MySQL Security Best Practices

Securing Amazon Aurora MySQL and Amazon RDS for MySQL resources on AWS

First published August 28, 2024

Last updated August 28, 2024

Notices

Customers are responsible for making their own independent assessment of the information in this document. This document: (a) is for informational purposes only, (b) represents current AWS product offerings and practices, which are subject to change without notice, and (c) does not create any commitments or assurances from AWS and its affiliates, suppliers, or licensors. AWS products or services are provided “as is” without warranties, representations, or conditions of any kind, whether express or implied. The responsibilities and liabilities of AWS to its customers are controlled by AWS agreements, and this document is not part of, nor does it modify, any agreement between AWS and its customers.

Contents

Abstract and introduction

Abstract

Amazon Relational Database Service (Amazon RDS) provides a managed platform on which customers can run a variety of relational databases, including MySQL, MariaDB, PostgreSQL, SQL Server, Oracle, Amazon Aurora MySQL-Compatible Edition, and Amazon Aurora PostgreSQL-Compatible Edition. This whitepaper outlines security best practices for securing databases running on Aurora MySQL and Amazon RDS for MySQL. The target audience for this whitepaper includes database administrators, enterprise architects, systems administrators, and developers who would like to run their database workloads on Amazon RDS.

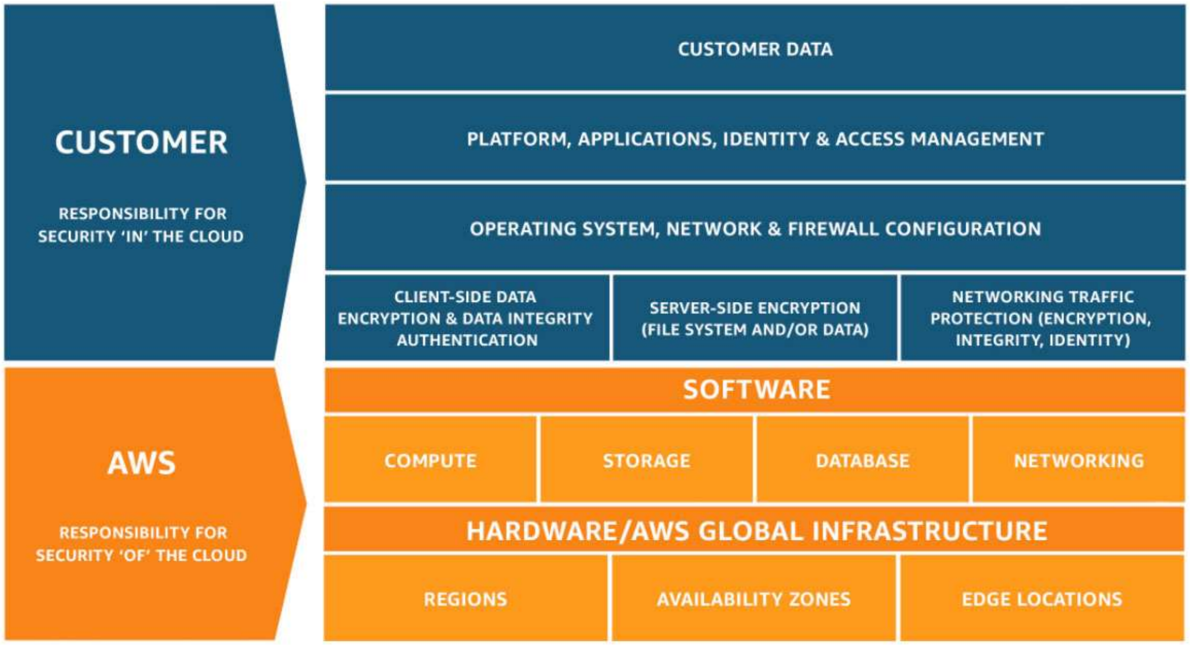
Introduction

Securing databases is crucial for protecting sensitive customer and business data. As organizations increasingly rely on cloud-based databases managed by services like Amazon RDS, it is important to understand best practices for configuring security controls in these database as a service (DBaaS) platforms. While database management services handle many complex security tasks behind the scenes, customers are still responsible for implementing appropriate access controls and encryption to ensure their data is protected.

This whitepaper outlines important security considerations and configuration steps for deploying MySQL and Aurora MySQL databases using RDS. We will discuss the shared responsibility model and how it applies to database security. Key topics covered include networking and access controls, encryption of data at rest and in transit, authentication and authorization methods, auditing and monitoring capabilities. Following the guidance in this whitepaper will help organizations maximize the security of their database workloads managed by RDS.

Shared responsibility model

Security in the cloud alongside the security of the cloud


CUSTOMER
RESPONSIBILITY FOR SECURITY 'IN' THE CLOUD
CUSTOMER DATA
PLATFORM, APPLICATIONS, IDENTITY & ACCESS MANAGEMENT
OPERATING SYSTEM, NETWORK & FIREWALL CONFIGURATION
CLIENT-SIDE DATA ENCRYPTION & DATA INTEGRITY AUTHENTICATION
SERVER-SIDE ENCRYPTION (FILE SYSTEM AND/OR DATA)
NETWORKING TRAFFIC PROTECTION (ENCRYPTION, INTEGRITY, IDENTITY)
AWS
RESPONSIBILITY FOR SECURITY 'OF' THE CLOUD
SOFTWARE
COMPUTE
STORAGE
DATABASE
NETWORKING
HARDWARE/AWS GLOBAL INFRASTRUCTURE
REGIONS
AVAILABILITY ZONES
EDGE LOCATIONS

AWS implements a shared responsibility model with regard to resources in the cloud. In short, AWS is responsible for the security of the cloud, and customers are responsible for security in the cloud.

Security of the cloud – AWS provides secure global facilities and infrastructure that spans regions, availability zones and edge locations. Layered on top of this infrastructure are compute, storage, database, and networking resources that serve as the foundation for every service offered by AWS. From the data centers, all the way up to the software that manages these services, AWS provides a secure cloud on which our customers can build secure and compliant applications.

Security in the cloud – Your responsibility is determined by the AWS service that you use. You are also responsible for other factors including the sensitivity of your data, your organization's requirements, and applicable laws and regulations. AWS provides a set of features and services to help you secure your data. This paper helps you understand how to apply the shared responsibility model when using Amazon RDS.

Infrastructure

AWS Nitro–based instances

The AWS Nitro System breaks apart the traditional role of the hypervisor by providing dedicated hardware and software to control the CPU, storage, networking, bios, and other physical hardware. The net result is that virtually all of the actual server resources go toward running your workload rather than managing a hypervisor.

From a security perspective, the AWS Nitro System provides a specific security chip. This chip is locked down and does not allow administrative or human access, including from Amazon employees. Furthermore, this chip is constantly monitoring the security of the instance hardware and firmware.

For a full list of Amazon EC2 instances that are built on the AWS Nitro System, see Instances built on the AWS Nitro System. The Aurora instance types that use the AWS Nitro System include: T3, T4g, R7g, R6g, R6i, R6gd, and R6id. The RDS instance types that use the AWS Nitro System include: T4g, M7g, M6g, M5, R7g, R6i, R6g, R5, and Z1d.

Most new instance types are AWS Nitro enabled. If you are running your workload on older hardware, we recommend moving to a newer AWS Nitro–enabled instance. There is no additional configuration required to take advantage of the performance advantages AWS Nitro offers.

Internodal encryption

Communication between Aurora instances and Aurora storage is automatically encrypted and requires no additional configuration on the part of the end user.

Identity and access management (IAM)

IAM is the cornerstone of resource management on AWS. IAM defines the access permissions granted to entities to create, modify, and delete resources on AWS. In order to create an RDS instance, one must first have the appropriate IAM permissions to do so.

Permissions within IAM are defined using policies. A policy is a document outlining a certain set of operations (create, modify, delete) that can be applied to certain services or resources. Once a policy is defined, that policy can be attached to an IAM identity (user, group, or role). These identities can be assumed by people directly accessing AWS, or by other resources on AWS. For example, it is common for an EC2 instance to assume a role that contains one or more policies. Let's say that one of those policies allows for writing files to an Amazon S3 bucket but does not allow for reading from the same bucket. In this case, application code running on that EC2 instance can generate log files and write them to the specified S3 bucket, but the application will not have access to read or delete those files. Similarly, let's say that a person has an AWS CloudFormation template that defines an Aurora cluster. In order to provision that template, the user will need to assume a role associated with a policy that grants permissions to create the cluster.

It is of critical importance to secure IAM identities and use restrictive policies. In order to secure IAM identities, it is best practice to only use the AWS account root user to create other users, and then lock away those root credentials. Likewise, when creating policies, it is best practice to only grant very narrow and specific permissions required to accomplish the task at hand. For example, if you wish to create a policy that allows a user to modify a single Aurora cluster, you can restrict access to the specific Amazon Resource Name (ARN) of that cluster. Or perhaps you would like to grant access to create new Aurora clusters, but not to delete them. This can be controlled with a restrictive policy. This is in contrast to attaching the AmazonRDSDataFullAccess built-in policy to a given identity.

Beyond creating, modifying, and deleting database instances and clusters, IAM can also be used to authenticate application users to your Aurora MySQL and RDS for MySQL databases. The general procedure can be summed up as follows:

1. Create or modify an Aurora MySQL cluster or RDS for MySQL instance and set the ***Enable IAM DB Authentication*** parameter to "yes."
2. Create a local database user as follows: ***CREATE USER db_user_name IDENTIFIED WITH AWSAuthenticationPlugin as 'RDS';***
3. Create an IAM policy that specifies connection rights to the user created in step 2. For example:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "rds-db:connect"
      ],
      "Resource": [
"arn:aws:rds-db:region:account-id:dbuser:dbi-resource-id/database-user-name"
      ]
    }
  ]
}
```

region	AWS Region (for example, us-east-2)
account-id	AWS account ID (for example, 123456789012)
dbi-resource-id	Resource ID of the cluster (for example, cluster-OXA6X2XDA225H7PKKDB7FORNAY)
database-user-name	Local database user created in step 2

At this point, the database is configured to authenticate the user specified above using IAM authentication. The next step is to authenticate from your application. This authentication occurs using the SDK of your choice and calling the ***generate-db-auth-token*** method to get a temporary authentication token in lieu of a password. Subsequently, you would use the specified username that you created and the temporary authentication token as the password.

Note that using IAM authentication initially places additional load on your database instance. However, using connection pooling ensures that this increased load is only at application startup.

We recommend using IAM authentication when possible to remove the need for password management. There are, however, caveats to using IAM authentication.

Networking

Network isolation

The main construct available to users to control network access to RDS resources is Amazon Virtual Private Cloud (Amazon VPC). RDS databases exist inside of a VPC. With a VPC, one can establish various subnets and define their connectivity to one another and the outside world. In nearly all circumstances, relational databases should be inaccessible, and they should not have access to network resources outside of the VPC. By adding your RDS databases to private subnets (with no direct route to an internet gateway), devices outside of your VPC do not have direct access to your RDS database. So long as there is no network address translation (NAT) device available to the subnet, your RDS databases do not have direct access to devices outside of your VPC.

When creating an RDS instance, there is an option to specify if the instance should be publicly accessible or not. If you choose that it should be publicly accessible, it will be granted a public IP address. If the instance is also in a publicly accessible subnet within the VPC, the instance will be accessible from the public internet. (However, in order to be given a routable hostname, the VPC must be configured to support DNS.) In nearly all circumstances, however, database instances should not be publicly accessible, and this parameter should be disabled.

VPCs also provide network access control lists (ACLs) to allow you to control network traffic at the subnet level. Traffic can be filtered based on protocol, port, and source. For databases, the default network ACL for a VPC is sufficient. However, there are often other considerations, based on the applications involved and organizational factors. For those unique situations, network ACLs are another filtering tool in your toolbelt.

A more commonly used tool for controlling network access to database resources is security groups. Security groups also allow you to control access at the network level, enabling traffic to be filtered based on protocol, port, and source. However, security groups also have the ability to filter another security group. For example, if you have a fleet of EC2 instances that need to communicate with your RDS instance, you can apply a security group named “ApplicationServers” to those EC2 instances and a second security group named “RdsResource” to your RDS database. To allow the EC2 instances to communicate with the RDS instance, you would add a rule to the RdsResource security group that allows ingress from the ApplicationServers security group. Adding a rule is more convenient than limiting access based solely on source IP ranges and provides more flexibility should underlying IP resources change.

Now you have deployed your RDS resources in a private subnet, inaccessible from external network devices, you need to administratively connect to those RDS resources. In this case, the most common approach is to use a bastion host. A bastion host is a compute resource that has access to both RDS resources and the outside world. In this scenario, although the bastion host is publicly accessible, the security group associated with this host should only allow access from known IP ranges. In turn, that security group will be granted access to the RDS resources that are not publicly accessible. From a user's perspective, one can directly SSH to the bastion host and issue commands from the bastion host. Alternatively, a user can opt to create an SSH tunnel that will allow them to use local applications on their computer and tunnel communication through the bastion host.

VPC flow logs

Once controls are put into place to manage network isolation, the next step is to audit network traffic. The primary tool to accomplish this is VPC flow logs. VPC flow logs identify network traffic flow. Flow logs can be useful in diagnosing overly restrictive network ACLs or security groups, or they can help you uncover gaps in your network controls that are allowing traffic that should be prohibited.

VPC flow logs have no impact on network performance and can be published to Amazon CloudWatch Logs, Amazon Simple Storage Solution (Amazon S3), or Amazon Data Firehose. Each of these destinations provides a different benefit. Amazon S3 is a cost-effective way to capture VPC flow logs as part of an audit trail that might be infrequently accessed. CloudWatch Logs provides a common logging location along with other log types, and it is straightforward to query from the AWS Management Console. Firehose is a great tool for ongoing, near real-time analysis of your VPC flow log records.

VPC endpoints

VPC endpoints provide a mechanism for applications to access service endpoints from within a VPC, rather than transmit data across the public internet. For example, you might run an EC2 instance that has assumed a role that gives it the ability to stop RDS instances in your development environment during off hours. Normally, the API call made to stop the RDS instance would need to travel on the public internet. If your EC2 instance is on a private subnet without a NAT gateway, it will not be able to connect to the RDS API to issue the command to stop the instance. By enabling AWS PrivateLink, your EC2 instance can now issue the stop command without being on a public subnet or using an NAT gateway. Traffic remains on the Amazon network, and neither your EC2 instance nor the relevant RDS instance is exposed to the public internet.

Encryption

AWS KMS

One of the most basic necessities of security when working with databases is to encrypt your data. To encrypt data requires encryption keys, and the management of those keys is a critical consideration when securing your databases. Fortunately, the AWS Key Management Service (AWS KMS) is specifically designed to create, manage, and rotate encryption keys, and it seamlessly integrates with the RDS platform. AWS KMS allows users to create symmetric and asymmetric keys for encryption and decryption purposes as well as HMAC authentication. AWS KMS is not specific to RDS and is integrated with a variety of other AWS services. This integration allows for a unified key management system across AWS. AWS

KMS offers keys that are scoped to a single AWS Region to provide key isolation as well as keys that can be replicated between Regions for seamless cryptographic functionality around the globe.

Encryption at rest

Encryption at rest is straightforward to implement on the RDS platform. Whenever you create an RDS database instance or an Aurora cluster, you need only check the box that indicates that your volume is to be encrypted, and then select the appropriate key to use for that encryption. From that point forward, the RDS platform will encrypt the entire database volume using the specified key. Snapshots and automated backups created from this volume will also be encrypted using the same key. When selecting a key, you can either choose a customer managed key that you have created or the AWS KMS key. Using a customer managed key offers you greater control with regard to key rotation, key material origin, and Region. Additionally, if you are planning to share an encrypted snapshot, it is possible to grant access to a customer managed key, but you cannot grant access to an AWS KMS key.

So far, we have discussed the encryption of on-disk data. If you wish to encrypt data at a more granular level such as row or column, MySQL offers several built-in encryption functions. AES_ENCRYPT and AES_DECRYPT enable symmetric encryption using a shared key. If your application has a secure way of managing this shared key, a shared key can be a suitable solution for cell-level encryption. Alternately, you can use AWS KMS. You can directly use the encrypt and decrypt functions of the AWS KMS SDK. If you decide to use the native MySQL encryption functions (or other application-specific functions), you can use GenerateDataKey to access keys securely stored by AWS KMS.

Secrets Manager and password rotation

Although there are various ways to authenticate to a database instance, perhaps the most common is using a username and password. Often, these credentials are stored unencrypted in code or configuration files. This presents a significant security risk. If your application requires username and password authentication, you should consider storing those credentials in AWS Secrets Manager. Secrets Manager allows you to encrypt these credentials and then access them using an IAM role, as previously discussed in this whitepaper. Encryption ensures that usernames and passwords are never stored in plaintext and are not embedded in application code. AWS provides tutorials on how to use Secrets Manager in your organization.

Auditing and monitoring

CloudTrail integration

AWS CloudTrail allows users to log actions performed in an AWS account. Actions can be initiated by a user, role, or AWS service. The actions logged include management events, including AWS API calls that make modifications to resources within your AWS account, as well as data events such as Amazon S3 object-level activities and AWS Lambda invocations. Regardless of the source (the console, CLI, SDK), these actions are recorded for future auditing, governance, and compliance of your AWS account. CloudTrail is automatically enabled and does not require any manual setup.

CloudTrail is an important part of RDS security, as it provides an audit mechanism for changes made to database resources. If, for example, an RDS instance is created and you need to know who created that instance, that information can be identified using CloudTrail. Additionally, CloudTrail provides the AWS CloudTrail Insights feature that helps you detect anomalies that CloudTrail discovers, alerting you to a potential security risk.

Event notifications

RDS event notification provides a mechanism to invoke an Amazon Simple Notification Service (Amazon SNS) topic when events related to RDS take place. For example, you might create an Amazon SNS topic that sends an email when invoked. You could then tie this SNS topic to an RDS event that is invoked when a database instance has an availability issue such as a shutdown or restart. Alternately, you can use a Lambda SNS endpoint, so that programmatic action takes place in response to the invoked event.

Although there are many RDS events, certain events should be specifically monitored for security., These include items related to credentials such as RDS-EVENT-0016 (reset master credentials), items related to available updates such as RDS-EVENT-0230 (update available), items related to database security groups such as RDS-EVENT-0038 (applied change to security group), and configuration changes such as RDS-EVENT-0092 (finished updating database parameter group).

Configuration

Master user

The master user is the initial user created when any RDS instance is created. This account is the closest login to root that is provided on an RDS instance. It is recommended best practice to provide a strong password for this login, create other users with some subset of the master user's permissions, and then store the master user credentials in a secure location such as Secrets Manager. It is not possible to authenticate as the master user using IAM. It is strongly recommended that the master user not be directly used in any application. This practice helps ensure the principle of least privilege and keeps your database secure.

Parameter groups

RDS parameter groups allow you to create a set of parameters that can be applied to one or many RDS instances of the same database engine type. For example, you might create a parameter group for MySQL 8.0 that can then be applied to all MySQL 8.0 instances in your account in the specified Region. You benefit here from centralized management and the standardization of parameters. Inside of the parameter group, you can configure engine-specific security features. Those features are explored later in this whitepaper.

Deletion protection

For production database instances, it is important to take note of the **enable deletion protection** parameter. By enabling this parameter, the instance or cluster you are creating cannot be deleted unless the parameter is turned off. This provides an additional step to delete RDS resources to help prevent accidental instance deletion.

RDS recommendations

At the bottom of the left navigation bar in the RDS console is a link to RDS recommendations. These are recommendations that will occasionally appear to notify you of best practices for your specific RDS resources. We advise that you periodically review these recommendations and consider implementing them where it makes sense for your applications.

Patch management

Patch management is critical to database security. Over time, new patches are released for any given database engine that can contain performance or stability improvements, but often contain additional security fixes. In the case of RDS, patches are applied at the OS level and the database-engine level. In most cases, it is best practice to enable the automatic minor version upgrades feature. This feature ensures that your database instances are always kept up to date with the latest security patches. In order to minimize disruption to existing workloads, the minor version upgrade happens during your specified maintenance window.

Operating system updates come in two varieties. The first is optional updates. Get notified when an optional operating system update is available by using the RDS event functionality discussed earlier and specifically subscribing to RDS-EVENT-0230.

The second category of operating system updates is the mandatory category. Unlike optional updates, which can be skipped at your discretion, mandatory updates come with an apply date. If you do not apply the update by that date, it will be automatically applied for you during your specified maintenance window.

Protecting data in transit

Secure connections

Both Aurora MySQL and RDS for MySQL support transport layer security (TLS), which allows you to encrypt the communications between your application and the database instance. The specific TLS versions and cipher suites in use can be specified at the parameter-group level using the tls_version and tls_ciphersuites parameters respectively. It is recommended best practice to use either TLS version 1.2 or 1.3. Additionally, the ssl_cipher parameter allows for control of the symmetric algorithms used for encryption. The specific values to use for these parameters will depend on your particular business and regulatory requirements.

For many users, it is a requirement that all connections to the database are encrypted. In this case, you can set the require_secure_transport parameter within the parameter group to ON. Since there is computational overhead required to encrypt and decrypt traffic, the default value for this parameter is OFF.

If your business requirements call for some unencrypted connections but require others to be encrypted, this mix is achievable by specifying the REQUIRE SSL parameter when creating or altering a user within the database. For example: ALTER USER 'encrypted_user'@'%' REQUIRE SSL;

When connecting to the database, the ssl-mode parameter governs whether and how to establish a secure connection. By setting this parameter to DISABLED, no encryption is used. PREFERRED will encrypt the connection if possible, but will fall back to unencrypted if encryption is unavailable. REQUIRED ensures that encryption is used or the connection fails. VERIFY_CA is the same as REQUIRED, but VERIFY_CA has the added requirement of ensuring the server certificate authority (CA) is verified against the configured CA certificates. VERIFY_IDENTITY is the same as VERIFY_CA but with the additional requirement of validating the hostname used by the client with the hostname of the certificate installed on the database instance. If you are using a custom CNAME to point your application to the database instance, this validation will fail, as the hostname specified in the application matches the CNAME and not that of the database instance itself.

Ultimately, encrypting connections will require additional computational overhead, which might result in a small impact to performance. However, it is recommended to encrypt all traffic between your application and database instance if possible.

Certificate authority

A CA identifies the root of a certificate chain and is used to sign other certificates. In the case of Aurora MySQL and RDS for MySQL, the CA is used when creating a certificate on the database instance. The specified CA can be changed using the console, CLI, or SDK. For more detailed information regarding specific CAs offered, see the documentation.

Authentication and authorization

Password management

Creating, altering, and granting permissions to users in Aurora MySQL and RDS for MySQL happens at the engine level and is identical to user management in community MySQL. That said, both Aurora MySQL and RDS for MySQL allow for the configuration of a default password lifetime controlled by the default_password_lifetime parameter in the provided parameter group. A longer lifetime can be specified for a complex password stored in Secrets Manager and used with an application. However, shorter password lifetimes can be required of passwords belonging to individual users, where the risk of exposure might be greater.

RDS for MySQL offers the password validation plugin for RDS for MySQL. This plugin enforces password policies specified by way of the instance parameter group using the password validation parameters. This plugin is helpful to ensure strong passwords for your database instances.

MySQL 8.14 introduced a new feature known as dual passwords. This feature allows for a login to have two passwords at once. Dual passwords are helpful when updating a password across a fleet of application servers or replicas. While the new password is being added, clients will still be able to connect using the old password. Once all clients are updated, remove the old password by using the “DISCARD” parameter of the ALTER USER command.

Kerberos authentication

Customers who currently use Kerberos authentication with Microsoft Active Directory might also wish to use Kerberos authentication with their RDS resources. Aurora MySQL and RDS for MySQL support Kerberos authentication. Kerberos authentication is recommended as a more secure approach to credential management than managing standalone usernames and passwords. Kerberos authentication shifts the management of users and passwords to a centralized Microsoft Active Directory and lifts that burden from the database administrator.

IAM authentication

Applications running on Amazon EC2, Lambda, and other AWS services often assume an IAM role. This role serves as the identity of the calling application and can be used to authenticate the application to your RDS database with IAM authentication. The primary benefit to using IAM authentication to your database is that no password management is required and there are very minimal code changes to the authentication portion for an application that is already using a username and password. IAM authentication allows you to create a user within the database and allocate permissions just as you would any other user. The primary difference is that, instead of specifying a password, you create the user with “IDENTIFIED WITH AWSAuthenticationPlugin as 'RDS';.” To connect to the database, connect as you would with a username and password, but instead of providing a predetermined password, call generate-db-auth-token, which will return a short-lived password token (15 minutes). There are some limitations on the possible number of connection requests in a given time period, and the initial connection call requires more resources than using a static password. IAM authentication is better suited, therefore, for applications that use connection pooling.

Authentication plugins

When using username and password authentication in MySQL 5.7, the default authentication plugin is mysql_native_password. This plugin uses the SHA1 hash algorithm. This is the least secure of the native authentication plugins. The sha256_password plugin uses the SHA256 hash algorithm, which is more secure but computationally expensive. The caching_sha2_password plugin offers the same SHA256 hashing algorithm, but uses a server-side cache for better performance. When possible, it is recommended to use the caching_sha2_password plugin.

When working with IAM authentication, the appropriate plugin to use is AWSAuthenticationPlugin.

Auditing and logging

Once a secure connection has been established between the application and the database server and a user has connected and performed granted activities, the next step is to audit what that user has done.

Aurora MySQL advanced auditing

Depending on the line of business, there are often compliance and regulatory requirements that necessitate auditing database activity. These requirements include, but are not limited to, applications in the financial sector. To support these compliance and regulatory requirements, Aurora MySQL offers an advanced auditing feature that allows users to capture a variety of events and log them to the cluster's storage volume. Advanced auditing is useful for capturing events within your Aurora MySQL cluster including:

- CONNECT – captures connections (success and failure), disconnects, and user information
- QUERY – captures all queries in plaintext
- QUERY_DCL – similar to QUERY, but only DCL commands
- QUERY_DDL – similar to QUERY, but only DDL commands
- QUERY_DML – similar to Query, but only DML commands
- TABLE – captures tables affected by query execution

Capturing this information comes with additional overhead. How much additional overhead depends on which events are being captured and the overall traffic volume.

Your specific business requirements might only require this information to be captured for a subset of users. In that case, you can filter what is captured by using the server_audit_incl_users and server_audit_excl_users parameters. Doing so will also reduce overhead accordingly.

Database Activity Streams

Another mechanism that helps you capture changes made to your databases is Database Activity Streams. Database Activity Streams allows you to stream changes in near real time to Amazon Kinesis Data Streams. The data captured in your Database Activity Stream can be analyzed in, and provide near real-time feedback about, the activity in your database. Furthermore, multiple third-party compliance applications, such as IBM's Security Guardium, can consume and monitor the data in your stream. The data captured in your Database Activity Stream is encrypted using an AWS KMS key that you specify.

MySQL-specific logs

MySQL, and by extension Aurora MySQL, generate other logs that, while not specific to security concerns, can still contain valuable security insights.

The error log contains error information, but it also captures other information such as mysqld startup and shutdown times and various diagnostic messages.

The general query log captures details about what mysqld is doing while running. This includes recording information about connections as well as capturing each SQL statement issued to the database engine. This activity adds certain overhead costs, and so it is disabled by default. This general query log can be useful to temporarily enable when attempting to track down specific activity from a specific source. If you enable this log, be sure to set the log_output parameter to FILE and not TABLE, as TABLE will incur significantly greater overhead.

The slow query log is used primarily to capture slow queries. You can set the long_query_time parameter to a threshold, measured in seconds. Any query that takes longer than that threshold to run will be captured in the log. Setting the long_query_time to 0 will capture all queries sent to the database. Although the slow query log is not specifically a security feature, it provides another mechanism to capture traffic to the database for further analysis.

CloudWatch logging

The audit log, error log, general query log, and slow query log are logged by default on storage local to the database instance. When working with a large number of database instances, it is not practical to connect to each instance to review these logs. Furthermore, processing these logs to find patterns or anomalies is also not a trivial undertaking. Fortunately, RDS offers the ability to publish these logs to CloudWatch Logs. By sending your logs to CloudWatch Logs, you now have a single, centralized repository to view logs. What's more, CloudWatch Logs offers the ability to query your logs through CloudWatch Logs Insights when researching a specific issue and detect anomalies in your logs using log anomaly detection to uncover unknown issues.

Intrusion detection and prevention

Amazon GuardDuty is an AWS service that can monitor CloudTrail event logs, CloudTrail management events, VPC flow logs, and DNS logs in an effort to detect communication with known bad actors and identify anomalous behavior. RDS is one of the many AWS services that GuardDuty supports. Aurora MySQL and Aurora PostgreSQL are also currently supported. Aurora enables GuardDuty to access login activity to your database clusters, identify anomalous behavior, and report the details to you. This is a recommended feature to detect and prevent unauthorized access to your Aurora databases.

Performance Insights

Amazon RDS Performance Insights is a tool primarily designed for capturing and analyzing the performance of your databases. However, anomalous behavior related to performance can be a tip-off that there has been a security incident requiring deeper investigation. RDS Performance Insights can show you the top SQL users and hosts, which might help you identify unanticipated behavior.

Data isolation techniques

While it is important to implement the correct features within a database to keep your data secure, equally important is employing best practices for data isolation. For example, you might have a table for which certain users should only have a filtered view. If the business requirement is that the table can be accessed in the live database, then it might make sense to utilize a view. If the end user can use a snapshot of the data, you might want to consider exporting a database snapshot to Amazon S3. With the database snapshot feature, you can export either a full or a filtered view of a given RDS snapshot to Amazon S3 in Parquet format. Consumers can then run their queries with Amazon Athena, Apache Spark, or any other tool capable of consuming Parquet files.

Conclusion

This whitepaper outlines the most important security best practices for deploying and protecting Aurora MySQL and RDS for MySQL databases. By using RDS security features like encryption at rest, IAM authentication, and advanced auditing, organizations can safely run their critical relational workloads in the cloud.

Contributors

Contributors to this document include:

- Steve Abraham, Principal Database Specialist Solutions Architect, AWS

Further reading

For additional information, refer to:

- AWS Architecture Center
- Security in Amazon Aurora
- Amazon Aurora security
- Security best practices for Amazon Aurora
- Security in Amazon RDS
- MySQL security on Amazon RDS
- Security best practices for Amazon RDS

Document revisions

Date	Description
August 28, 2024	First publication