(NVARCHAR(25), DECRYPTBYPASSPHRASE(@Password,cc.
CardNumberEncrypted, 0)) AS CreditCardNumber
        FROM Person.Person p
        INNER JOIN Sales.PersonCreditCard pcc
                ON p.BusinessEntityID = pcc.BusinessEntityID
                INNER JOIN Sales.CreditCard cc
                        ON pcc.CreditCardID = cc.CreditCardID
        WHERE p.BusinessEntityID = @BusinessEntityID ;
END
GO
```

Encrypting Data with Keys and Certificates

When encrypting data using the SQL Server encryption hierarchy, data can be encrypted using a symmetric key, an asymmetric key, or a certificate. Table 5-3 describes the functions that are exposed by SQL Server for encrypting and decrypting data using keys and certificates.

■ **Tip** Keys and certificates within the hierarchy can be encrypted using further keys and certificates.

Table 5-3. *Cryptographic Functions*

Encryption Type	Encryption Function	Decryption Function
Symmetric	`ENCRYPTBYKEY()`	`DECRYPTBYKEY()`
Asymmetric	`ENCRYPTBYASYKEY()`	`DECRYPTBYASYKEY()`
Certificate	`ENCRYPTBYCERT()`	`DECRYPTBYCERT()`

■ **Tip** For performance reasons, you should always use a symmetric key, unless there is a very good reason (usually a regulatory requirement) not to.

To demonstrate how to encrypt data using a symmetric key, you first create a certificate. You then create a symmetric key, which is encrypted using this new certificate, in the AdventureWork2016 database. Next, you update the `CreditCardNumberEncrypted` column so that the credit card numbers are encrypted using this symmetric key, as opposed to a passphrase.

The `CREATE CERTIFICATE` T-SQL statement accepts the arguments described in Table 5-4 when being used to generate a new key.

Table 5-4. *CREATE CERTIFICATE Arguments*

Argument	Description
AUTHORIZATION	Specifies the owner of the certificate.
ACTIVE FOR BEGIN_DIALOG	Specifies if the certificate can be used to initiate a Service Broker conversation.
ENCRYPTION BY PASSWORD	Specifies the password that is used to encrypt the certificate's private key.
WITH SUBJECT	Specifies a subject for the certificate.
START_DATE	Specifies a date on which the certificate becomes valid.
EXPIRY_DATE	Specifies a date on which the certificate expires, after which it is no longer valid.

■ **Tip** The `CREATE CERTIFICATE` statement can also be used to import a certificate that is stored within an assembly or to create a certificate that uses existing keys stored within a file. For information on the available arguments when using these options, please refer to `https://msdn.microsoft.com/en-us/library/ms187798.aspx`.

The `CREATE SYMMETRIC KEY` T-SQL statement accepts the arguments described in Table 5-5.

Table 5-5. *CREATE SYMMETRIC KEY Arguments*

Argument	Description
AUTHORIZATION	Specifies the owner of the key.
FROM PROVIDER	If the key is managed by an EKM provider, specifies the EKM provider to use.
KEY_SOURCE	Specifies a passphrase from which to generate the key.
IDENTITY_VALUE	Specifies a value from which to generate a GUID that can be used for temporary tagging data that is encrypted with a temporary key.
PROVIDER_KEY_NAME	Specifies the name by which the key is known to the EKM provider (if one is used).

(*continued*)

Table 5-5. (*continued*)

Argument	Description
CREATION_DISPOSITION	If an EKM provider is used, specifies if a new key should be created in the EKM or if an existing key should be used. The following are acceptable values: • CREATE_NEW: Specifies that a new key is created in the EKM provider. • OPEN_EXISTING: Specifies that an existing key is opened in the EKM provider.
ENCRYPTION BY	Specifies how the key is encrypted. The following are acceptable values: • CERTIFICATE (followed by the name of the certificate) • PASSWORD (followed by the password to use) • SYMMETRIC KEY (followed by the name of the key to use) • ASYMMETRIC KEY (followed by the name of the key to use)
ALGORITHM	Specifies the algorithm to encrypt the key. The following are acceptable values: • DES • TRIPLE_DES • TRIPLE_DES_3KEY • RC2 • RC4 • RC4_128 • DESX • AES_128 • AES_192 • AES_256

The `ENCRYPTBYKEY()` function accepts the parameters described in Table 5-6.

Table 5-6. *ENCRYPTBYKEY() Parameters*

Parameter	Description
`key_GUID`	The GUID of the key that is used to encrypt the data.
`cleartext`	The value to be encrypted.
`add_authenticator`	Specifies if an authenticator should be used.
`authenticator`	The value to be used to derive an authenticator.

Listing 5-11 demonstrates how to create the symmetric key and use it to encrypt the CardNumberEncrypted column. Notice that you need to open the key before you use it. You then close the key after you have completed the activity.

Listing 5-11. Encrypt Data with a Symmetric Key

```
USE AdventureWorks2016
GO

--Create the certificate
CREATE CERTIFICATE CreditCardCert
WITH SUBJECT = 'Credit Card Numbers';
GO

--Create the symmetric key
CREATE SYMMETRIC KEY CreditCardKey
WITH ALGORITHM = AES_128
ENCRYPTION BY CERTIFICATE CreditCardCert;

--Open the key
OPEN SYMMETRIC KEY CreditCardKey
DECRYPTION BY CERTIFICATE CreditCardCert;

--Encrypt the data, using the symmetric key
UPDATE Sales.CreditCard
        SET CardNumberEncrypted = ENCRYPTBYKEY(Key_GUID('CreditCardKey'),
        CardNumber);

--Close the key
CLOSE SYMMETRIC KEY CreditCardKey ;
```

Data encrypted with a symmetric key can be decrypted using the DECRYPTBYKEY() function. This function accepts the parameters described in Table 5-7.

Table 5-7. *DECRYPTBYKEY() Parameters*

Parameter	Description
ciphertext	The value to be decrypted.
add_authenticator	Specifies if an authenticator is required to decrypt the data.
authenticator	The authenticator data.

The script in Listing 5-12 demonstrates how to use the DECRYPTBYKEY() function to read the CardNumberEncrypted column. Notice that you once again need to open and close the key.

Listing 5-12. Decrypt Data With DECRYPTBYKEY()

```
USE AdventureWorks2016
GO

--Open the key
OPEN SYMMETRIC KEY CreditCardKey
DECRYPTION BY CERTIFICATE CreditCardCert;

--Decrypt the data, using the symmetric key
SELECT CONVERT(NVARCHAR(30), DECRYPTBYKEY(CardNumberEncrypted)) AS
CreditCardNumber
FROM Sales.CreditCard ;

--Close the key
CLOSE SYMMETRIC KEY CreditCardKey ;
```

Transparent Data Encryption

When implementing a security strategy for your sensitive data, one important aspect to consider is the risk of data being stolen. Imagine a situation in which a privileged user with malicious intent uses detach/attach to move a database to a different instance, which they have created and therefore have `sysadmin` access to. The result is the user having permissions to data that they are not authorized to view.

Another potential scenario to consider is a malicious user gaining access to a backup of a database that contains data that he is not authorized to view. The user restores the backup file to an instance that he has created and has `sysadmin` access, and suddenly, he has the permissions to access the confidential data.

Transparent Data Encryption (TDE) protects against both of these scenarios by encrypting all data pages and the log file of a database. Data is encrypted using a symmetric key called the *database encryption key*. This key is stored in the boot record of the database and encrypted using a server certificate stored within the master database. This means that if the database is stolen, it cannot be decrypted, because the key used to decrypt it is stored in a different database.

■ **Caution** Obviously, if the master database or a backup of the server certificate is also stolen, then the data could be decrypted.

After you have enabled TDE on a database, the data and log pages are encrypted before they are written to disk. They are decrypted when they are read into memory. This means that the encryption is transparent to users, and applications do not need to be modified in order to access the data.

TDE also provides several other advantages over the encryption of data within columns. First, it does not cause bloat. A database encrypted with TDE is the same size that it was before it was encrypted. Also, although there is performance overhead, this is significantly less than the performance overhead that is caused cell-level encryption. The fact that developers do not need to modify their code to use TDE is another significant advantage in itself because it improves time-to-market (both for implementing TDE and for future application enhancements).

Considerations for TDE with Other Technologies

When planning the implementation of TDE, be mindful of how it interacts with other technologies. For example, you are able to encrypt a database that uses In-Memory OLTP, but the data within the In-Memory filegroup is not encrypted—even when data is persisted alongside the schema.

■ **Tip** Even though the memory optimized data is not encrypted, log records associated with in-memory transactions are encrypted.

It is also possible to encrypt databases that use FILESTREAM, but again, data within a FILESTREAM filegroup is not encrypted. If you use full-text indexes, new full-text indexes are encrypted. Existing full-text indexes are only encrypted after they are imported during an upgrade.

■ **Caution** Using full-text indexing with TDE, is not a good practice because data is written to disk in plain text during the full-text indexing scan operation. This leaves a window of opportunity for attackers to access sensitive data.

If your database is replicated, then it is important to manually enable TDE on the subscribers. This is because replication does not automatically send the data from a TDE-encrypted database to the subscribers in an encrypted form.

Due to the nature of `TempDB`, this system database is always encrypted using TDE, if any user database on the instance has TDE enabled. This stops potential attackers from stealing data at rest while it is spooled to `TempDB` or stored in a temporary table, and so forth. It does mean, however, that databases on the instance that are not enabled for TDE may still notice a performance penalty, which is caused by TDE.

TDE is incompatible with instant file initialization. Instant file initialization speeds up operations that create or expand files, because the files do not need to be zeroed out. If your instance is configured to use instant file initialization, then it no longer works for any files that are associated with any databases that you encrypt with TDE. This is because of a hard technical requirement for files to be zeroed out when TDE is enabled on a database.

Files used by buffer cache extensions are not encrypted by TDE. If you wish to encrypt the files associated with buffer cache extensions, then you must use system-level encryption tooling.

Implementing TDE

Implementing Transparent Data Encryption involves the following steps:

1. Create a service master key for the master database (If one does not already exist).
2. Create a certificate or asymmetric key in the master database.
3. Create a database encryption key in the database that you wish to encrypt.
4. Alter the database to enable Transparent Database Encryption.

■ **Note** The certificate or asymmetric key must be encrypted using the service master key in the master database. If you encrypt the certificate by password only, then SQL Server will not allow you to use it to encrypt the database encryption key.

■ **Tip** An asymmetric key can only be used if it is managed by an EKM.

When you enable TDE for a database, a background process moves through each page in every data file and encrypts it. This does not make the database inaccessible, but it does take out locks, which stop maintenance operations from taking place. While the encryption scan is in progress, the following operations cannot be performed:

- Dropping a file
- Dropping the database
- Taking the database offline
- Detaching a database
- Setting a database or filegroup as `READ_ONLY`

The operation to enable TDE will fail if any of the filegroups within a database are marked as `READ_ONLY`. This is because all pages within all files need to be encrypted when TDE is enabled, and this process involves changing the data within the pages to obfuscate them.

The script in Listing 5-13 follows the steps required to encrypt the AdventureWorks2016 database. The arguments accepted by the `CREATE DATABASE ENCRYPTION KEY` statement are described in Table 5-8.

Table 5-8. *CREATE DATABASE ENCRYPTION KEY Arguments*

Argument	Description
`WITH ALGORITHM`	Specifies the algorithm that should be used by the database encryption key. Acceptable values are as follows: • AES_128 • AES_192 • AES_256 • TRIPLE_DES_3KEY
`ENCRYPTION BY SERVER`	Specifies the certificate or asymmetric key that is used to encrypt the database encryption key. Acceptable values are as follows: • CERTIFICATE (followed by the name of the certificate to use) • ASYMMETRIC KEY (followed by the name of the asymmetric key to use)

Listing 5-13. Encrypt the AdventureWorks2016 Database

```
USE Master
GO

Create the Database Master Key (if it does not already exist)

CREATE MASTER KEY ENCRYPTION BY PASSWORD = 'Pa$$w0rd';
GO

--Create the Server Certificate

CREATE CERTIFICATE TDECert WITH SUBJECT = 'Certificate For TDE';
GO

USE AdventureWorks2016
GO

--Create the Database Encryption Key

CREATE DATABASE ENCRYPTION KEY
WITH ALGORITHM = AES_128
ENCRYPTION BY SERVER CERTIFICATE TDECert ;
GO
```

```
--Enable TDE on the database

ALTER DATABASE AdventureWorks2016
SET ENCRYPTION ON ;
GO
```

Administering TDE

When working with TDE-encrypted databases, there are administrative scenarios that you should be aware of. These are discussed in the following sections.

Backing up the Certificate

When configuring TDE, you are given a warning that the certificate used to encrypt the database encryption key has not been backed up. Backing up this certificate is critical and you should do so before you configure TDE or immediately afterward. If the certificate becomes unavailable, you have no way to recover the data within your database. You can back up the certificate by using the script in Listing 5-14.

Listing 5-14. Backing up the Certificate

```
BACKUP CERTIFICATE TDECert
TO FILE = 'C:\certificates\TDECert'
WITH PRIVATE KEY (file='C:\certificates\TDECertKey',
ENCRYPTION BY PASSWORD='Pa$$w0rd') ;
```

Migrating an Encrypted Database

Once TDE is enabled on a database, an attempt to attach or restore the database to a new instance will fail. Therefore, if you need to migrate a TDE-encrypted database to a new instance, you need to take the cryptographic artifacts into account.

Before migrating a database to a new instance, you must first create a service master key with the same password, and then restore the server certificate and private key to the new instance. You can restore the server certificate that you created earlier by using the script in Listing 5-15.

Listing 5-15. Preparing for a Database Migration

```
CREATE MASTER KEY ENCRYPTION BY PASSWORD = 'Pa$$w0rd' ;
GO
CREATE CERTIFICATE TDECert
FROM FILE = 'C:\Certificates\TDECert'
WITH PRIVATE KEY
(
FILE = 'C:\Certificates\TDECertKey',
DECRYPTION BY PASSWORD = 'Pa$$w0rd'
) ;
```

■ **Tip** Make sure that the SQL Server service account has permissions to the certificate and key files in the operating system; otherwise, you will receive an error stating that the certificate is not valid, does not exist, or that you do not have permissions to it. This means that you should check the restore immediately and periodically repeat the test.

Always Encrypted

Always Encrypted is a new technology introduced in SQL Server 2016. It is the first SQL Server encryption technology that protects data against privileged users, such as members of the `sysadmin` role. Because DBAs cannot view the encrypted data, Always Encrypted provides true segregation of duties. This can help with compliance issues for sensitive data when your platform support is outsourced to a third-party vendor. This is especially true if you have a regulatory requirement not to make your data available outside of your country's jurisdiction and the third-party vendor is using offshore teams.

Always Encrypted uses two separate types of keys: a *column encryption key* and a *column master key*. The column encryption key is used to encrypt the data within a column and the column master key is used to encrypt the column encryption keys.

■ **Tip** The column master key is a key or a certificate located within an external store.

Having the second layer of key means that SQL Server needs only to store an encrypted value of the column encryption key; it does not need to store it in plain text. The column master key is not stored in the database engine at all. Instead, it is stored in an external key store. The key store used could be an HSM (hardware security module), Windows Certificate Store, or an EKM provider, such as Azure Key Vault or Thales. SQL Server then stores the location of the column master key within the database metadata.

Instead of SQL Server being responsible for the encryption and decryption of data, this responsibility is handled by the client driver. Of course, this means that the application must be using a supported driver. See `https://msdn.microsoft.com/en-gb/library/mt147923.aspx` for information on working with supported drivers.

When an application issues a request that requires data to either be encrypted or decrypted, the client driver liaises with the database engine to determine the location of the column master key. The database engine also provides the encrypted column encryption key and the algorithm used to encrypt it.

The client driver can now contact the external key store and retrieve the column master key, which it uses to decrypt the column encryption key. The plain text version of the column encryption key can then be used to encrypt or decrypt the data, as required.

The entire process is transparent to the application, meaning that changes are not required to the application's code in order to use Always Encrypted. The only change that may be required is to use a later supported driver.

■ **Note** The client driver caches the plain text version of column encryption keys as an optimization, which attempts to avoid repeated round trips to the external key store.

The diagram in Figure 5-2 depicts the high-level architecture of Always Encrypted.

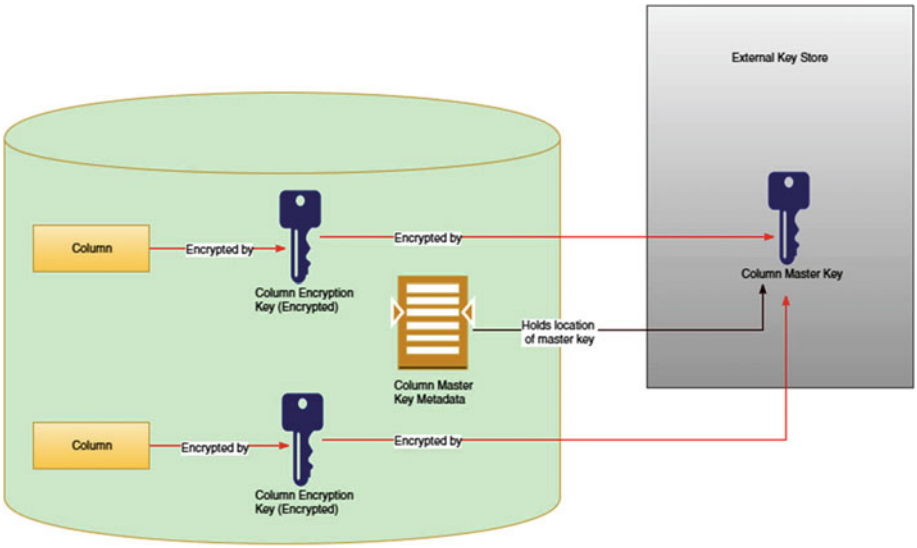


Figure 5-2. *Always Encrypted architecture*

Implementing Always Encrypted

When implementing Always Encrypted, the creation of tables with encrypted columns and the creation of key metadata are supported in T-SQL, PowerShell, or via the SSMS GUI. Other activities, however, such as provisioning keys and the actual encryption of data are only supported in PowerShell or via the SSMS GUI. They cannot currently be achieved with T-SQL. Therefore, this section demonstrates how to configure Always Encrypted via SSMS.

You will use Always Encrypted to secure the `CreditCardNumber`, `ExpMonth`, and `ExpYear` columns of the `Sales.CreditCard` table of the AdventureWorks2016 database. To achieve this, the first step is to create a column master key. You will use the Windows Certificate Store to store this key.

■ **Tip** If you are following along with the demonstrations in this chapter, you should delete the `Sales.usp_InsertSalesOrder_inmem` stored procedure before continuing. This is a natively compiled stored procedure, which is not supported by Always Encrypted.

In Object Explorer, drill though Databases ➤ AdventureWorks2016 ➤ Security ➤ Always Encrypted Keys and select New Column Master Key from the context menu of the Column Master Keys node. This causes the New Column Master Key dialog box to be invoked, as illustrated in Figure 5-3.

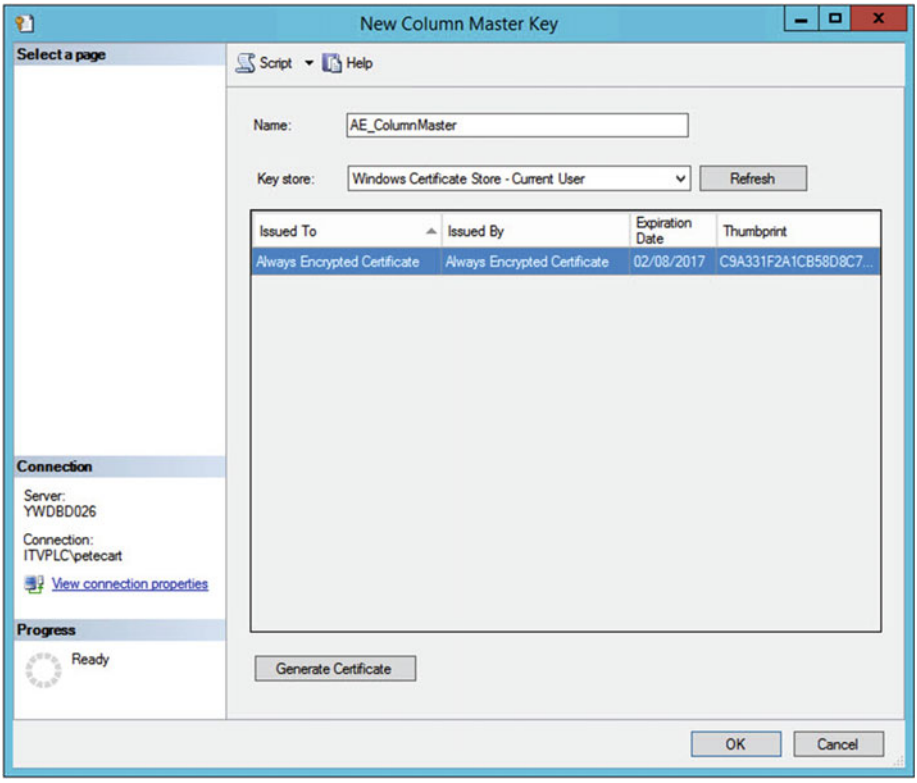


Figure 5-3. *New Column Master Key dialog box*

In this dialog box, you enter a name for the column master key and then selected the type of store that the key is stored in from the Key Store drop-down list. Table 5-9 describes all possible values for Key Store. You can choose an existing key or certificate, or alternatively use the Generate Certificate button to create a self-signed certificate in the appropriate store to use as the column master key. In this example, a self-signed certificate was generated.

Table 5-9. *Key Store Values*

Key Store Type	Description
Windows Certificate Store - Current User	The key or certificate is stored in the area of the Windows Certificate Store that is reserved for the profile of the user that created the certificate. This option may be appropriate if you use the database engine's service account interactively to create the certificate.
Windows Certificate Store - Local Machine	The key or certificate is stored in the area of the Windows Certificate Store that is reserved for the local machine.
Azure Key Vault	The key or certificate is stored in the Azure Key Vault EKM service.
Key Storage Provider (CNG)	The key or certificate is stored in an EKM store that supports Cryptography API: Next Generation.

If you generate the certificate, as opposed to selecting an existing certificate, it immediately appears within the chosen key store. For example, Figure 5-4 shows the certificate within the Current User area of the Windows Certificate Store.

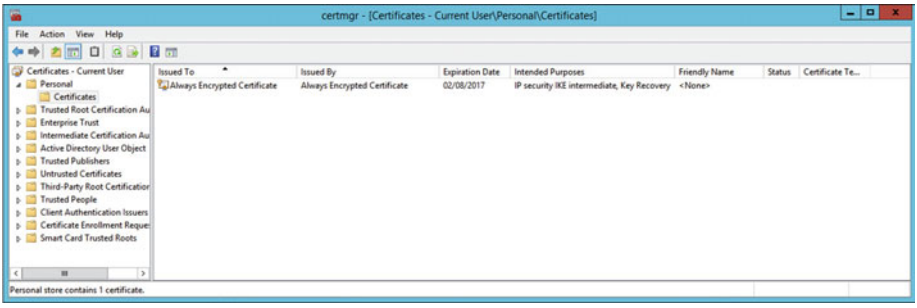


Figure 5-4. *Windows Certificate Store*

Now that the column master key has been created, you can generate a column encryption key. To do this, you select New Column Encryption Key from the context menu of the Databases ➤ AdventureWorks2016 ➤ Security ➤ Always Encrypted Keys ➤ Column Encryption Keys node in Object Explorer. This causes the New Column Encryption Key dialog box to be invoked, as illustrated in Figure 5-5.

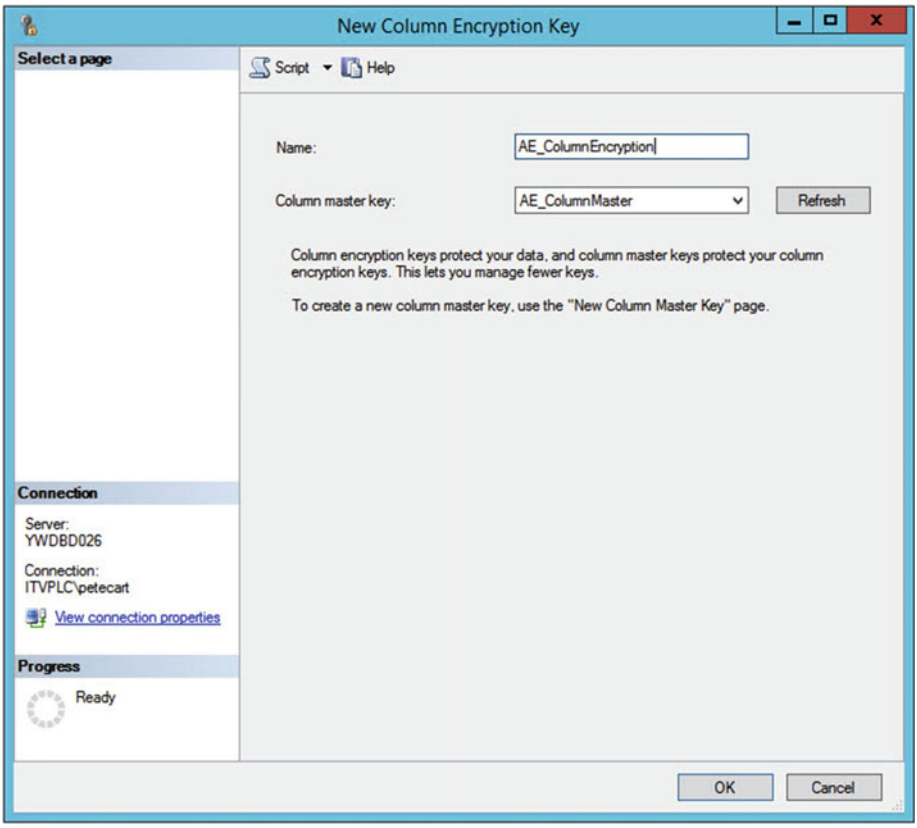


Figure 5-5. *New Column Encryption Key dialog box*

In this dialog box, you have entered a name for the column encryption key and selected the appropriate column master key from the drop-down list.

The final step is to encrypt the `CreditCardNumber`, `ExpMonth`, and `ExpYear` columns. When encrypting the data, you have a choice of two methods: deterministic or randomized. This is an important decision to understand, as it may have an impact of performance as well as security.

Deterministic encryption always produces the same encrypted value for the same plain text value. This means that if deterministic encryption is used, operations—including equality joins, grouping and indexing—are possible on an encrypted column. This leaves the possibility of attacks against the encryption, however.

If you use *randomized encryption*, then different encrypted values can be generated for the same plain text values. This means that while encryption loopholes are plugged, equality joins, grouping, and indexing are not supported against the encrypted data.

You will use deterministic encryption because you expect the columns to have a high cardinality. You will again use SSMS for this action because T-SQL only has support for encrypting data in new columns, not existing columns. The process of encrypting the data includes changing the column collation to BIN2, because this is the only collation currently supported by Always Encrypted.

■ **Caution** Data should be encrypted during a maintenance window. DML statements against the table while encryption is in progress could potentially result in data loss.

To invoke the Always Encrypted wizard for the `CardNumber` column, drill though Databases ➤ AdventureWorks2016 ➤ Tables in Object Explorer. Select Encrypt Columns from the context menu of the `Sales.CreditCard` table. After passing through the welcome page of the wizard, the Column Selection page is displayed, as illustrated in Figure 5-6.

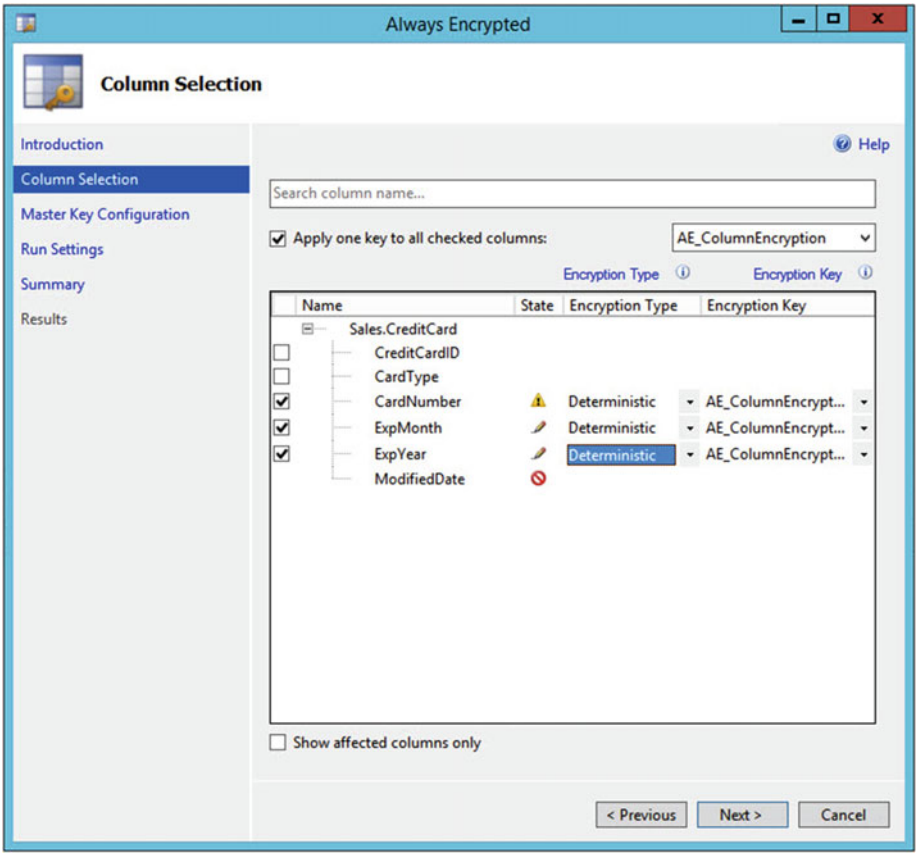


Figure 5-6. *Column selection page*

On this page, you first use the check boxes on the left-hand side to select the columns that you want to encrypt. You then have a choice of selecting an encryption key for each column individually, or using the check box and drop-down list at the top of the page to choose a single key that is used to encrypt all selected columns. You have used the latter option. Finally, you need to specify if each column should be encrypted using deterministic or randomized encryption.

The warning next to the `CardNumber` column is informing you that the column's collation is changed to the supported BIN2 collation. It is not possible to select the `DateModified` column because there is a default constraint on the column. This is not supported by Always Encrypted.

The Master Key Configuration page simply informs you that no further configuration is required. If you had chosen to create new column encryption keys on the Column Selection page, then you could use this page to associate the new keys with a column master key.

The Run Settings page provides an option for performing the encryption immediately or scripting the action out to PowerShell. The Summary page provides an overview of the actions to be performed. After clicking the Finish button on the Summary page, the encryption is performed. The Results page should be reviewed for success status.

Always Encrypted Limitations

Not all features are supported by Always Encrypted. You should check the columns for compatibility before planning your encryption strategy. The following data types are not supported:

- XML
- TIMESTAMP
- ROWVERSION
- IMAGE
- NTEXT
- TEXT
- SQL_VARIANT
- HIERARCHYID
- GEOGRAPHY
- GEOMETRY

The following features are also not fully supported:

- User defined-types are not supported
- FILESTREAM columns are not supported
- Columns with the ROWGUIDCOL property specified are not supported
- String columns are only supported when they use a BIN2 collation
- Clustered and non-clustered and full-text index key columns are only supported for deterministic encryption

- Columns referenced by computed columns are only supported when the expression does not perform unsupported operations
- Sparse column sets are not supported
- Columns that are referenced by statistics are not supported
- Columns using alias data types are not supported
- Partitioning key columns are not supported
- Columns with default constraints are not supported
- Columns referenced by unique constraints are only supported for deterministic encryption
- Primary key columns are only supported when both of the following are true:
    - Deterministic encryption is used
    - Change tracking is not implemented on the column
- Referencing columns in foreign keys are only supported when both of the following are true:
    - Deterministic encryption is used
    - The referenced and referencing columns are encrypted using the same key
- Columns referenced by check constraints are not supported
- Columns in tables that use change data capture are not supported
- Columns that are masked using Dynamic Data Masking are not supported
- Columns in existing Stretch Database tables cannot be encrypted. However, tables can be enabled for stretch, after their columns are encrypted with Always Encrypted.
- Columns in external PolyBase tables are not supported
- Columns in table variables

Summary

SQL Server 2016 provides an array of encryption options that DBAs can use to provide defense-in-depth. The encryption technologies make use of the encryption hierarchy. This starts with the *service master key*, which is encrypted by the Windows DPAPI and then used to encrypt service master keys. The service master keys are then used to encrypt keys and certificates.

Data can be encrypted at rest by using a passphrase, a combination of keys, and certificates within the encryption hierarchy. Keys and certificates stored in external key vaults are also supported through EKM integration. This method of encryption allows specific columns to be encrypted, but can cause significant bloat as well as performance implications. Applications and ETL processes need to be modified to access the encrypted data.

Transparent Data Encryption provides a low overhead method of encrypting data at rest. As the name suggests, TDE is transparent to applications, so no changes are required to applications or ETL processes in order to access the encrypted data. TDE protects your organization against the theft of a database or a backup file; however, any user with privileges to access data is able to decrypt the data.

Always Encrypted is a new technology introduced in SQL Server 2016. It allows the separation of roles and responsibilities. It is the first SQL Server encryption technology that prevents data from being accessed by highly privileged users, such as DBAs. Always Encrypted provides encryption for data, both at rest and in transit, as the data is decrypted by the client driver. Because SQL Server does not store the plain-text version of the encryption keys, even privileged users cannot decrypt the data. This is especially useful when outsourcing platform support to third parties.

CHAPTER 6

Security Metadata

Although a large amount of security metadata can be viewed from SQL Server Management Studio, there is some metadata that can only be viewed using T-SQL. Even for metadata that can be viewed through the GUI, such as the roles that a database user belongs to, there are times that it is best to use T-SQL; for example, if you need to script an action that you perform on a regular basis, or if you need to review metadata for many principals.

A complete guide to security metadata within SQL Server is worthy of a book in its own right. Therefore, this chapter explains some of the most useful and interesting metadata objects and provides insights into how you may use them.

Security Principal Metadata

When you are implementing, reviewing, or auditing a security policy on an instance, it is likely that you need to retrieve information about many security principals or securable objects. As an example of this, part of your security policy might state that all databases must be owned by the `sa` account, and you need to verify that this is the case. You could, of course, enter the context menu of each database, select Properties, and then review the Owner field on the General page of the Database Properties dialog box. If the instance hosts 200 databases, however, then this may be a rather tedious and time-consuming task.

Instead of using the GUI, it makes sense to use SQL Server metadata. The database owner of each database can be returned by using the `sp_MShasdbaccess` stored procedure or by querying the `sys.databases` catalog view.

The `sp_MShasdbaccess` stored procedure does not accept parameters; it returns the name and owner of each database, as well as the status of each database.

■ **Tip** The `sp_MShasdbaccess` procedure only returns rows for databases that the caller has access to. Providing that the procedure is run by a database administrator, this should not be an issue.

P. A. Carter, *Securing SQL Server*, DOI 10.1007/978-1-4842-2265-2_6

To retrieve the data from sys.databases, you need to return the SID (security identifier) of each database by retrieving the owner_sid column and passing this column to the SUSER_SNAME() system function. This is demonstrated in Listing 6-1.

Listing 6-1. Retrieve Database Owners from sys.databases

```
SELECT name
      ,SUSER_SNAME(owner_sid)
FROM sys.databases ;
```

The SUSER_SNAME() function accepts a SID as a parameter and returns the login name associated with the SID. If no parameter is passed to the function, then it returns the login name of the caller.

■ **Tip** Many people get confused about the difference between the SUSER_SNAME() function and a very similar function called SUSER_NAME(). They both return a login name. The difference is that SUSER_SNAME() accepts a SID as a parameter. SUSER_NAME() accepts a login id (principal id) as a parameter.

Finding a User's Effective Permissions

When you have a complex hierarchy of server roles and database roles, as well as permissions granted directly to users, it can sometimes be challenging to work out exactly which permissions a user has. A system function that can help with this issue is sys.fn_my_permissions(). This function accepts the parameters described in Table 6-1.

Table 6-1. *sys.fn_my_permissions Parameters*

Parameter	Description
securable	The name of the securable that you wish to determine a user's permissions against.
securable_class	The type of securable that is interrogated; for example, SERVER, DATABASE, or OBJECT.

The function returns the columns described in Table 6-2.

Table 6-2. *sys.fn_my_permissions*

Column	Description
entity_name	The name of the securable.
subentity_name	If the securable has columns, then subentity_name contains the name of the column; otherwise, it is NULL.
permission_name	The name of the permission assigned to the security principal.

The function is designed to return information about the caller of the function, but you can change this behavior by using the EXECUTE AS statement. The EXECUTE AS statement is used to specify the name of a login or user whose identity should be used as the execution context of the session.

As an example of how the EXECUTE AS statement works, please review Listing 6-2.

Listing 6-2. EXECUTE AS Example

```
USE master
GO

CREATE LOGIN DemoLogin WITH PASSWORD=N'Pa$$w0rd', CHECK_EXPIRATION=OFF,
CHECK_POLICY=OFF

ALTER SERVER ROLE sysadmin ADD MEMBER DemoLogin

USE AdventureWorks2016CTP3
GO

SELECT SUSER_SNAME() ;

EXECUTE AS LOGIN = 'DemoLogin' ;

SELECT SUSER_SNAME() ;

REVERT ;

SELECT SUSER_SNAME() ;
```

The script returns three results. The first result is your own login name. The second is DemoLogin's login name. And the third result is your own login again. This is because the REVERT keyword is used to change the session context back to your own security context.

The script in Listing 6-3 demonstrates how the sys.fn_my_permissions function can be used in conjunction with the EXECUTE AS clause to find a user's effective permissions at the instance, the database, and the object (within the current database) levels in a

single query. I first wrote about this technique back in 2011 on the "SQL Server: Down & Dirty" blog (www.sqlserverdownanddirty.blogspot.com); since then, the method has been used and replicated by many others.

■ **Caution** If a database uses a different collation to the server, then you may need to use the COLLATE statement within the query to avoid issues with running the script.

Listing 6-3. Find a User's Effective Permissions

```
EXECUTE AS LOGIN = 'DemoLogin'
   SELECT o.name
        , a.entity_name
        , a.subentity_name
        , a.permission_name
   FROM sys.objects o
   CROSS APPLY sys.fn_my_permissions(CONCAT(
                                        QUOTENAME(
                                                SCHEMA_NAME(schema_id))
                                      , '.'
                                      , QUOTENAME(o.name))
                                      , 'OBJECT') a
   UNION ALL
   SELECT d.name
        , a.entity_name
        , a.subentity_name
        , a.permission_name
   FROM sys.databases d
   CROSS APPLY fn_my_permissions(QUOTENAME(d.name), 'DATABASE') a
   UNION ALL
   SELECT @@SERVERNAME COLLATE Latin1_General_CI_AS
          , a.entity_name
          , a.subentity_name
          , a.permission_name
   FROM fn_my_permissions(NULL, 'SERVER') a
   ORDER BY 1
REVERT
```

The script works by running three separate queries and creating a union of the results. The first query returns each object name from sys.objects and passes this name, along with the schema name, into the sys.fn_my_permissions() function. The second query does the same thing, but instead of interrogating sys.objects, the script interrogates sys.databases to retrieve permissions at the database level. The final query resolves the user's effective permissions against the instance itself.

Securable Metadata

There are ways in which your security profile may determine that your objects need to be secured. The following sections explore some of these potential requirements and demonstrate how metadata can help you verify or enforce your policy.

Code Signing

Code injection attacks can cause security breaches. You can protect against them by using code signing. For now, however, let's simply assume that your security policy states that all assemblies and stored procedures must be code signed to help minimize the security footprint.

The script in Listing 6-4 reports which stored procedures in the database have been code signed and if the signature is valid. The script uses two security metadata objects. The first is `sys.Certificates`. The columns returned by this catalog view are described in Table 6-3.

Table 6-3. *sys.Certificates Columns*

Column	Description
`name`	The name of the certificate.
`certificate_id`	The id of the certificate.
`principal_id`	The id of the database user that owns the certificate.
`pvt_key_encryption_type`	The encryption method of the private key. Possible values are as follows: • NA: Indicates that there is no private key associated with the certificate. • MK: Indicates that encryption is by the database master key. • PW: Indicates that encryption is by password. • SK: Indicates that encryption is by the service master key.
`pvt_key_encryption_type_desc`	The textual description of the private key encryption type. Possible values are as follows: • NO_PRIVATE_KEY • ENCRYPTED_BY_MASTER_KEY • ENCRYPTED_BY_PASSWORD • ENCRYPTED_BY_SERVICE_MASTER_KEY

(*continued*)

Table 6-3. (*continued*)

Column	Description
`is_active_for_begin_dialog`	Specifies if the certificate is allowed to be used to begin an encrypted Service Broker conversation. • 0: Indicates that it is not allowed to start an encrypted Service Broker conversation. • 1: Indicates that it is allowed to start an encrypted Service Broker conversation.
`issuer_name`	The name of the authority that issued the certificate.
`cert_serial_number`	The serial number of the certificate.
`sid`	The login SID of the certificate.
`string_sid`	The name of the login SID.
`subject`	The subject associated with the certificate.
`expiry_date`	The certificate's expiry date.
`start_date`	The certificate's start date.
`thumbprint`	The SHA-1 hash of the certificate.
`attested_by`	Internal use.
`pvt_key_last_backup_date`	The date and time that the certificate was last backed up.

The second metadata object used by the script is `sys.fn_check_object_signatures()`. This system function is used to return information regarding object signatures and their validity, based on the thumbprint of a certificate or asymmetric key. The function accepts the parameters described in Table 6-4.

Table 6-4. *sys.fn_check_object_signatures Parameters*

Parameter	Description
`@Class`	The type of thumbprint that the function checks. The acceptable values are • Certificate • Asymmetric key
`@Thumbprint`	The thumbprint to be checked.

The `sys.fn_check_object_signatures` function returns the columns described in Table 6-5.

Table 6-5. *sys.fn_check_object_signatures Columns*

Column	Description
type	The type description of the entity.
entity_id	The object id of the evaluated entity.
is_signed	Denotes if the object is signed or not. • 0: Indicates that the object is not signed. • 1: Indicates that the object is signed.
is_signature_valid	Denotes if the object's signature is valid. If the object is not signed, it returns 0. • 0: Indicates that either the object is not signed or that the signature is not valid. • 1: Indicates that the object's signature is valid.

Listing 6-4. Check Objects' Signatures

```
DECLARE @thumbprint VARBINARY(20) ;

SET @thumbprint =
(
SELECT thumbprint
FROM sys.certificates
WHERE name LIKE '%SchemaSigningCertificate%'
) ;

SELECT entity_id
         , SCHEMA_NAME(o.schema_id) + '.' + OBJECT_NAME(entity_id) AS
           ProcedureName
         , is_signed
         , is_signature_valid
FROM sys.fn_check_object_signatures ('certificate', @thumbprint)  cos
INNER JOIN sys.objects o
        ON cos.entity_id = o.object_id
WHERE cos.type = 'SQL_STORED_PROCEDURE' ;
GO
```

The first part of the script retrieves the thumbprint of the database's code signing certificate from the `sys.Certificates` catalog view. The second part of the script passes this thumbprint into the `sys.fn_check_object_signatures` and joins the results to the `sys.objects` catalog view to retrieve the schema name of the procedure.

Permissions Against a Specific Table

You may have a specific table or set of tables that contains sensitive information, and your security policy may state that you need to regularly audit who has permissions to that table and who assigned those permissions. Using SQL Server metadata, this is a straightforward task.

The `sp_table_privileges` system stored procedure identifies all permissions that principals have against a specific table, along with who granted those permissions. The procedure accepts the parameters described in Table 6-6.

Table 6-6. *sp_table_privileges Parameters*

Parameter	Description
`@table_name`	The name of the table to report on.
`@table_owner`	The name of the schema to which the table belongs.
`@table_qualifier`	The name of the database that hosts the table.
`@fUsePattern`	Specifies if _, %, [, and] should be treated as wildcard characters. • 0 indicates that they should be treated as literals. • 1 indicates that they should be treated as wildcard characters.

The columns returned by the `sp_table_privileges` procedure are described in Table 6-7.

Table 6-7. *sp_table_privileges Columns*

Column	Description
`TABLE_QUALIFIER`	The database in which the table resides.
`TABLE_OWNER`	The schema in which the table resides.
`TABLE_NAME`	The name of the table.
`GRANTOR`	The security principal that granted the permission.
`GRANTEE`	The security principal assigned the permission.
`PRIVILEGE`	The permission that has been assigned.
`IS_GRANTABLE`	Specifies if the grantee has the `WITH GRANT`* assignment.

**Please see Chapter 2 for further details*

The statement in Listing 6-5 returns results for all tables in the current database.

Listing 6-5. sp_table_privileges

```
EXEC sp_table_privileges @Table_name = '%' ;
```

Audit Metadata

As discussed in Chapter 3, SQL Server Audit provides a granular and lightweight method of auditing users' actions within SQL Server. One of the advantages of SQL Server Audit is that you are able to "audit the audit" in an attempt to avoid non-reputability. For example, if an ill-intending DBA turned off the auditing, while they performed a malicious act, the action itself would not be audited, but the fact that the DBA had turned the audit off and then turned it back on again would be audited.

SQL Server exposes many metadata objects that assist a DBA in his work. One of the objects that I find most useful is the `sys.fn_get_audit_file()` function. The function returns the contents of a SQL Server Audit file. This can be inserted into a table for further analysis. The function accepts three parameters, which are described in Table 6-8.

Table 6-8. *sys.fn_get_audit_file() Parameters*

Parameter	Description
`file_pattern`	The name of the audit file that you wish to read. This path can contain the * wildcard to read multiple files. This is useful when you have rollover files.
`initial_file_name`	Specifies the path and the name of a specific file in the audit file set where the file read should begin. If not required, pass `NULL`.
`audit_record_offset`	Specifies a known location with the file specified for the `initial_file_name` parameter and begins the file read at this record. If not required, pass `NULL`.

The `sys.fn_get_audit_file()` function returns the columns described in Table 6-9.

Table 6-9. *sys.fn_get_audit_file() Columns*

Column	Description
`event_time`	The date and time at which the audited event occurred.
`sequence_number`	A sequence number of records within a single audit entry where the entry was too large to fit inside a buffer and was broken down.
`action_id`	The id of the action.
`succeeded`	Specifies if the action that caused the audit event to fire was successful. • 0: Indicates that the action failed. • 1: Indicates that the action succeeded.

(*continued*)

Table 6-9. (*continued*)

Column	Description
permission_bitmask	Where appropriate, specifies the permissions that were assigned or revoked.
is_column_permission	Specifies if the permission (in the permission_bitmask column) was a column-level permission. • 0: Indicates that it was not a column-level permission. • 1: Indicates that it was a column-level permission.
session_id	The id of the session in which the event occurred.
server_principal_id	The principal id of the login that performed the action, which caused the audit event to fire.
database_principal_id	The principal id of the database user that performed the action, which caused the audit event to fire.
target_server_principal_id	Where applicable, returns the principal id of the login that was subject to a permission assignment or revocation.
target_database_principal_id	Where applicable, returns the principal id of the database user that was subject to a permission assignment or revocation.
object_id	Where applicable, returns the object id of the target object that caused the audit event to fire.
class_type	The type of auditable entity on which the auditable event occurred.
session_server_principal_name	The name of the login that the session was executing in. This is blank if no session was established. For example, where a failed login has been audited
server_principal_name	The name of the login that performed the action, which caused the audit event to fire.
server_principal_sid	The SID of the login that performed the action, which caused the audit event to fire.
database_principal_name	The name of the database user that performed the action, which caused the audit event to fire.
target_server_principal_name	Where applicable, returns the name of the login that was subject to a permission assignment or revocation.
target_server_principal_sid	Where applicable, returns the SID of the login that was subject to a permission assignment or revocation.

(*continued*)

Table 6-9. (*continued*)

Column	Description
target_database_principal_name	Where applicable, returns the name of the database user that was subject to a permission assignment or revocation.
server_instance_name	The server\instance name of the instance where the audit event occurred
database_name	The name of the database in which the audit event occurred.
schema_name	The schema context in which the audit event occurred.
object_name	The name of the object, which was the subject of the auditable event.
statement	The T-SQL statement that caused the audit event to fire.
additional_information	For some events, an XML document is returned, containing additional information. For example, if a failed login is audited, the additional information includes the IP address that the login attempt originated from.
file_name	The fully qualified name of the audit file.
audit_file_offset	The buffer offset of the audit record, within the file.
user_defined_event_id	When an audit event has been written using sp_audit_write, returns the user defined event id.
user_defined_information	When an audit event has been written using sp_audit_write, returns user defined additional information.

The query in Listing 6-6 returns all records from all audit files stored within the c:\audit folder.

Listing 6-6. Read an Audit File

```
SELECT * FROM sys.fn_get_audit_file('c:\audit\*',NULL,NULL) ;
```

Encryption Metadata

Chapter 5 discusses encryption in SQL Server, and as you can imagine, there are a raft of metadata objects that expose information regarding your encryption configuration. The following sections discuss useful metadata that is exposed around Always Encrypted and TDE.

Always Encrypted Metadata

Because there can be a one-to-many relationship between column master keys and column encryption keys, followed by a one-to-many relationship between column encryption keys and encrypted columns, metadata can be invaluable in keeping track of how your data is encrypted. The query in Listing 6-7 joins `sys.tables` and `sys.columns` to the new `sys.column_encryption_keys`, `sys.column_encryption_key_values`, and `sys.column_master_keys` catalog views to provide a complete path through the hierarchy from the column to key store location of the column master key.

The `sys.column_encryption_keys` view returns the columns described in Table 6-10.

Table 6-10. *sys.column_encryption_keys Columns*

Column	Description
`name`	The name of the column encryption key.
`column_encryption_key_id`	The id of the column encryption key.
`create_date`	The date and time that the key was created.
`modify_date`	The date and time that the key was last modified.

The `sys.column_encryption_key_values` view returns the columns described in Table 6-11.

Table 6-11. *sys.column_encryption_key_values Columns*

Column	Description
`column_encryption_key_id`	The id of the column encryption key.
`column_master_key_id`	The id of the column master key that has been used to encrypt the column encryption key.
`encrypted_value`	The value of the column encryption key, encrypted using the column master key.
`encryption_algorithm_name`	The algorithm used to encrypt the column encryption key.

** Use this view as an intermediate join between sys.column_encryption_keys and sys.column_master_keys*

The `sys.column_master_keys` view returns the columns described in Table 6-12.

Table 6-12. *sys.column_master_keys*

Column	Description
name	The name of the column master key.
column_master_key_id	The id of the column master key.
create_date	The date and time that the key was created.
modify_date	The date and time that the key was last modified.
key_store_provider_name	The type of key store that the column master key is stored in.
key_path	The path to the key within the key store.

Listing 6-7 demonstrates how these metadata objects can be drawn together.

Listing 6-7. Interrogate Always Encrypted Metadata

```
SELECT
          t.name AS TableName
        , c.name AS ColumnName
        , c.encryption_type_desc
        , c.encryption_algorithm_name
        , cek.name AS ColumnEncryptionKeyName
        , cev.encrypted_value
        , cev.encryption_algorithm_name
        , cmk.name as ColumnMasterKeyName
        , cmk.key_store_provider_name AS column_master_key_store_provider_
          name
        , cmk.key_path
FROM sys.columns c
INNER JOIN sys.column_encryption_keys cek
    ON c.column_encryption_key_id = cek.column_encryption_key_id
INNER JOIN sys.tables t
    ON c.object_id = t.object_id
JOIN sys.column_encryption_key_values cev
    ON cek.column_encryption_key_id = cev.column_encryption_key_id
JOIN sys.column_master_keys cmk
    ON cev.column_master_key_id = cmk.column_master_key_id ;
```

TDE Metadata

■ **Note** For described information regarding TDE, please refer to Chapter 5.

TDE metadata is exposed through the `sys.databases`, `sys.certificates`, and `sys.database_encryption_keys` catalog views. The `sys.databases` catalog view contains a column called `is_encrypted`. This column has the data type of `BIT` and returns 0 if a database is not encrypted with TDE and 1 if it is encrypted. Information on the certificate used to encrypt the database encryption key is exposed through `sys.certificates`.

The `sys.database_encryption_keys` catalog view exposes details of the keys used to encrypt the databases. It returns one row for each database that has a database encryption key associated with it. Table 6-13 details the columns returned by this catalog view.

Table 6-13. *sys.database_encryption_keys Columns*

Column	Description
`database_id`	The id of the database that is encrypted using the key.
`encryption_state`	Specifies the current state of encryption for the database indicated by the `database_id` column. The following are possible values: • 0: Indicates that no encryption key is present. You will not see this status under normal operations, because if no key exists, the catalog view does not return a row. • 1: Indicates that the database is not encrypted. You see this status when TDE has been encrypted, but the database encryption key has not been dropped. • 2: Indicates that the database is currently being encrypted. You see this status immediately after enabling TDE on a database while the background encryption thread is still running. • 3: Indicates that the database is encrypted. • 4: Indicates that a change to the database encryption key is currently in progress. • 5: Indicates that the database is currently being decrypted. You see this status immediately after turning off TDE for a database, before the background thread completes. • 6- Indicates that a change to the database encryption key or the server certificate used to encrypt the database encryption key is currently in progress.
`create_date`	The date and time that the database encryption key was created.
`regenerate_date`	The date and time that the database encryption key was regenerated.
`modify_date`	The date and time that the database encryption key was last modified.
`set_date`	The date and time that the database encryption key was associated with the database.

(*continued*)

Table 6-13. (*continued*)

Column	Description
opened_date	The date and time that the database encryption key was last opened.
key_algorithm	The algorithm used to encrypt the database encryption key.
key_length	The length of the key.
encryptor_thumbprint	The encrypted value of the certificate used to encrypt the database encryption key.
encryptor_type	Indicates the type of encryptor that was used to encrypt the database encryption key. Possible values are • ASYMMETRIC KEY • CERTIFICATE
percent_complete	If the encryption_state column indicates a status of 2 or 5, this column indicates how far through the encryption or decryption process the background thread is. If the encryption_state column indicates a different status, then this column returns 0.

The metadata exposed for TDE can be useful at various times; for example, to return a list of encrypted database on the instance, use the script in Listing 6-8.

Listing 6-8. Return a List of Encrypted Databases

```
SELECT name
FROM sys.databases
WHERE is_encrypted = 1 ;
```

If you need to ensure that all of the server certificates that are used to encrypt database encryption keys have been backed up, you can use the query in Listing 6-9. This query returns a list of certificates used in TDE that have not been backed up.

Listing 6-9. Ensure that Certificates Have Been Backed Up

```
SELECT
     DB_NAME(dek.database_id) AS DatabaseName
    ,c.name AS CertificateName
FROM AdventureWorks2014.sys.dm_database_encryption_keys dek
INNER JOIN master.sys.certificates c
ON c.thumbprint = dek.encryptor_thumbprint
WHERE c.pvt_key_last_backup_date IS NULL ;
```

If you had a task of encrypting many databases on an instance, you could even use metadata to create a metadata-driven script that would do the hard work for you. I used this script recently—when a friend of mine mentioned that he had been quoted three months by his DBA team—to encrypt 400+ databases on an instance.

The script in Listing 6-10 firsts creates a server certificate that is used to encrypt the database encryption key for each database. The script then uses the `sp_msforeachdb` system stored procedure to loop around each database.

Inside the loop, the script first checks to ensure that it is not in the context of a system database, and then checks to ensure that the database has not already been encrypted. This makes the script re-runnable, should you have an issue part way through. After the checks are complete, it creates a database encryption key before enabling TDE.

Listing 6-10. Metadata Driven Encryption Script

```
USE master
GO

CREATE CERTIFICATE TDECert WITH SUBJECT = 'My DEK Certificate';
GO

EXEC sys.sp_MSforeachdb @command1 = 'USE ?
IF (SELECT DB_ID()) > 4
BEGIN
    IF (SELECT is_encrypted FROM sys.databases WHERE database_id = DB_ID())
= 0
        BEGIN
        CREATE DATABASE ENCRYPTION KEY
        WITH ALGORITHM = AES_128
        ENCRYPTION BY SERVER CERTIFICATE TDECert

        ALTER DATABASE ?
        SET ENCRYPTION ON
        END
END' ;
```

Securing Metadata

While metadata can prove incredibly useful—not just from the security perspective, but also in every other area of SQL Server administration, it can also prove to be a security hole in its own right. If metadata were accessible to everybody, then an attacker could use it to gain information, regarding the configuration of your instance.

Therefore, most metadata only becomes visible to a user after they have been granted permissions to use the object in some way. For example, if you grant the user Phil the `SELECT` permission on `dbo.MyTable`, Phil automatically sees the row within `sys.tables` and `sys.objects` that relates to the `dbo.MyTable` object.

If a user needs to see metadata about an object that they should not have permissions to use in any other way, then the `VIEW DEFINITION` permission can be granted upon that object. The `VIEW DEFINITION` permission can also be granted at the scope of a database or an entire instance. At the instance level, the permission `VIEW ANY DEFIITION` gives complete access to metadata instance-wide. This can be useful when you are creating metadata-driven automated scripts and you wish to apply the principal of least privilege.

There are some metadata objects in which users cannot be automatically granted VIEW DEFINITION permission when other permissions are assigned to the object. This is because the objects sit outside of the permissions structure. Take partitions, for example. Each table can be split across three partitions: one for in-row data, another for LOB (large object) data, and the third for overflow data. There is no way to assign permissions on partitions, because they are not directly accessible.

In these circumstances, the public role has the ability to view the associated metadata, and the VIEW DEFINITION permission does not apply. The metadata objects that are visible to the public role are as follows:

```
sys.partition_functions
sys.partition_range_values
sys.partition_schemes
sys.data_spaces
sys.filegroups
sys.destination_data_spaces
sys.database_files
sys.allocation_units
sys.partitions
sys.messages
sys.schemas
sys.configurations
sys.sql_dependencies
sys.type_assembly_usages
sys.parameter_type_usages
sys.column_type_usages
```

Risks of Metadata Visibility

Even with the security measures that are in place to protect SQL Server, an attacker may still be able to expose some metadata if the overall security design of your application is weak. For example, imagine that you have a web application that handles security in the application tier and then connects to a SQL Server instance by using a single, highly privileged account.

If the web application is vulnerable to SQL injection, then an attacker could force the execution of the query in Listing 6-11.

Listing 6-11. Forced Information Disclosure

```
SELECT 1 + name FROM sys.tables
```

When run against the AdventureWorks2016 database, the query in Listing 6-11 returns the error message shown in Figure 6-1.

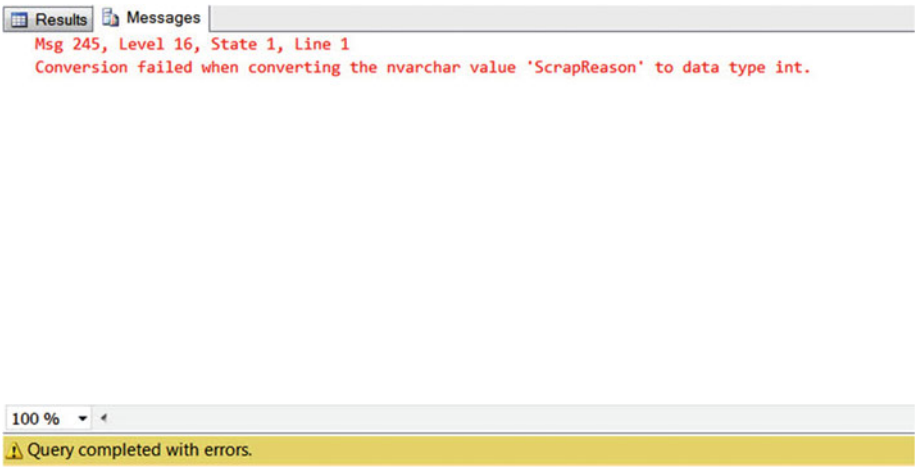


Figure 6-1. *Forced error message*

This error message has provided the attacker with the following information:

- There is a table in the database s `ScrapReason`.
- The application is leaking metadata.
- The application is (probably) running through a highly privileged account.

This information gives an attacker plenty of insight into where to start an attack. For example, if the attacker is correct and the application does run through a single account, then it is likely that there is a user's table specifying permissions. The attacker could amend his query to filter by tables that contain the wildcard strings `%user%` or `%login%`. Once the attacker has this information, he can attack the table specifically and start spoofing user identities!

■ **Tip** The moral of the story is that you should always evaluate the security profile of an application holistically to minimize the risk of attack. Even if your instance is secure, a poorly designed application tier could leave you vulnerable.

Summary

SQL Server exposes a vast amount of metadata. This includes metadata that relates to the security implementation within your instance. This security-related metadata assists a DBA in ensuring that the security policy is met. For example, metadata checks a user's effective permissions at every level of the hierarchy, checks for modules that have not been code signed, and checks which principals have what permissions to a specific securable.

Useful metadata is also exposed about encryption artifacts. This metadata can be used for a variety of purposes, from auditing the column master key locations of an Always Encrypted implementation to automating a TDE implementation.

Although metadata brings many advantages, it also brings risks. If metadata is exposed, then it launches an attack against SQL Server. This is an even higher risk when an application uses a single, highly privileged user to connect to the database engine.

CHAPTER 7

Implementing Service Accounts for Security

A *service account* is simply an account that is used as the security context that Windows uses to run a service. Each SQL Server service needs to be configured with a service account, which is granted the appropriate permissions to run the service. This chapter discusses the general considerations for service accounts. You then examine how service accounts can be exploited. Finally, I discuss the appropriate service account strategy.

Service Account Types

SQL Server 2016 supports the following types of account as service accounts:

- Local user
- Domain user
- Built-in accounts
- Managed service accounts (MSAs)
- Virtual accounts

The *local user* account type refers to a Windows user that has been created on the local server. The *domain user* account type refers to an AD (Active Directory) user that has been created at the domain level. *Built-in accounts* refer to the `NETWORK SERVICE`, `LOCAL SERVICE`, and `LOCAL SYSTEM` accounts that are always available in Windows environments. The concepts of MSAs and *virtual accounts* are relatively new, however, so let's spend some time looking at these in more detail.

Virtual Accounts

Virtual accounts were introduced in Windows Server 2008R2 and Windows 7. They are created locally on the server, but are able to access domain resources by using the computer account's credentials.

P. A. Carter, *Securing SQL Server*, DOI 10.1007/978-1-4842-2265-2_7

Virtual accounts were introduced to improve isolation between services in environments where administrators would typically run services such as SQL Server, Exchange, or IIS under the context of the `LOCAL SERVICE` account. In addition to isolation, virtual accounts also offer the benefit of being "managed." This means that administrators do not need to register an SPN (service principal name) and do not need to manage the password for the account. This is taken care of automatically by the Windows environment.

By the nature of virtual accounts being local, they cannot be used on multiple servers. This means that they are inappropriate for implementations such as AlwaysOn failover clusters. If you are performing a stand-alone installation of a SQL Server instance on Windows Server 2008R2 or higher, or on Windows 7 or higher, then the accounts for the following services defaults to a virtual account:

- Database Engine
- SQL Server Agent
- SQL Server Analysis Services (SSAS)
- SQL Server Integration Services (SSIS)
- SQL Server Reporting Services (SSRS)
- SQL Server Distributed Relay Controller
- SQL Server Distributed Relay Client
- FD Launcher (Full-Text Daemon Launcher service)

Even if you are installing a clustered instance of SQL Server, the following services default to a virtual account. This is because the services are not cluster-aware and are typically installed as stand-alone on each cluster node.

- SSIS
- SSRS
- FD Launcher

Managed Service Accounts

MSAs are similar to virtual accounts in the respect that they are automatically managed and there is no need for administrators to manually configure an SPN or manage passwords. The difference between a virtual account and an MSA is that MSAs are domain-level accounts, rather than local.

Because MSAs are domain-level accounts, they should be used in preference over virtual accounts when there is a need for the service to interact with network level resources, such as file shares and so forth. Despite being created at the domain level, an MSA is still only assigned to a single machine on the network. This means that MSAs cannot be used for implementations such as failover cluster instances, because they can only run a service on a single machine.

To overcome this limitation, gMSAs (group managed service accounts) were introduced in Windows Server 2012 R2. gMSAs are the same as MSAs, except that they have the ability to run services on multiple servers. This means that they can be used for cluster implementations.

Naturally, as gMSAs were only introduced in Windows Server 2012 R2, your server must have this version of Windows Server or higher and your domain controller must be running at the Windows Server 2012 R2 functional level. At the time of writing, most organizations are not at a place where they are ready to move to a Windows 2012 R2 functional level, but if you do have this luxury, then gMSAs are a great feature.

■ **Tip** If you plan to use an MSA or a gMSA, then the account must be created on the domain controller prior to the installation of SQL Server.

SQL Server Services

When planning an installation of SQL Server, you should consider the service account requirements for each SQL Server service, and as with all other security configuration, you should always attempt to apply the principal of least privilege.

■ **Tip** The principal of least privilege states that each security principal (in this case, the service account) should only be given the minimum set of permissions required to carry out its day-to-day activities. If higher permissions are required for a one-off task, then permissions should be elevated when required and reduced after the activity has completed.

Table 7-1 describes the minimum set of permissions and assignments that are required for each service account to perform its basic functions. Of course, if the service needs to interact with other resources, such as file shares, then these permissions should be appended to the list to meet your specific requirements. The permissions and assignments described here are granted to the appropriate service accounts during the instance installation process.

Table 7-1. *Service Account Permission and Assignment Requirements*

Service	User Rights Assignments	Permissions

SQL Server Database Engine	• Log on as a service
	• Replace a process-level token
	• Bypass traverse checking
	• Adjust memory quotas for a process
		• Start SQL Writer
		• Read the Event Log service
		• Read the Remote Procedure Call service
		• Instid\MSSQL\backup
		  • Full control
		• Instid\MSSQL\binn
		  • Read
		  • Execute
		• Instid\MSSQL\data
		  • Full Control
		• Instid\MSSQL\FTData
		  • Full Control
		• Instid\MSSQL\Install
		  • Read
		  • Execute
		• Instid\MSSQL\Log
		  • Full Control
		• Instid\MSSQL\Repldata
		  • Full Control
		• 130\shared
		  • Read
		  • Execute

SQL Server Agent	• Log on as a service
	• Replace a process-level token
	• Bypass traverse checking
	• Adjust memory quotas for a process
		• Instid\MSSQL\binn
		  • Full Control
		• Instid\MSSQL\binn
		  • Full Control
		• Instid\MSSQL\Log
		  • Read
		  • Write
		  • Delete
		  • Execute
		• 130\com
		  • Read

(*continued*)

Table 7-1. (*continued*)

Service	User Rights Assignments	Permissions
		• Execute • 130\shared • Read • Execute • 130\shared\Errordumps • Read • Write • ServerName\EventLog • Full Control
SQL Server Analysis Services	• Increase a process working set • Adjust memory quotas for a process • Lock pages in memory (If paging is disabled) • Increase scheduling priority (when installed as a failover clustered instance) • Log on as a service (When installed in Tabular mode)	• 130\shared\ASConfig • Full Control • Instid\OLAP • Read • Execute • Instid\Olap\Data • Full Control • Instid\Olap\Log • Read • Write • Instid\OLAP\Backup • Read • Write • Instid\OLAP\Temp • Read • Write • 130\shared\Errordumps • Read • Write
SQL Server Reporting Services	• Log on as a service	• Instid\Reporting Services\Log Files • Read • Write • Delete • Instid\Reporting Services\ReportServer • Read

(*continued*)

Table 7-1. (*continued*)

Service	User Rights Assignments	Permissions
		• Execute • Instid\Reportingservices\Reportserver\global.asax • Full Control • Instid\Reportingservices\Reportserver\Reportserver.config • Read • Instid\Reporting Services\reportManager • Read • Execute • Instid\Reporting Services\RSTempfiles • Read • Write • Execute • Delete • 130\Shared • Read • Execute • 130\shared\Errordumps • Read • Write
SQL Server Integration Services	• Log on as a service • Bypass traverse checking • Impersonate a client after authentication	• Write to application event log • 130\dts\binn\MsDtsSrvr.ini.xml • Read • 130\dts\binn • Read • Execute • 130\shared • Read • Execute • 130\shared\Errordumps • Read • Write

(*continued*)

Table 7-1. (*continued*)

Service	User Rights Assignments	Permissions

Full-text Search	• Log on as a service
	• Adjust memory quotas for a process
	• Bypass traverse checking
		• Instid\MSSQL\FTData
		  • Full Control
		• Instid\MSSQL\FTRef
		  • Read
		  • Execute
		• 130\shared
		  • Read
		  • Execute
		• 130\shared\Errordumps
		  • Read
		  • Write
		• Instid\MSSQL\Install
		  • Read
		  • Execute
		• Instid\MSSQL\jobs
		  • Read
		  • Write

SQL Server Browser	• Log on as a service	• 130\shared\ASConfig
		  • Read
		• 130\shared
		  • Read
		  • Execute
		• 130\shared\Errordumps
		  • Read
		  • Write

SQL Server VSS Writer		No permissions are granted for this service, because it runs under the context of `LOCAL SYSTEM`, which already has all required permissions

SQL Server Distributed Replay Controller	• Log on as a service	• <ToolsDir>\DReplayController\Log\
		  • Read
		  • Execute
		  • List

(*continued*)

Table 7-1. (*continued*)

Service	User Rights Assignments	Permissions

		• <ToolsDir>\DReplayController\ DReplayController.exe
		  • Read
		  • Execute
		  • List
		• <ToolsDir>\DReplayController\ resources\
		  • Read
		  • Execute
		  • List
		• <ToolsDir>\DReplayController\
		  • Read
		  • Execute
		  • List
		• <ToolsDir>\DReplayController\ DReplayController.config
		  • Read
		  • Execute
		  • List
		• <ToolsDir>\DReplayController\ IRTemplate.tdf
		  • Read
		  • Execute
		  • List
		• <ToolsDir>\DReplayController\ IRDefinition.xml
		  • Read
		  • Execute
		  • List

SQL Server Distributed Replay Client	• Log on as a service	• <ToolsDir>\DReplayClient\Log\
		  • Read
		  • Execute
		  • List
		• <ToolsDir>\DReplayClient\ DReplayClient.exe

(*continued*)

Table 7-1. (*continued*)

Service	User Rights Assignments	Permissions
		• Read • Execute • List • <ToolsDir>\DReplayClient\resources\ • Read • Execute • List • <ToolsDir>\DReplayClient\ • Read • Execute • List • <ToolsDir>\DReplayClient\ DReplayClient.config • Read • Execute • List • <ToolsDir>\DReplayClient\IRTemplate.tdf • Read • Execute • List • <ToolsDir>\DReplayClient\IRDefinition. xml • Read • Execute • List
Launchpad	• Log on as a service • Replace a process-level token • Bypass traverse checking • Adjust memory quotas for a process	• %binn • Read • Execute • ExtensiblilityData • Full Control • Log\ExtensibiltityLog • Full Control

As you can see, even if only the minimum set of required permissions are granted to service accounts, they are still highly privileged and are a prime target for attackers.

How Service Accounts Can Become Compromised

There are many methods that an attacker could use to attempt to compromise your service accounts. For example, if an attacker already has access to an instance—either because the attack is internal or because the attacker has already compromised the instance by using SQL injection and so forth, then an SMB (Server Message Block) attack could be used to compromise the service account credentials.

Extended stored procedures, such as the undocumented `xp_dirtree` system stored procedure, can be executed with no permissions above the public role. This means that if you can access the instance, you can run it. This particular procedure is used to list the contents of a folder or a file share that you pass into it as a parameter. In order to gain access to the folder, it must first authenticate using the SQL Server Database Engine service account.

While the procedure is authenticating to the folder, the attacker can use an SMB capture from a tool such as Metasploit to capture the authentication request. This contains the hashed password, but also the nonce (pseudo-random cryptographic number), meaning that the attacker has enough information to reveal the clear text password by using specialized tools.

Designing a Pragmatic Service Account Strategy

The service account model that you choose is key to both the security and manageability of your environment. Different organizations have different requirements for service account models. You may be constrained by compliance requirements, overarching IT policies, and other factors. Essentially, the choice that you make is a trade-off between the security and operational supportability of your environment.

For example, the Microsoft best practice is to isolate services by using a separate service account for every service and to ensure that every server in your environment uses a discrete set of service accounts, since this fully enforces the principle of least privilege, as described in the “SQL Server Services” section of this chapter.

In reality, however, you find that this approach introduces significant complexity into your SQL Server estate. And it can increase the cost of operational support, while also risking longer outages in disaster scenarios. On the flip side, I have worked in organizations where the service account model is very coarse, to the point where there is only a single set of SQL Server service accounts for each region. This approach can also cause significant operational issues. Imagine, for example, that you have a large estate and the whole estate uses the same service account. Now imagine that you have a compliance requirement to change service account passwords on a 90-day basis. This means that you would cause an outage to your entire SQL Server estate at the same time. This simply is not practical.

There is no right or wrong answer to this problem. The solution depends on the requirements and constraints of individual organizations. For organizations that have the correct domain functional level, MSAs and gMSAs, along with the use of virtual accounts, often provide a workable solution.

However, even if the operating system and domain level prerequisites are met, database administration teams that rely heavily on automation to manage the maintenance routines within their enterprise may argue that virtual accounts do not meet their requirements because their automation is configured to use the service account of the local instance to authenticate and run maintenance routines.

For organizations that use domain accounts as service accounts, I tend to recommend a distinct set of service accounts for each data-tier application. So if you imagine an environment as shown in Figure 7-1, where your data-tier application consists of a two-node cluster and an ETL server in a primary site, and two DR servers in a secondary site, this design would involve a common set of service accounts used by all of these instances, but not shared with other data-tier applications. In the example shown in Figure 7-1, Data-Tier Application2 requires its own set of service accounts.

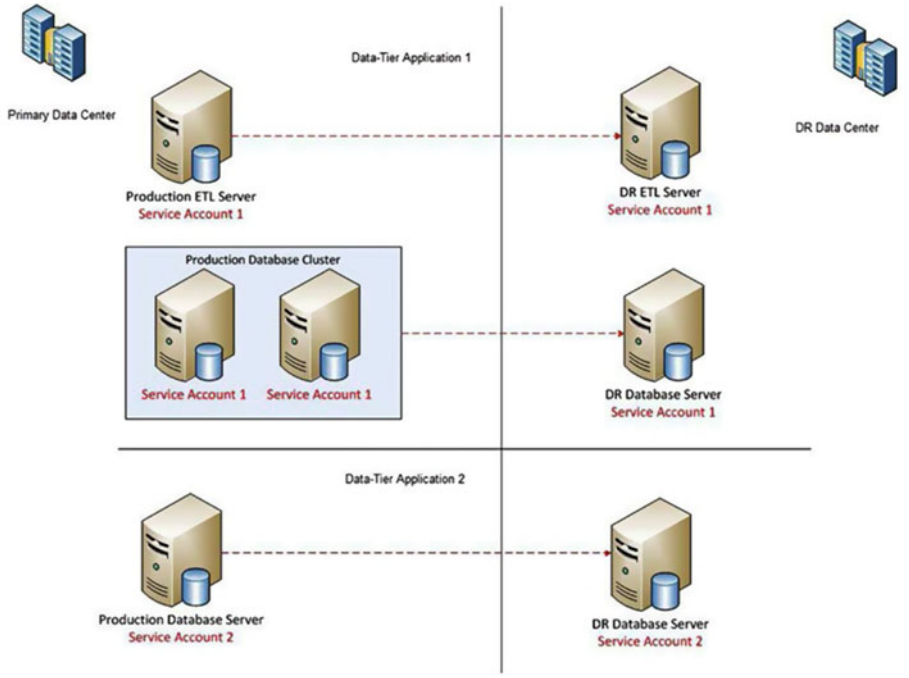


Figure 7-1. *Service account model example*

■ **Caution** In the preceding example, each data-tier application also requires a discrete set of service accounts for lower environments. Production service accounts should never be used in lower environments, such as UAT or development, due to the reduced security measures and the higher chance of compromise in these environments.

Summary

Depending on the components of SQL Server 2016 that are installed, there may be a single instance and up to 11 service accounts in use across shared features. Each of these service accounts has its own set of minimum permissions requirements, but even when the principal of least privilege is followed, the range of user rights assignments and permissions assigned to each account make them highly desirable accounts to attackers who wish to compromise your system. Attackers can use many methods to attempt to compromise your service account credentials, including SMB capture attacks, which force the database engine to disclose the credentials of its service account.

To mitigate the risks of service account exploits, it is important to follow the principal of least privilege. Complete isolation of services can lead to an environment that is very difficult to manage. This means that any service account strategy should be pragmatic and feed into the overall security strategy.

Services can run under the security context of the `LOCAL SYSTEM` account, the `LOCAL SERVICE` account, the `NETWORK SERVICE` account, a local user account, a domain user account, a virtual account, a managed service account, or a group managed service accounts. While each of these account types have their own advantages and disadvantages, virtual accounts, MSAs, and gMSAs are generally considered the best solution, provided that you have the correct domain functional level, operating system, and the ability to support within your DBA tooling.

CHAPTER 8

Protecting Credentials

Stealing the credentials of a security principal with the intent of elevating your allowed permissions is known as *identity spoofing*. There are various ways that an attacker may attempt to steal credentials. This chapter discusses some of those methods, as well as countermeasures that you can put in place to mitigate the risk.

Protecting the sa Account

Although it has long been best practice to use Windows authentication rather than mixed mode authentication, which allows authentication using both Windows authentication and SQL authentication, the majority of corporate instances (in my experience) are still configured to use mixed mode authentication.

While there seems to be many cultural reasons for DBAs using mixed mode authentication as standard, as opposed to by exception, there are also often technical reasons, despite these reasons often being a little tenuous.

For example, many applications provide security at the application tier, and then use a single login to connect to the database engine. The most cost effective way to implement this (with the shortest time to market) is to use a login that uses SQL authentication. It is possible for the application to use Windows authentication, but this would require a Windows service to be written, which would authenticate to the instance.

Also, the tooling that many DBAs have built and grown internally relies on the use of the `sa` account to run maintenance routines on local servers. The arguments against changing the approach include cost and operational risk. In this particular scenario, however, you usually find that the risk is bigger than it seems on the surface. Often, these scripts work because the `sa` account has the same password on every instance. This means that if the `sa` account is compromised on one server, the attacker has administrator-level permissions on every instance within the environment.

Within instances that use mixed mode authentication, the `sa` account is particularly susceptible. This is for two reasons: first, it is a highly privileged account and therefore very desirable to attackers, and second, because the `sa` account is very well known. Anybody with the skills and ambition to launch an attack against a SQL Server instance will of course know of the `sa` account. For these reasons, it is very important to mitigate the risk of attacks against spoofing of the `sa` account.

P. A. Carter, *Securing SQL Server*, DOI 10.1007/978-1-4842-2265-2_8

DBA Steps to Mitigate the Risks

When running in mixed mode, there are various steps that a DBA can take to protect the sa account, depending on the requirements of the application and the environment. The following sections discuss each of these steps.

Disabling the sa Account

If the sa account is not specifically required, then it should be disabled and administrator access granted to accounts that use Windows authentication. The sa account is disabled with the command in Listing 8-1.

Listing 8-1. Disable the sa Account

```
ALTER LOGIN sa DISABLE ;
GO
```

Renaming the sa Account

If there are applications or processes that specifically require an administrative account with SQL authentication, then you should consider renaming the sa account. The command in Listing 8-2 renames the sa account.

Listing 8-2. Rename the sa Account

```
ALTER LOGIN sa WITH NAME = AdminAccount ;
GO
```

Ensuring Reputability

In the worst-case scenario, you have third-party or legacy applications that have the sa account name hard-coded, and which simply cannot be modified to follow the principal of least privilege. This scenario is a lot more common than you may think.

If this is the case, then you should at least ensure that the sa account is configured to inherit the domain's password complexity and password expiration settings. This avoids the sa account's password being set to a simple password or never being changed. The command to achieve this is shown in Listing 8-3.

Listing 8-3. Force Password Polices for sa Account

```
ALTER LOGIN sa
WITH CHECK_EXPIRATION=ON
   , CHECK_POLICY=ON ;
```

It is important to remember that anybody with administrative control over an instance can change any of these settings for the `sa` account. Therefore, if you are concerned with a large enterprise, as opposed to a single instance, then you should configure auditing, so that you can trace any alterations to administrative accounts. Let's now create a server audit and an associated server audit specification that captures any changes made to the `sa` account.

■ **Tip** This SQL Server Audit specification is used in conjunction with the one discussed in Chapter 2; it captures any additions to the `sysadmin` fixed server role.

To create a server audit via SQL Server Management Studio, drill through the Security node in Object Explorer and select New Audit from the Audits context menu. This causes the General page of the Create Audit dialog box to display, as shown in Figure 8-1.

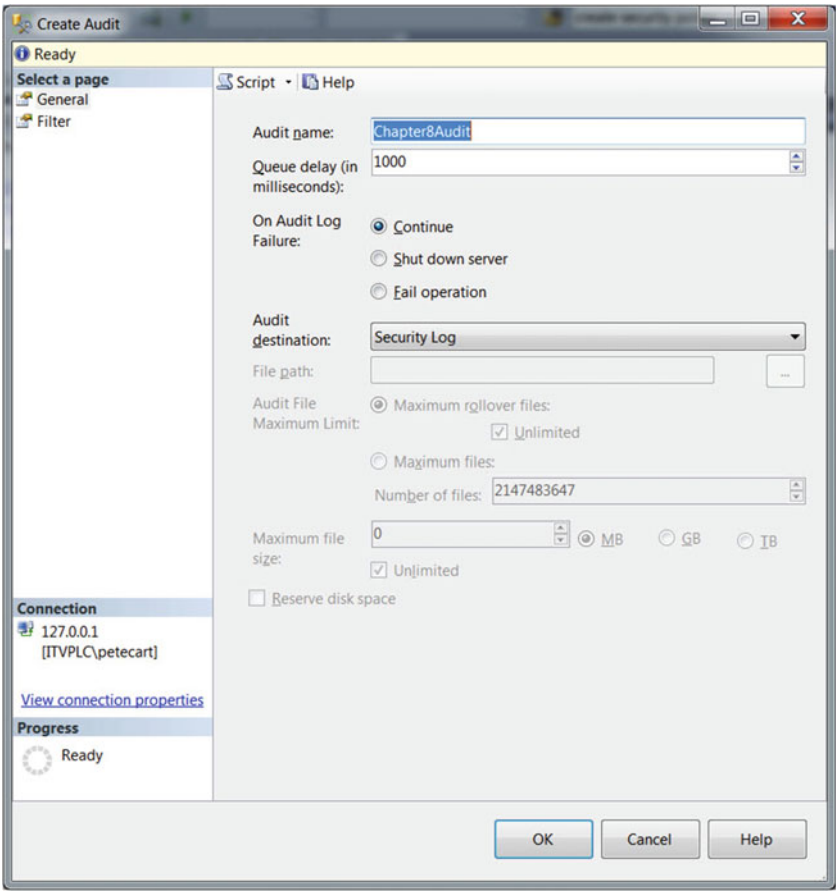


Figure 8-1. *Create Audit dialog box: General page*

This page shows that you gave the audit object a name and specified Security Log as the audit destination. This means that ill-intentioned DBAs are unable to tamper with the results.

■ **Note** Using the Windows Security log as a destination means that additional configuration is required at the operating system level; this is discussed shortly.

On the Filter page of the dialog box, illustrated in Figure 8-2, you add a filter so that only changes made to the `sa` account are audited, as opposed to auditing changes for all logins.

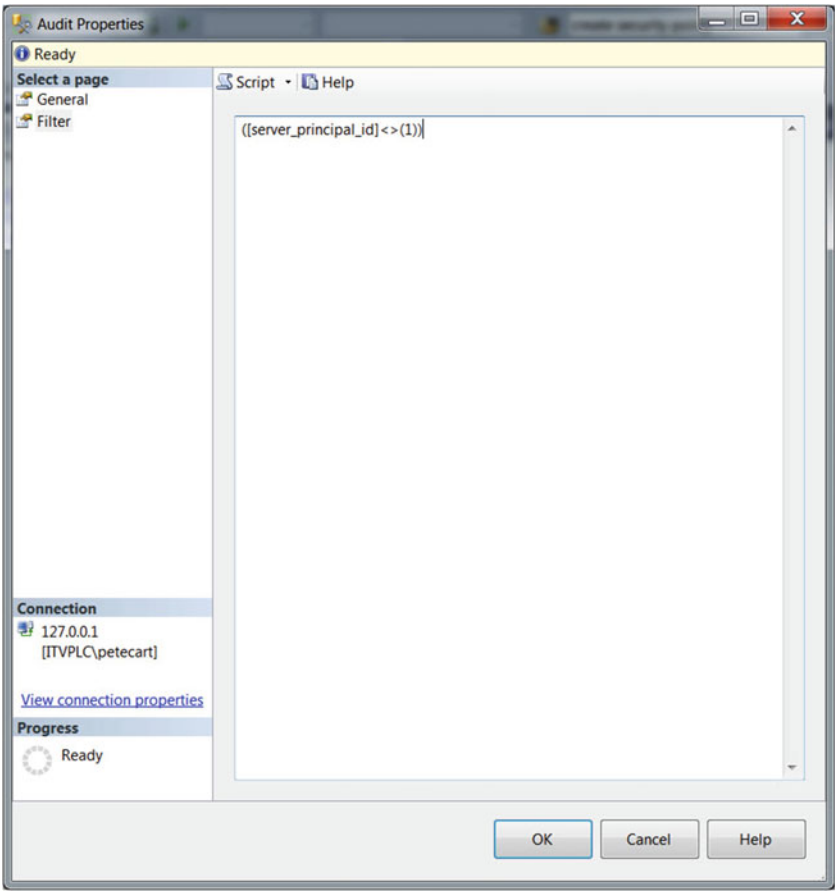


Figure 8-2. *Create Audit dialog box: Filter page*

The sa account always has a server_principal_id of 1, but this can be confirmed by using the query in Listing 8-4.

Listing 8-4. Check server_principal_id of the sa Account

```
SELECT
    name
   ,server_principal_id
FROM sys.server_principals
WHERE name = 'sa' ;
```

You will now create a server audit specification, which defines the audit action groups that you wish to log. To create a new server audit specification, drill through the Security node in Object Explorer and select New Server Audit Specification from the Server Audit Specifications context menu. This causes the Create Server Audit Specification dialog box to be displayed, as illustrated in Figure 8-3.

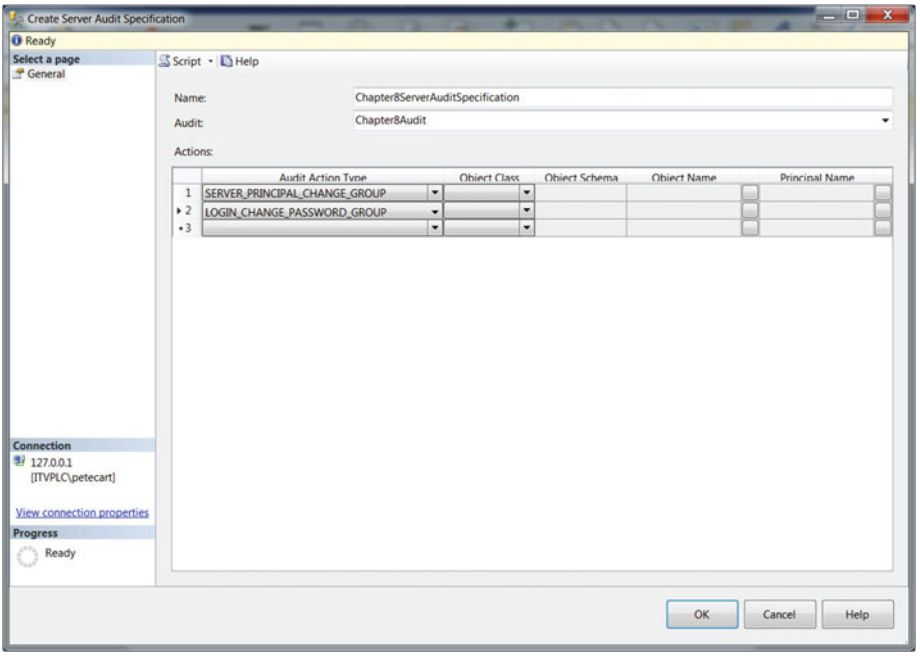


Figure 8-3. *Create New Server Audit Specification dialog box*

This dialog box shows that you have given the server audit specification a name and used the Audit drop-down box to link it to the Chapter8Audit server audit object.

In the Actions pane, you use the drop-down boxes in the Audit Action Type column to add the SERVER_PRINCIPAL_CHANGE_GROUP and LOGIN_CHANGE_PASSWORD_GROUP audit action groups. The query in Listing 8-5 returns a list of actions that is audited by these two groups.

Listing 8-5. List Audit Actions

```
SELECT
    name
        ,covering_parent_action_name
FROM sys.dm_audit_actions
WHERE covering_parent_action_name IN ('LOGIN_CHANGE_PASSWORD_GROUP','SERVER_
PRINCIPAL_CHANGE_GROUP') ;
```

If you were to attempt to enable the audit at this point, then the action would fail because you have not configured the operating system to allow the SQL Server database engine service account the appropriate permissions to the Windows Security log.

The first action that you need to take to configure the operating system is to alter the audit policy to allow application-generated audit events to occur. This can be performed using the auditpol.exe utility. Open the command prompt as an administrator and run the command in Listing 8-6.

Listing 8-6. Enable Application-Generated Audit Events

```
auditpol /set /subcategory:"application generated" /success:enable /
failure:enable
```

The second task is to grant the SQL Server database engine service account the Generate Security Audits user rights assignment. This is performed by using the Local Security Policy Windows snap-in. Run the secpol.exe utility and drill through Security Options and User Rights Assignment in the Local Security Policy console. This is illustrated in Figure 8-4.

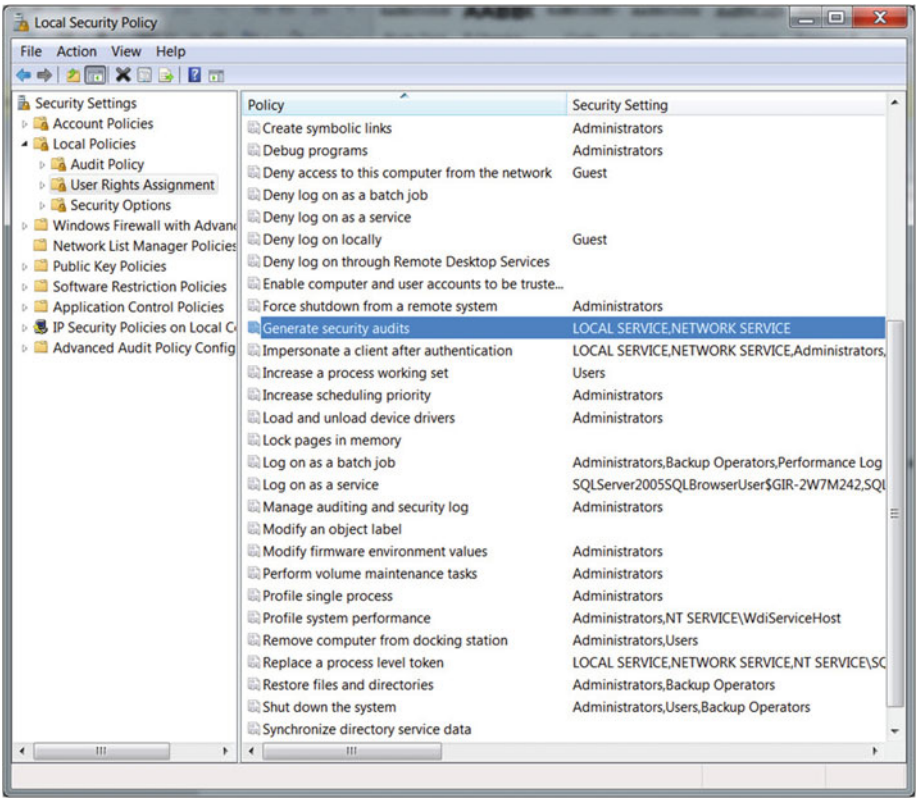


Figure 8-4. *Local Security Policy console*

Now double-click the Generate Security Audits User Rights Assignment to display the Generate Security Audits Properties dialog box, which is shown in Figure 8-5.

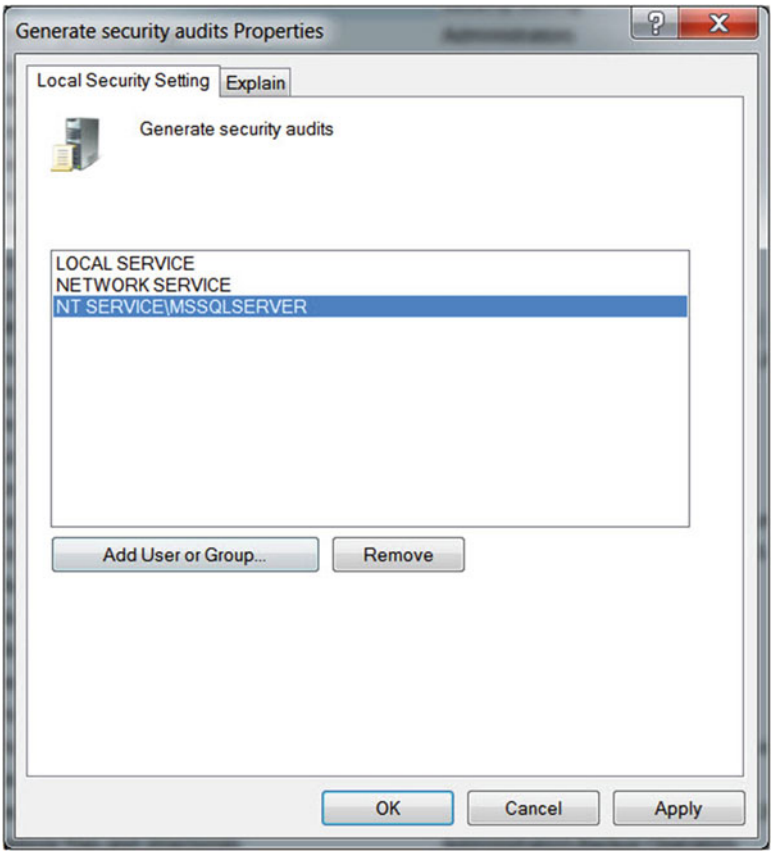


Figure 8-5. *Generate Security Audits Properties dialog box*

In this dialog box, you use the Add Users or Group button to add the database engine service account, thus granting it the user rights assignment.

The server audit specification and server audit objects can now be enabled by selecting Enable from the context menus in Object Explorer.

Enforcing Constant Password Changes

If you are in a situation where you have instances that use mixed mode authentication, but you want to avoid the `sa` account from being used, another method (that I have seen implemented at some FTSE 100 companies) is to change the password of the `sa` account every hour. This is a deterrent against software development teams promoting code that relies on the use of the `sa` account. They know in advance that if they do this, their release almost instantly starts failing. It also avoids the possibility of DBAs re-enabling the account "for convenience."

If your organization runs enterprise-level security software such as CyberArk, then this tooling configures rapid cycling of the `sa` account password; otherwise, you can create a simple routine using SQL Server.

To create a routine in SQL Server, simply create a SQL Server Agent job. To do this, drill through SQL Server Agent in Object Explorer and select New Job from the Jobs context menu. This causes the General page of the New Job dialog box to be displayed. On this page, you specify a name for the job and a job owner. In this instance, you have used the database engine service account as the owner, as shown in Figure 8-6.

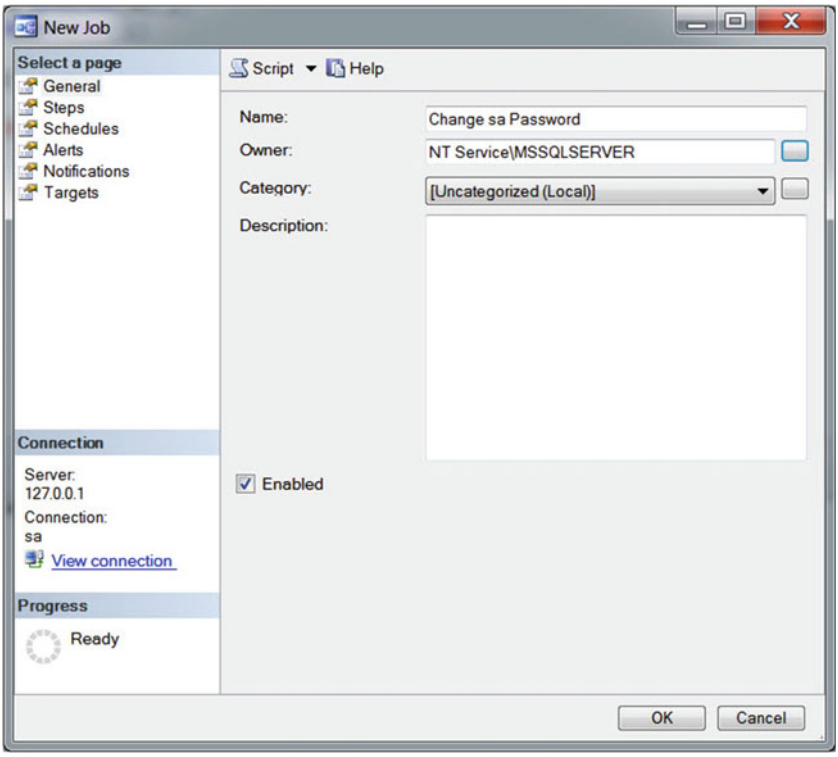


Figure 8-6. *New Job dialog box: General page*

On the Steps page, you use the New button to invoke the New Job Step dialog box, which is illustrated in Figure 8-7. In this dialog box, you specify a name for the job step, ensure that the Type drop-down is configured as `Transact-SQL Script(T-SQL)`, and enter the script that you want to run.

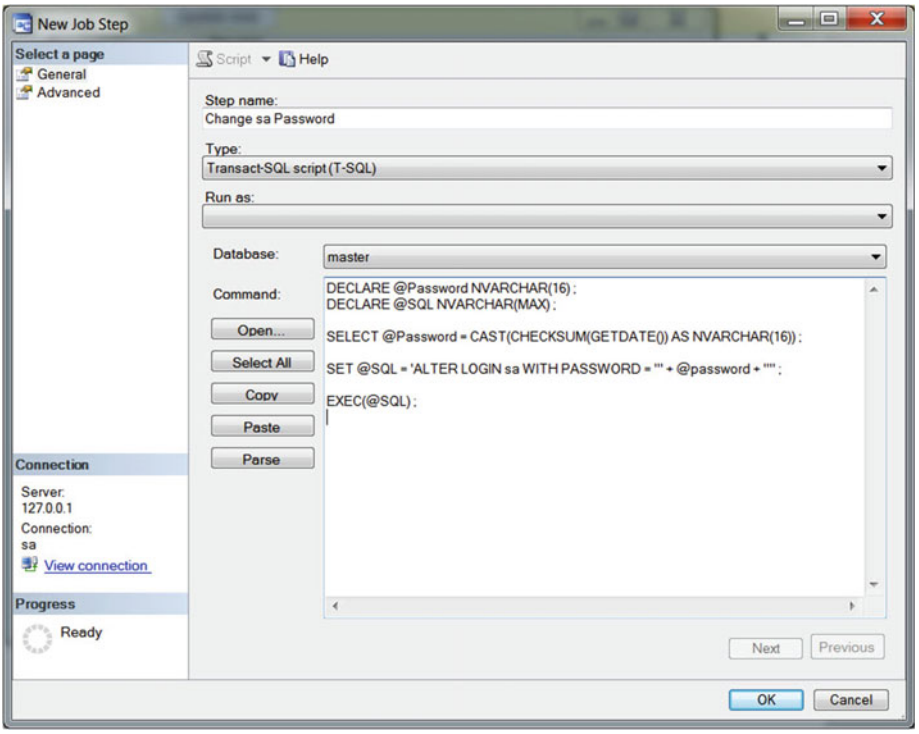


Figure 8-7. *New Job Step dialog box: General page*

The script itself uses the `CHECKSUM()` function to return a hashed representation of the current timestamp, which is retrieved using the `GETDATE()` function. This value is then converted to a `NVARCHAR(16)` and used as the new password for the `sa` account.

■ **Tip** The command to actually change the `sa` account password is executed as dynamic SQL because a variable cannot be passed to the `ALTER LOGIN` statement.

Listing 8-7 provides the script that is used in the SQL Server Agent job step.

Listing 8-7. Change sa Account Password to a Dynamic Value

```
DECLARE @Password NVARCHAR(16) ;
DECLARE @SQL NVARCHAR(MAX) ;

SELECT @Password = CAST(CHECKSUM(GETDATE()) AS NVARCHAR(16)) ;

SET @SQL = 'ALTER LOGIN sa WITH PASSWORD = ''' + @password + '''' ;

EXEC(@SQL) ;
```

After exiting the New Job Step dialog box, you return to the New Job dialog box, where you should navigate to the Schedule page and use the New button to invoke the New Schedule dialog box, which is shown in Figure 8-8. In this dialog box, you specify a name for the schedule and use the Occurs drop-down list to indicate that you want the schedule to run daily. Finally, you use the Occurs Every radio button to specify that the schedule should run every hour. After exiting the New Schedule dialog box, you can also exit the New Job dialog box, and the job is created.

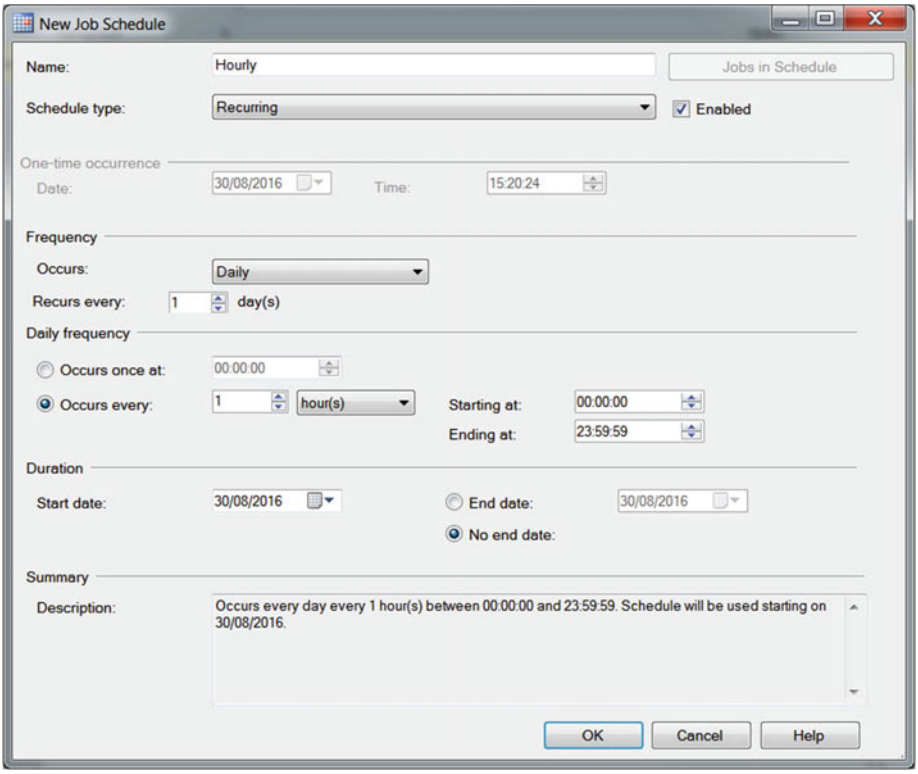


Figure 8-8. *New Schedule dialog box*

■ **Tip** The New Schedule dialog box is dynamic. Making changes to the Schedule Type or Occurs drop-down lists causes the dialog box to be refreshed with different options.

A full discussion of SQL Server Agent is beyond the scope of this book. For further information, or to discover how to use Server Agent to schedule this (and other) jobs across the enterprise, I recommend my book *Expert Scripting and Automation for SQL Server DBAs* (Apress, 2016).

Protecting User Accounts

Logins (other than the sa account) can also come under attack; therefore, it is important to ensure that passwords are complex and changed frequently to avoid brute-force attacks and word list attacks. A brute-force attack occurs when an attacker attempts to log in by using every possible combination of letters and numbers as the password. During a word list attack, a hacker uses a password dictionary to try to crack a password; this attempt to authenticate uses a single login and a vast array of passwords.

The best way of ensuring that passwords are complex and are frequently changed is to enforce the domain-level password policy. If you are in a situation, where you have not implemented this policy enforcement, however, it may be a good idea to perform spot-checks to ensure that logins are not using very common passwords.

■ **Tip** Even if your environment does enforce password policies, it is still a good idea to perform an occasional spot-check, because it is fairly simple for privileged users to circumvent the functionality.

Auditing Passwords Susceptible to Word List Attacks

Common passwords are the first passwords that are attempted during a word list attack. The following is a list of the 25 most common passwords (as collated by SplashData at https://www.teamsid.com/worst-passwords-2015/), ranked by the most common.

1. 123456
2. password
3. 12345678
4. qwerty
5. 12345
6. 123456789
7. football
8. 1234
9. 1234567
10. baseball
11. welcome
12. 1234567890
13. abc123
14. 111111

15. 1qaz2wsx
16. dragon
17. master
18. monkey
19. letmein
20. login
21. princess
22. qwertyuiop
23. solo
24. passw0rd
25. starwars

To audit the logins within an instance to ensure that none of these words are used as passwords, you can use the `PWDCOMPARE()` function. The `PWDCOMPARE()` function accepts three parameters, which are described in Table 8-1. The `PWDCOMPARE()` function returns 1 if the clear text password matches the password hash, and 0 if it does not match.

Table 8-1. *PWDCOMPARE Parameters*

Parameter	Description
`clear_text_password`	Specifies the clear-text version of a password that you wish to compare.
`password_hash`	The hashed password value that you wish to audit.
`version`	This parameter is obsolete and should not be used.

The script in Listing 8-8 demonstrates how the `PWDCOMPARE()` function checks to ensure that no logins within the instance use any of the 50 most common passwords. The script first creates a temporary table, which holds the list of 25 most common passwords. It then uses a `CROSS JOIN` to create a combination of every possible password in the list and the password of every login in the instance. It then filters the results on where the password in the list matches the hashed version of the password from the `sys.logins` catalog view.

Listing 8-8. Audit Common Passwords

```
CREATE TABLE ##Passwords
(
[Password]    NVARCHAR(128)
) ;

INSERT INTO ##Passwords
VALUES ('123456'),
```

```
('password'),
('12345678'),
('qwerty'),
('12345'),
('123456789'),
('football'),
('1234'),
('1234567'),
('baseball'),
('welcome'),
('1234567890'),
('abc123'),
('111111'),
('1qaz2wsx'),
('dragon'),
('master'),
('monkey'),
('letmein'),
('login'),
('princess'),
('qwertyuiop'),
('solo'),
('passw0rd'),
('starwars') ;

SELECT l.name,
       p.[password]
FROM sys.sql_logins l
CROSS JOIN ##Passwords p
WHERE PWDCOMPARE(p.Password,l.password_hash) = 1 ;

DROP TABLE ##Passwords ;
```

Summary

It is important that DBA professionals reduce the likelihood of a successful attack by enforcing password policies and ensuring that those policies are being adhered to. The `sa` account is a particular target for attackers, so it is best practice to use Windows authentication wherever possible. When this is not possible and mixed mode authentication must be used, DBAs should disable or rename the `sa` account. If this is not possible, then at a minimum, auditing should be configured to avoid non-reputability for internal attacks.

Other logins may also be targeted by attackers with either brute-force or word list attacks. To mitigate the risks of this, you should enforce domain-level password policies wherever possible. If you inherit an enterprise where this has not been enforced, then you can check for logins that have common passwords configured by using the `PWDCOMPARE()` function.

CHAPTER 9

Reducing the Attack Surface

The surface area of SQL Server comprises all aspects of the suite that can potentially be attacked. This includes features, services, and endpoints. The attackable surface area can also be increased or reduced by operating system, or network components, such as firewall design. The larger the attack surface, the greater the chance of a determined attacker successfully exploiting a vulnerability. The following sections discuss network configuration and ensuring that unsafe features are not turned on.

Network Configuration

The following sections provide an overview of ports and protocols, before diving into considerations for firewall configuration.

Understanding Ports and Protocols

The following sections discuss protocols, static vs. dynamic ports, and policy-based management.

When computers communicate with each other across a network, they both need to understand exactly how information is exchanged and what format it is in. This information is laid out using protocols. SQL Server is able to listen on three different protocols:

- Shared memory
- Named pipes
- TCP/IP

Shared memory can only be used when the connecting client is on the same server as the database engine instance. Because it is not good practice for applications to be installed on the same server as SQL Server, then it is rare that shared memory can be used. Its principal usage is in troubleshooting scenarios when you suspect that there may be an issue with other protocols.

Named pipes is a protocol that was designed to be used with a LAN. When used across a WAN, named pipes can still be used, but performance can become an issue due to a series of named pipe messages needing to be sent from the client before the network read begins.

P. A. Carter, *Securing SQL Server*, DOI 10.1007/978-1-4842-2265-2_9

■ **Tip** From SQL Server 2008 onward, Kerberos is supported for named pipes, as well as TCP/IP. This is due to the format of SPNs (service principal names) changing.

TCP/IP is generally the protocol of choice in corporate environments. It has many security features built-in and also includes standards for network routing. Unless there is a specific reason for enabling named pipes, most SQL Server instances should be configured to use TCP/IP only.

Static vs. Dynamic Ports

A port is a communication endpoint within an operating system, which are used by transport layer protocols. SQL Server uses TCP (Transmission Control Protocol) ports and UDP (User Datagram Protocol) ports, for clients to establish communication to an instance.

A static port always remains constant, even if the server (or service using the port) is restarted. Some static ports are predefined (such as TCP 1433 for a default instance of SQL Server) and others are configured by an administrator.

Dynamic ports, on the other hand, change each time the service starts. This is because they simply request a port number from the operating system, which assign the service a port number that is not in use by another service within the high port range. In Windows Server 2008 and above, and Windows Vista and above, this range is 49152–65535. Prior to these operating system levels, the dynamic port range was from 1024–5000.

IANA, the Internet Assigned Numbers Authority, is responsible for coordinating the allocation of Internet protocol resources, such as IP addresses, domain names, protocol parameters, and the port numbers of network services. The IANA website is at `www.internetassignednumbersauthority.org`. SQL Server has the following ports registered in the IANA database:

- TCP 1433
- UDP 1433
- TCP 1434
- UDP 1433

■ **Note** The ports used by SQL Server is discussed in the "Ports Required by SQL Server" section of this chapter.

Configuring Protocols

Allowed protocols can be configured in SQL Server Configuration Manager. Drilling through SQL Server Network Configuration | Protocols for [Instance Name] causes a list of protocols to be displayed in the right-hand pane of the Window, as shown in Figure 9-1.

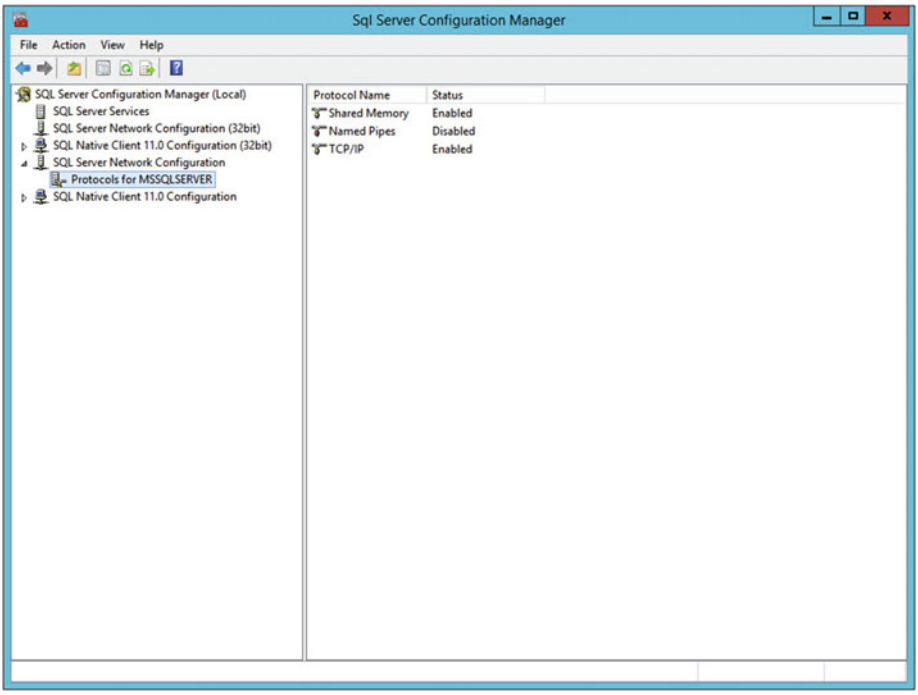


Figure 9-1. *SQL Server protocols*

Entering the context menu of any of these protocols presents you with options to enable or disable (as appropriate) the protocol for the instance. The database engine service needs to be restarted for the change to take effect.

The shared memory protocol has no configurable options, but entering the properties of the named pipes protocol allows you to specify the name of the pipe to use, as illustrated in Figure 9-2.

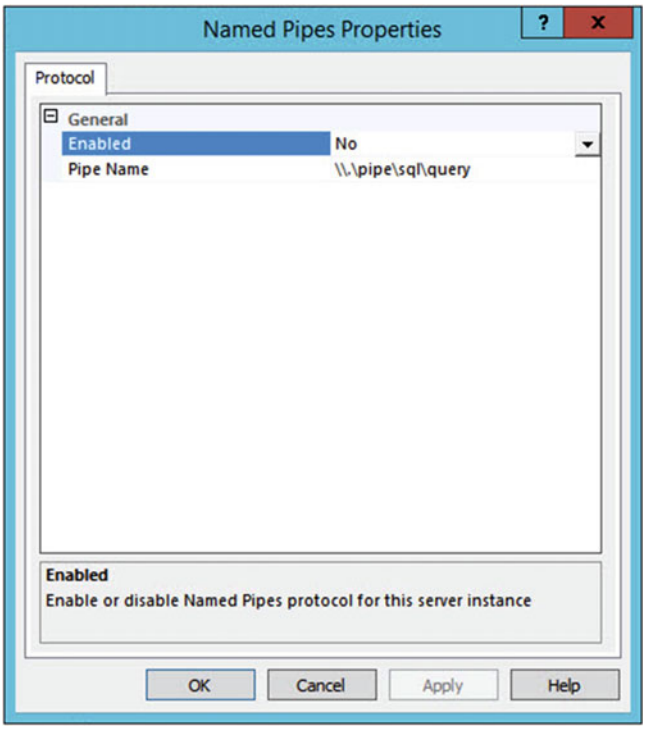


Figure 9-2. *Configuring named pipes*

Entering the properties of the TCP/IP protocol presents you with a dialog box consisting of two tabs. The first tab, illustrated in Figure 9-3, allows you to configure protocol-related options. The Keep Alive setting specifies how long the interval should be between checking that an idle connection is still intact. The Listen All settings specify whether the instance should listen on all available IP addresses, or if a specific subset of IP addresses is configured.

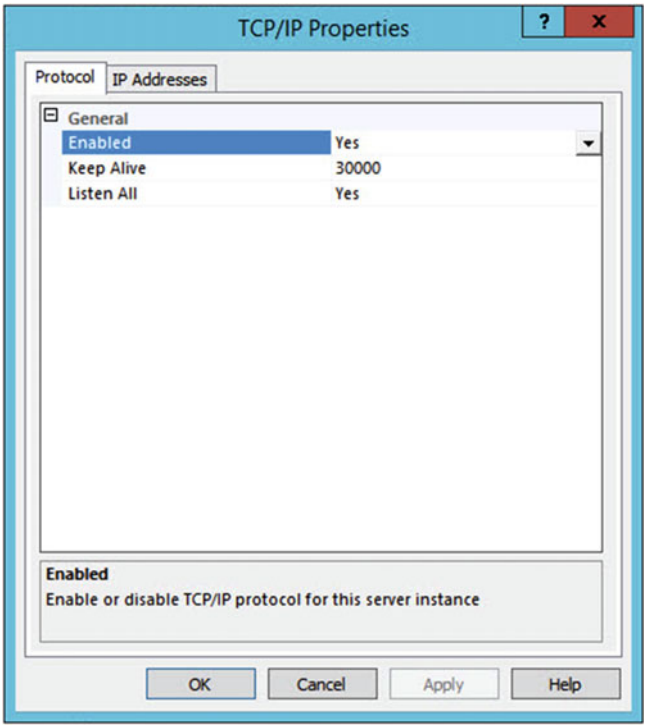


Figure 9-3. *TCP/IP Properties: Protocol tab*

The IP Addresses tab in the TCP/IP Properties dialog box allows you to configure the port that the instance listens on. If the instance uses static ports (which are best practice to assist firewall configuration), then TCP Dynamic Ports should be left blank.

If Listen All in the Protocol tab is set as False, then you should configure the port for each IP address available. You should also mark which IP address(es) should be used by the instance. If Listen All is configured as True, then you should only configure the IPAll section at the bottom of the tab, as all other configurations are ignored. The IP Addresses tab is seen in Figure 9-4.

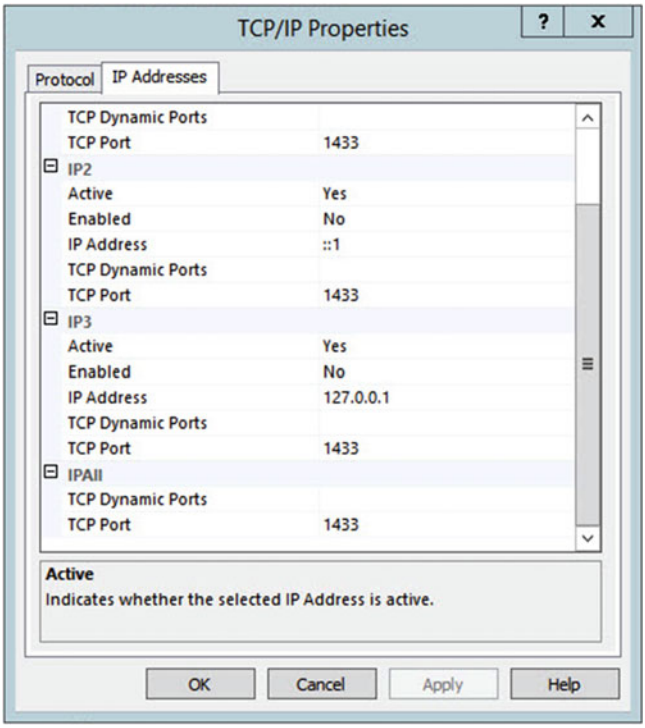


Figure 9-4. *IP Addresses tab*

Firewall Requirements for SQL Server

A *firewall* blocks network traffic to specific ports. Firewall rules open specific ports to allow either open access for communication on specific ports or access to specific IP addresses to communicate on the ports. Ports can be opened for inbound traffic, outbound traffic, or bidirectional traffic.

In a corporate environment, network traffic to and from SQL Server almost certainly travels through at least one firewall, but is more likely to have to travel through several firewalls. Figure 9-5 illustrates a simple firewall topology. When thinking about the attack surface of SQL Server, firewalls are one of the first considerations that you should take into account. Although you are unlikely to be directly responsible for firewall configuration, you need to ensure that the requests you make to the firewall administrators open enough ports for your data-tier application to function correctly, while keeping the attack surface as small as possible.

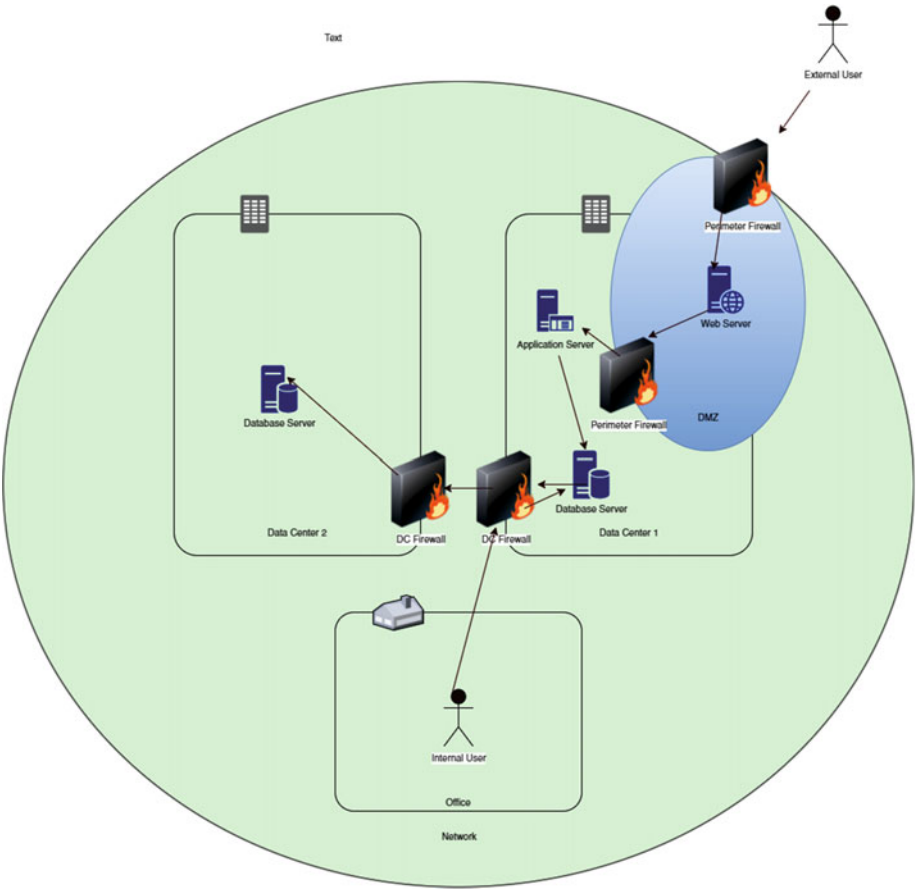


Figure 9-5. *Typical firewall topology*

■ **Tip** More complex firewall technologies may have different subnets for application, database, and user tiers—with firewalls between each of these subnets.

In addition to corporate hardware-based firewalls, some companies also have Windows Firewall (also known as a *local firewall*) in use on their servers. This, of course, adds an additional layer of complexity. And requests to have ports opened need to be directed to the Windows administration team and the firewall administration team.

How Clients Communicate with SQL Server

When using the TCP/IP protocol, clients can either communicate directly with a named instance of SQL Server by specifying the port number that the instance listens on in the connection string, or by passing the instance name and letting the SQL Server Browser service resolve the name. If a client communicates with a default instance of SQL Server, then it can connect directly—without the need for the SQL Server Browser service. The diagram in Figure 9-6 illustrates the decision tree process that occurs when a client communicates with an instance.

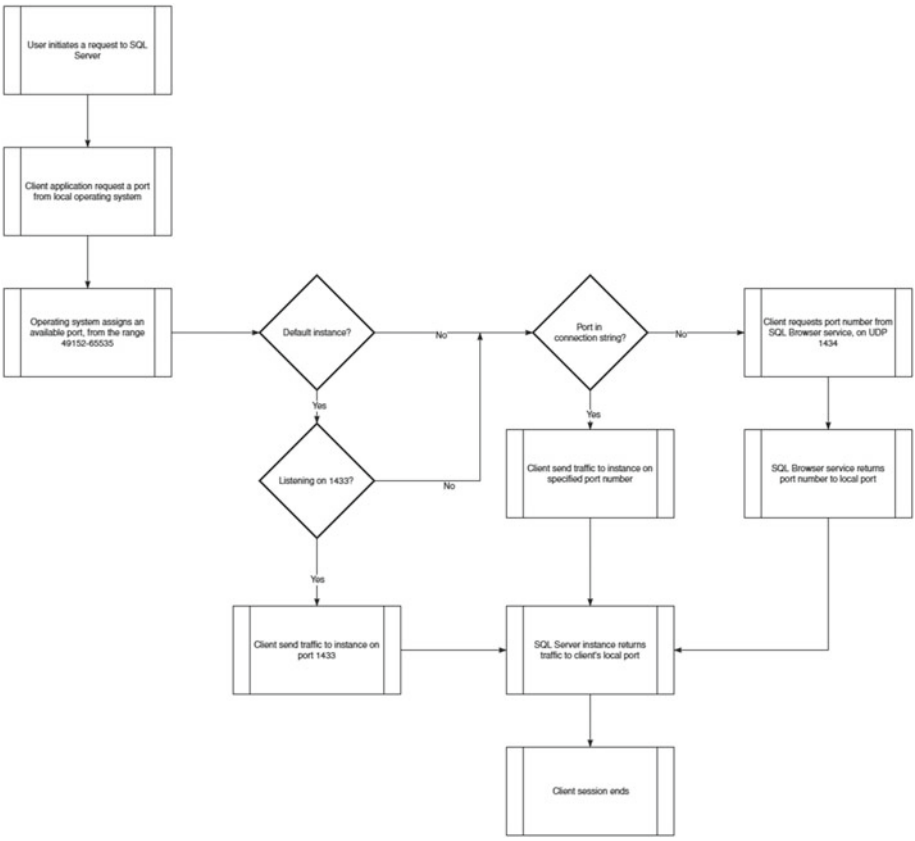


Figure 9-6. *Communication process*

When a client communicates on named pipes, as opposed to TCP/IP, then traffic is always sent on port 445. This is the port normally used for file and printer sharing.

■ **Tip** If you do not plan to use named pipes, then disabling file and printer sharing and blocking port 445 is a good way to reduce the attack surface.

Ports Required by SQL Server

Depending on the SQL Server features that you plan to use, there are many different ports that may be required by SQL Server. These ports are described in Table 9-1.

Table 9-1. *Ports Required by SQL Server*

Feature/Component	Port Requirements
Default instance	TCP 1433 (can be changed by a DBA)
Named instance	Dynamic port (can be changed by a DBA)
DAC (dedicated administrator connection) on default instance	TCP 1434
DAC on named instance	Dynamic port
SQL Server Browser service	UDP 1434
Instance running over HTTP endpoint	TCP 80 (can be changed by an IIS administrator)
Instance running over HTTPS endpoint	TCP 443 (can be changed by an IIS administrator)
Service Broker	No default port, but 4022 is used as standard
Database mirroring	No default port, but 7022 is used as standard. If there are multiple instances with database mirroring endpoints on the same server, then the port number is often incremented with + 1
AlwaysOn Availability Groups	No default port, but 7022 is used as standard. If there are multiple instances with database mirroring endpoints on the same server, then the port number is often incremented with + 1 (Availability Groups use the database mirroring endpoint)
Replication - Instance connections	Connections to the instance use the port configured for the instance
Replication - Web sync	TCP 80 (can be changed by an IIS administrator and a DBA)
Replication FTP	TCP 21 (can be changed by a DBA)
Replication - File sharing	UDP 137, UDP 138, TCP 139 (TCP 445 is also required if NetBIOS is used)
T-SQL debugger	TCP 135
Analysis Services	TCP 2382 (can be changed by a DBA)
SQL Server Browser service (when used with Analysis Services named instances)	TCP 2382

(*continued*)

Table 9-1. (*continued*)

Feature/Component	Port Requirements
Analysis Services over HTTP	TCP 80 (can be changed by an IIS administrator)
Analysis Services PivotTable service over HTTP	TCP 80 (can be changed by and IIS administrator)
Analysis Services over HTTPS	TCP 443 (can be changed by an IIS administrator)
Analysis Services PivotTable service over HTTPS	TCP 443 (can be changed by an IIS administrator)
Reporting Services Web Service through HTTP	TCP 80 (can be changed by an IIS administrator)
Reporting Services Web Service through HTTPS	TCP 443 (can be changed by an IIS administrator)
Integration Services Runtime	TCP 135
WMI (Windows Management Instrumentation)	TCP 135
MSDTC (Microsoft Distributed Transaction Coordinator)	TCP 135
IPSec (encrypts server-to-server communications	UDP 500 and UDP 5000

Ensuring that Unsafe Features Remain Disabled

SQL Server disables unsafe features (features that increase the attack surface) by default. If you need to turn on any of these features (or off again), you can do so via SQL Server Management Studio (SSMS). The following sections discuss how to manually configure the surface area and how to manage the surface area with Policy-Based Management.

Manually Configuring the Surface Area

To enable or disable features through SSMS, enter the instance context menu in Object Explorer and select Facets. In the View Facets dialog box, you can now select Surface Area Configuration from the drop-down list of available facets. Upon selecting the facet, the lower pane of the screen automatically updates to reveal the current status of each feature, as illustrated in Figure 9-7.

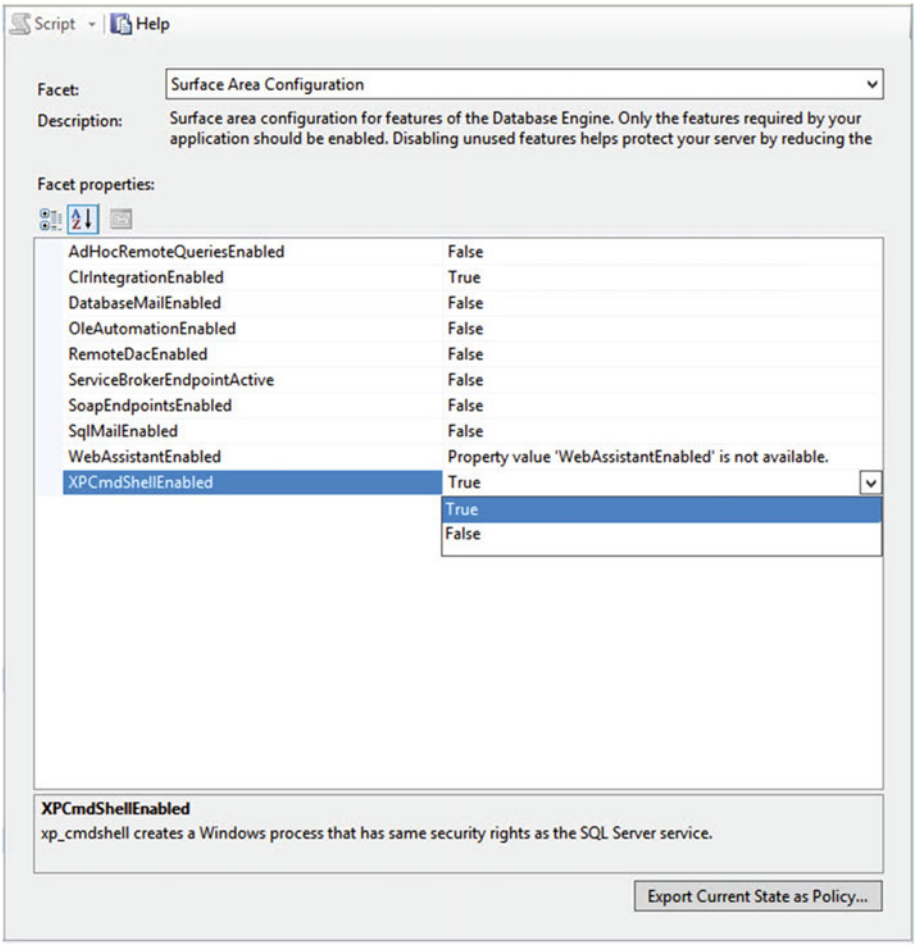


Figure 9-7. *Surface Area Configuration facet*

In this particular case, you can see that CLR Integration is enabled, as is `xp_cmdshell`. You can use the drop-down box next to each feature to change the status.

Managing Features with Policy-Based Management

While managing which features are enabled manually may be OK for a few instances, if you have a large enterprise to manage, then it quickly becomes impossible. In this scenario, you can use Policy-Based Management (PBM) to help you.

Policy-Based Management Concepts

Policy-Based Management comprises of targets, facets, conditions, and policies. Targets are entities that PBM manages, such as databases or tables—or for this purpose, the surface area. Facets are collections of properties that relate to a target. For example, the surface area configuration facet includes the same properties as the surface area configuration instance level facet that was described earlier.

■ **Tip** The properties within this facet control the surface area of the database engine only. Additional facets are supplied for Analysis Services and Reporting Services. These additional facets are exposed through Policy-Based Management only; they are not available as configurable instance-level facets.

Conditions are Boolean expressions that can be evaluated against a property. A policy binds conditions to targets. The following sections discuss each of these concepts.

Facets

A *facet* is a collection of properties that relate to a type of target, such as View, that has properties, including `IsSchemaBound`, `HasIndex`, and `HasAfterTrigger`. SQL Server 2016 provides 96 facets in all; for reducing the surface area, the following are three facets of interest:

- ISurfaceAreaConfigurationForAnalysisServer
- ISurfaceAreaConfigurationForReportingServices
- ISurfaceAreaFacet

Conditions

A condition is a Boolean expression that is evaluated against an object property to determine whether or not it matches a specified value. Each facet contains multiple properties that you can create conditions against, but each condition can only access properties from within a single facet. Conditions can be evaluated using the following operators:

- =
- !=
- LIKE
- NOT LIKE
- IN
- NOT IN

Targets

A *target* is an entity to which a policy can be applied. This can be almost any object within SQL Server, such as a table, a database, or an instance. When adding targets to a policy, you can use conditions to limit the number of targets. This means, for example, if you

create a policy to enforce database naming conventions on an instance, you can use a condition to avoid checking the policy against database names that contain the words "SharePoint," "bdc," or "wss," since these are your SharePoint databases and they may contain GUIDs that would be disallowed under your usual naming conventions.

Policies

A policy contains one condition and binds this condition to one or more targets (targets may also be filtered, using separate conditions). The policy also specifies an evaluation mode. Depending on the evaluation mode that you select, the policy may also contain a schedule on which you would like the policy to be evaluated. Policies support the following four evaluation modes:

- On Demand
- On Schedule
- On Change: Log Only
- On Change: Prevent

If the evaluation mode is configured as On Demand, then the policy is only evaluated when a DBA manually evaluates them. If the evaluation mode is configured as On Schedule, then you create a schedule at the point when you create the policy. The policy is then evaluated periodically in line with the schedule specification.

If the evaluation mode is configured as On Change: Log Only, then whenever the relevant property of a target changes, the policy is evaluated. If the change has caused the policy to fail validation, a message is generated in the log, therefore logging any violation of your policies. If the policy is violated, then Error 34053 is logged with a severity level of 16.

If the evaluation mode is configured as On Change: Prevent, then when a property is changed, SQL Server evaluates the property, and if there is a violation, an error message is thrown and the statement that caused the policy violation is rolled back.

Because policies work based on DDL events being fired, depending on the properties within the facet, not all evaluation modes can be implemented for all facets. Table 9-2 specifies the evaluation modes that can be configured for each of the surface area configuration facets.

Table 9-2. *Evaluation Modes Supported by Surface Area Configuration Facets*

Facet	On Demand	On Schedule	On Change: Log Only	On Change: Prevent
`ISurfaceAreaFacet`	YES	YES	YES	NO
`ISurfaceArea Configuration ForAnalysis Server`	YES	NO	NO	NO
`ISurfaceArea ConfigurationFor ReportingServices`	YES	NO	NO	NO

■ **Tip** In addition to evaluating surface area, Policy-Based Management can help with implementing security in other ways. For example, imagine that you wanted to ensure that developers did not elevate their own permissions by unauthorized use of `EXECUTE AS` in their code. In this scenario, you could create a policy that prevented any stored procedures from being created or modified if they contained the string `EXECUTE AS`.

Creating a Policy for Surface Area Configuration

To create a policy to manage the surface area of the database engine, you need to create two objects: a condition and a policy.

To create the condition, drill through the Management ➤ Policy-Based Management in Object Explorer and select New Condition from the context menu of the Conditions node. This causes the Create New Condition dialog box to be invoked, as illustrated in Figure 9-8.

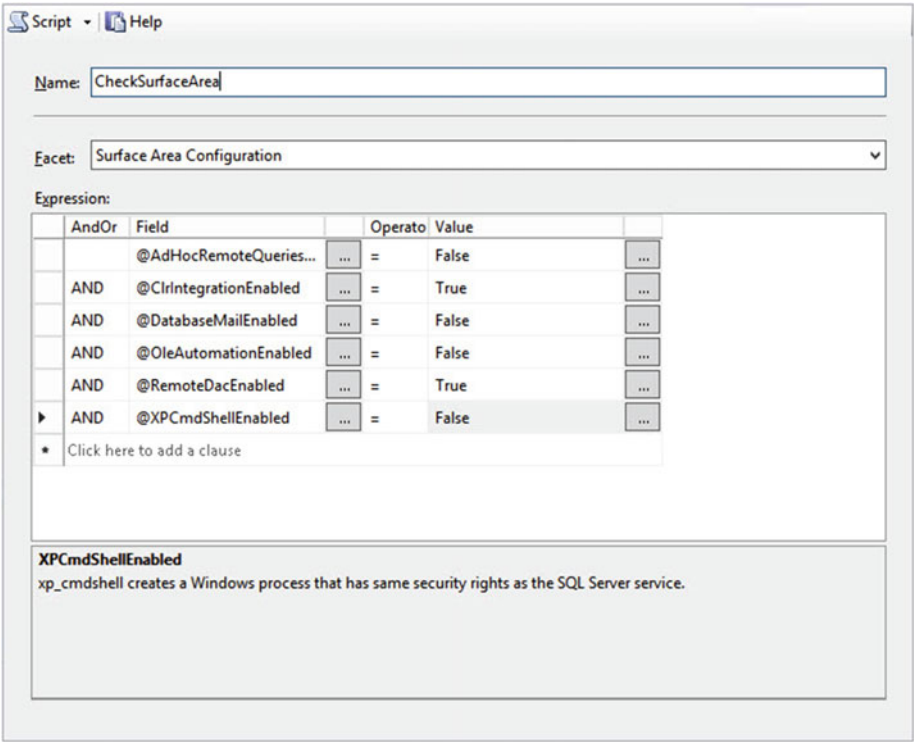


Figure 9-8. *Create New Condition dialog box*

In this dialog box, you specify a name for the condition and then select the Surface Area Configuration facet from the Facet drop-down box.

■ **Note** Notice that when viewed through SSMS, the friendly name of the facet is returned, as opposed to the system name. For example, the facet that you are using is displayed as Surface Area Configuration, as opposed to ISurfaceAreaFacet.

In the Expression area of the screen, you select the properties that you want to include in the condition by using the drop-down boxes in the Field column. You specify whether they should be enabled or disabled in the Value column.

You can now create the policy by drilling through Management ➤ Policy-Based Management in Object Explorer, and then selecting New Policy from the Policies context menu. This causes the Create New Policy dialog box to be invoked, as illustrated in Figure 9-9.

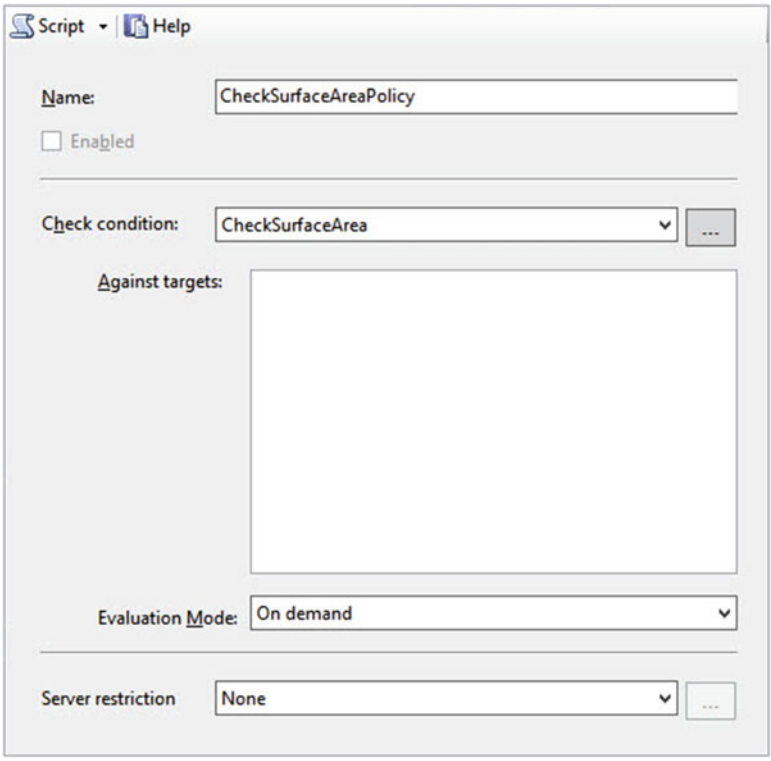


Figure 9-9. *Create New Policy dialog box*

In this dialog box, you specify a name for the policy, and then select the condition that you want to evaluate from the Check condition drop-down list. You select On Demand as the evaluation mode, meaning that the policy is only evaluated when a DBA manually evaluates it.

■ **Tip** Because you have chosen the On Demand evaluation mode, the Enable check box is not selectable. This does not affect the ability to evaluate the policy. Policies can also be evaluated manually, regardless of the evaluation mode or enabled status.

Evaluating the Policy Against a Single Instance

To evaluate the policy against the instance where it was created, drill through Management ➤ Policy-Based Management ➤ Policies. And then select Evaluate from your policy's context menu. This causes the Evaluate Policies dialog box to be invoked, as illustrated in Figure 9-10.

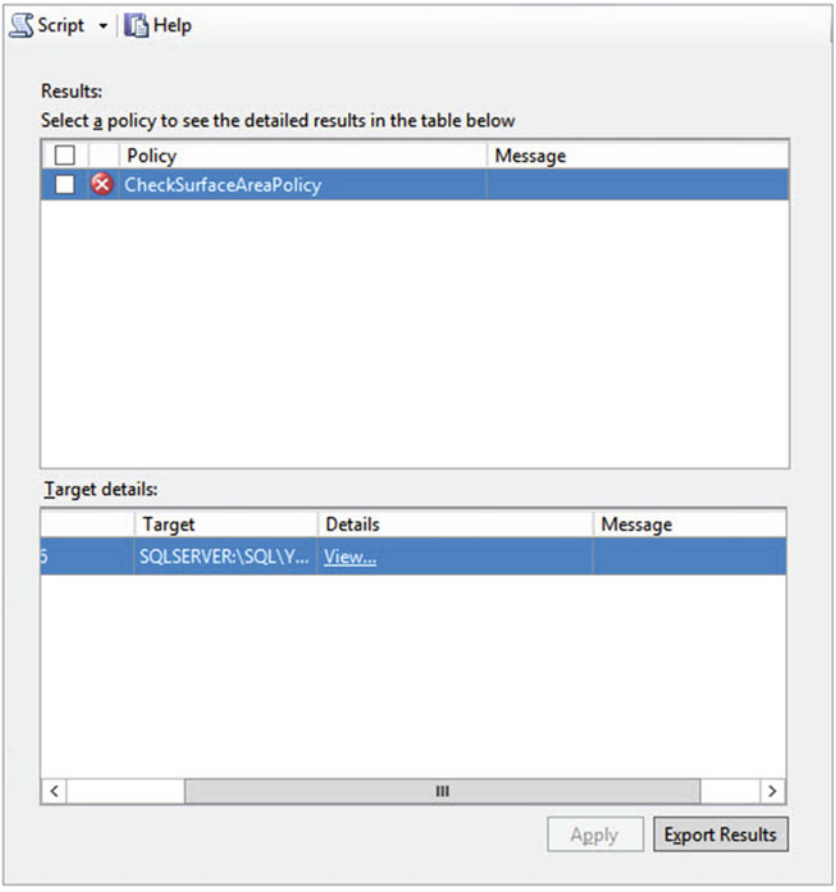


Figure 9-10. *Evaluated Policies dialog box*

The Evaluate Policies dialog box shows each policy that has been evaluated; its evaluation status is in the top pane of the window. The lower pane of the window describes the status of policy against each target. Clicking the View link in the Details column displays the Results Detailed View dialog box, as shown in Figure 9-11. As you can see, this dialog box provides the status of each property that has been evaluated.

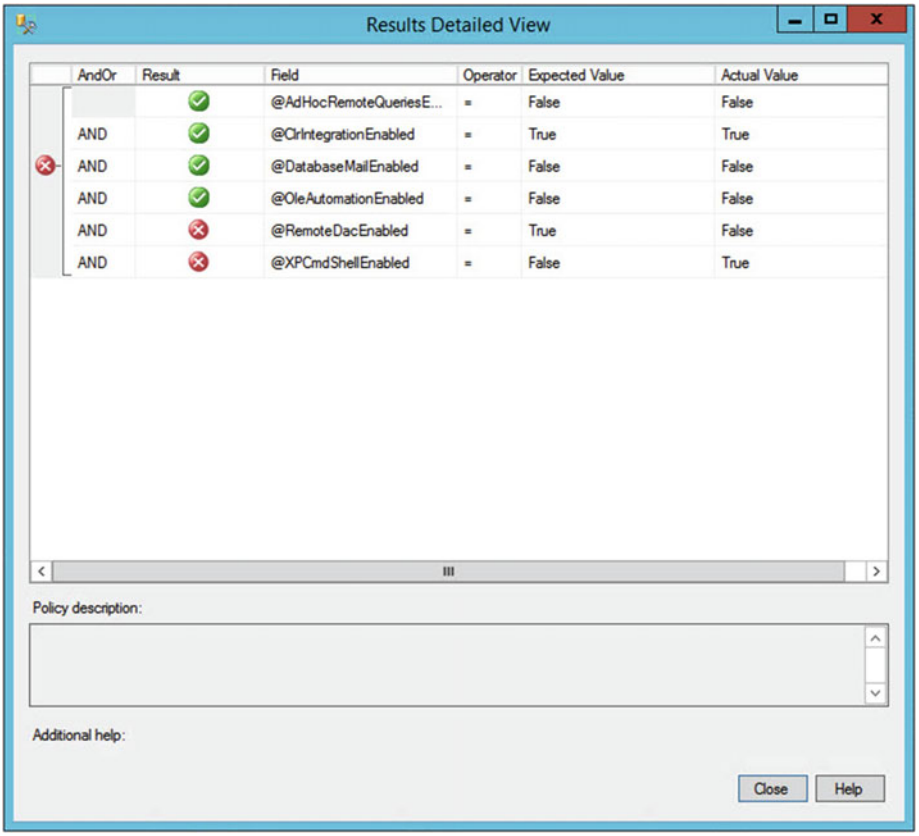


Figure 9-11. *Results Detailed View dialog box*

Evaluating Policies Against Multiple Instances

While evaluating policies against a single instance certainly has merit in some scenarios, when evaluating the surface area of the database engine, there is no value added—over what can be viewed within the instance level Surface Area Configuration facet.

The real benefit of Policy-Based Management comes from the ability to evaluate a policy against a large number of instances at the same time. To achieve this, you need to use ac management server.

Central management servers are provided by SSMS, which allows you to register an instance as a central management server and then register other instances as registered servers of this central management server. Once you have registered servers under a central management server, you can run queries or evaluate policies against all servers managed by the CMS (central management server). Alternatively, you can create server groups underneath the CMS, which gives you the flexibility to run queries or evaluate policies against servers within a specific group.

■ **Note** A full discussion of central management servers is beyond the scope of this book. For a detailed discussion on the subject, however, I recommend my book *Pro SQL Server Administration* (Apress, 2015).

Policies can also be evaluated against multiple instances using PowerShell. With this approach, the `Invoke-PolicyEvaluation` cmdlet evaluates a policy against a target server, which can be placed in a `ForEach` loop to run the cmdlet against multiple servers.

■ **Tip** A full discussion on using PowerShell to manage SQL Server can be found in my book *Expert Scripting and Automation for SQL Server DBAs* (Apress, 2016).

Summary

The SQL Server surface area comprises all aspects of the suite that can potentially be attacked. Reducing the attack surface makes it harder to launch a successful threat against your SQL Server instance(s).

Understanding the port requirements of SQL Server is key to implementing a secure firewall policy. Most firewall engineers are aware that port 1433 is used as a standard by the default instance of SQL Server; but they may not be aware of the other port requirements. This can lead to confusion and ultimately too many ports being opened just to get an application working. As a SQL Server professional, being able to advise and guide the firewall team leads to a more secure environment.

It is important to ensure that unsecure features remain disabled, unless they are specifically required and there is no workaround. Unsecure features can be configured manually for an instance by using the Surface Area Configuration facet.

Policy-Based Management provides a Surface Area Configuration facet and also provides additional facets for evaluating the surface area of SQL Server Analysis Services and SQL Server Reporting Services. Once you have created policies, they can either be evaluated locally against an instance or they can be evaluated across many instances using either central management servers or PowerShell.

Index

P. A. Carter, *Securing SQL Server*, DOI 10.1007/978-1-4842-2265-2

E

F

G

H

I, J, K, L

M

N, O

P, Q

R

S

T

U

V, W, X, Y, Z