Oracle® Berkeley DB

Security Guide

Release 18.1
E90768-04
February 2019

Oracle Berkeley DB Security Guide, Release 18.1

E90768-04

Primary Author: Sayantani Ghosh

Contributing Authors: Carol Sandstrom

Contents

Preface

This document describes how you can configure security for Oracle Berkeley DB.

This book is aimed at the developer who wishes to write a secure application which incorporates Oracle Berkeley DB.

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc`.

Accessible Access to Oracle Support

Oracle customers who have purchased support have access to electronic support through My Oracle Support. For information, visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info` or visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs` if you are hearing impaired.

1 Introducing Oracle Berkeley DB Security

Welcome to the Oracle Berkeley DB Security Guide. Berkeley DB is a general-purpose embedded database engine capable of providing a wealth of data management services. It is designed for high-throughput applications requiring in-process, bullet-proof management of critical data. Berkeley DB can gracefully scale from managing a few bytes to literally terabytes of data.

You use Berkeley DB through a series of programming APIs, which give you the ability to read and write your data, manage your database(s), and perform other advanced activities, such as managing transactions. Because Berkeley DB is an embedded database engine, it is extremely fast. You compile and link it to your application in the same way as you would any third-party library. This means that Berkeley DB runs in the same process space as does your application, allowing you to avoid the high cost of interprocess communications incurred by stand-alone database servers.

Be sure to run the `db_verify` utility on any Berkeley DB file provided by another party before doing anything else with it. This also applies if you have any reason to believe that a Berkeley DB file is corrupt, or that it may not be a valid Berkeley DB file. See the Oracle Berkeley DB Standalone Utilities Guide for more information about `db_verify`.

Security within Berkeley DB is achieved primarily through the use of encryption. The encryption support provided with Berkeley DB is intended to protect data from an attacker who has obtained access to the media on which a Berkeley DB database is stored. However, it will not protect applications from attackers who are able to read system memory on the system where Berkeley DB is running. Other measures outside of Berkeley DB (and hence beyond the scope of this document) are required to provide this kind of system security.

Berkeley DB distribution packages including the letters "`NC`" in the package name do not offer encryption. If you need encryption, ensure that you have downloaded the correct package and that you are following all legal requirements for encryption.

Assuming you have a package that allows encryption, cryptography is enabled for Berkeley DB base libraries, and disabled when building the optional Berkeley DB SQL and JDBC libraries. It you want to change the defaults, see the configuration flag "--with-cryptography" in the *Oracle Berkeley DB Installation and Build Guide* for more information.

You can configure security for your Oracle Berkeley DB installations by following the steps detailed in the chapters below.

Related Topics

- --with-cryptography
- Designing Secure Applications Using Berkeley DB
- Designing Secure Highly Available (HA) Applications
- Authenticating Berkeley DB SQLite User
- Configuring Security for Distributed Applications with Berkeley DB

2

Designing Secure Applications Using Berkeley DB

A Berkeley DB environment is an encapsulation of one or more databases, log files, and region files. You can administer a Berkeley DB environment simply by creating a single home directory that stores the files for the applications that will share the environment.

Berkeley DB never creates the environment home directory by itself. You must create the environment home directory before running any Berkeley DB application. You can then identify the environment by the name of that directory. See “The Berkeley DB Environment” in the *Oracle Berkeley DB Programmer's Reference Guide* for more information.

To provide security, you can use the Berkeley DB API to encrypt an entire environment and its contents.

Related Topics

- The Berkeley DB Environment
- Supporting Encryption
- Considering Additional Security

Supporting Encryption

Berkeley DB optionally supports encryption using the Rijndael/AES (also known as the Advanced Encryption Standard and Federal Information Processing Standard (FIPS) 197) algorithm for encryption or decryption.

The algorithm is configured to use a 128-bit key. Berkeley DB uses a 16-byte initialization vector generated using the Mersenne Twister. All encrypted information is additionally checksummed using the Secure Hash Algorithm, which produces a 160-bit message digest.

The encryption support provided with Berkeley DB is intended to protect applications from an attacker obtaining physical access to the media on which a Berkeley DB database is stored. It also protects applications from attackers who compromise a system on which Berkeley DB is running but who are unable to read system or process memory on that system. However, the encryption support provided with Berkeley DB will not protect applications from attackers who are able to read system memory on the system where Berkeley DB is running.

To encrypt a database, you must configure the database for encryption prior to creating it. If you are using a database environment, you must also configure the environment for encryption.

- Follow the steps below to create an encrypted database within an environment:
  1. Configure the environment for encryption using the `DB_ENV->set_encrypt()` method.

2. Open the database environment.
3. Specify the `DB_ENCRYPT` flag to the database handle.
4. Open the database.

- Follow the steps below for databases not created in an environment:
  1. Specify the `DB_ENCRYPT` flag to the database handle.
  2. Call the `DB->set_encrypt()` method.
  3. Open the database.

     Once you complete the configuration steps, all of the databases that you create in the environment are encrypted/decrypted by the password you specify using the `DB_ENV->set_encrypt()` method.

> **Note:**
>
> The only way to encrypt an unencrypted database is to dump the contents with the utility `db_dump` and re-create it in an encrypted form with `db_load`. Also, encrypted databases cannot be read on systems with a different endianness than the system that created the encrypted database.

When a database is encrypted, its log files are also encrypted, so accessing the logs also requires the encryption key. By default, logs are placed in the same directory as the environment. When using the SQL API, the logs are placed in the journal directory. Log files should never be simply deleted. For instructions on how to properly remove log files, see "Log File Removal" in the *Oracle Berkeley DB Programmer's Reference Guide*.

Each encrypted database environment (including all its encrypted databases) is encrypted using a single password and a single algorithm. For applications to have finer granularity of database access they must either use multiple database environments or implement additional access controls outside of Berkeley DB.

The only encrypted parts of a database environment are its databases and its log files. Specifically, the shared memory regions supporting the database environment are not encrypted (see, "Shared Memory Regions", in the *Oracle Berkeley DB Programmer's Reference Guide* for more information). For this reason, it may be possible for an attacker to read some or all of an encrypted database by reading the on-disk files that back these shared memory regions.

There are two ways to prevent such attacks. One is for applications to use in-memory filesystem support (on systems that support it). Another is to use `DB_PRIVATE` or `DB_SYSTEM_MEM` flags in the `DB_ENV->open()` method. These flags ensure that the shared memory regions are placed in memory that is never written to a disk. As some systems back up system memory to a disk, it is important to consider the specific operating system running on the machine as well.

Finally, when backing up database environment shared regions with the filesystem, Berkeley DB can be configured to overwrite the shared regions before removing them by specifying the `DB_OVERWRITE` flag. This option is effective only in the presence of fixed-block filesystems. Journaling or logging filesystems will require operating system support and probably modification of the Berkeley DB sources.

Whereas all user data is encrypted, parts of the databases and log files in an encrypted environment are maintained in an unencrypted state. Specifically, log record headers are not encrypted, only the actual log records are. Additionally, database internal page header fields are not encrypted. These page header fields include information such as the page's `DB_LSN` number and position in the database's sort order.

Log records and database pages distributed by a replication master to replicated clients are transmitted to the clients in unencrypted form. If encryption is desired in a replicated application, the use of a secure transport is strongly suggested and all sites in the replication group must use encryption.

Berkeley DB supports encryption using Intel's Performance Primitive (IPP) on Linux. This works only on Intel processors. To use Berkeley DB with IPP encryption, you must have IPP installed along with a cryptography extension. The IPP performance is higher in most cases compared to the current AES implementation. See "--with-cryptography" in the *Oracle Berkeley DB Installation and Build Guide* for more information. And for more information on "IPP", see *Intel Integrated Performance Primitives – Documentation*.

Related Topics

- Designing Secure Applications Using Berkeley DB
- Log File Removal
- DB_ENV->set_encrypt() Method
- DB_ENCRYPT Flag
- DB->set_encrypt() Method
- DB_PRIVATE Flag
- DB_LSN Handle
- DB_SYSTEM_MEM Flag
- DB_OVERWRITE Flag
- DB_ENV->open() Method
- Shared Memory Regions
- --with-cryptography
- Intel Documentation on IPP

Considering Additional Security

Multiple security issues are taken into consideration when you are writing Berkeley DB applications. The following sections detail these security issues further.

Database Environment Permissions

To ensure that files in the environment are not accessible to unauthorized users, set permissions on the directory used as the Berkeley DB database environment. You must carefully check the applications that add to the user's permissions (for example, `UNIX` `setuid` or `setgid` applications) to prevent illegal use of those permissions. An example of such a permission is general file access in the environment directory.

Environment Variables

Setting the `DB_USE_ENVIRON` and `DB_USE_ENVIRON_ROOT` flags and allowing the use of environment variables during file naming can be dangerous. Setting those flags in Berkeley DB applications with additional permissions (for example, `UNIX setuid` or `setgid` applications) could potentially allow users to read and write databases to which they would not normally have access.

File Permissions

By default, Berkeley DB always creates files that are readable and writable by the owner and the group (that is, `S_IRUSR`, `S_IWUSR`, `S_IRGRP`, and `S_IWGRP`; or octal mode 0660 on historic UNIX systems). The group ownership of created files is based on the system and directory defaults, and is not further specified by Berkeley DB.

Note:

When opening a database on a UNIX/Linux machine, you can specify the file permissions with the mode flag.

Temporary Backing Files

If an unnamed database is created and the cache is too small to hold the database in memory, Berkeley DB will create a temporary physical file to enable it to page the database to disk as needed. In this case, environment variables such as `TMPDIR` may be used to specify the location of that temporary file. Although temporary backing files are created readable and writable by the owner only (`S_IRUSR` and `S_IWUSR`, or octal mode 0600 on historic UNIX systems), some filesystems may not sufficiently protect temporary files created in random directories from improper access. To be absolutely safe, applications storing sensitive data in unnamed databases should use the `DB_ENV->set_tmp_dir()` method to specify a temporary directory with known permissions.

Tcl API

The Berkeley DB Tcl API evaluates user input as Tcl commands. For this reason, it may be dangerous to pass unreviewed user input through the Berkeley DB Tcl API, as the input may subsequently be evaluated as a Tcl command. Additionally, the Berkeley DB Tcl API initialization routine resets effective user and group IDs to the real user and group IDs. This minimizes the effectiveness of a Tcl injection attack.

External Files

External files are not encrypted even when encryption has been specified at the environment or database level. The only way to protect the external files is by using the file system permission restrictions to the directory tree in which the external files are stored.

Private Environments

When using the environment flag `DB_PRIVATE`, it is possible for old data in the heap to leak onto unused parts of the database page. You can avoid this by not using that flag, or by compiling Berkeley DB with the `UMWR` flag.

Related Topics

- Designing Secure Applications Using Berkeley DB
- Supporting Encryption

3

Designing Secure Highly Available (HA) Applications

Berkeley DB includes support for building highly available applications based on replication.

Berkeley DB replication groups consist of a certain number of independently configured database environments. There is a single master database environment and one or more client database environments. Master environments support both database reads and writes; client environments support only database reads. If the master environment fails, applications may upgrade a client to be the new master. The database environments might be on separate computers, on separate hardware partitions in a non-uniform memory access (NUMA) system, or on separate disks in a single server.

As always with Berkeley DB environments, any number of concurrent processes or threads may access a database environment. In the case of a master environment, any number of threads of control may read and write the environment, and in the case of a client environment, any number of threads of control may read the environment.

If encryption is desired in a replicated application, all sites in the replication group must use encryption. For the base replication product, you must design a secure application. The application must provide whatever security policies are needed. The use of a secure transport is strongly recommended because some log and database information is transmitted between sites in unencrypted form. For example, the application may choose to encrypt data, use a secure sockets layer (SSL), or do nothing at all. The level of security is left to the sole discretion of the application.

Related Topics

- Enabling Secure Sockets Layer (SSL) for the Replication Manager

Enabling Secure Sockets Layer (SSL) for the Replication Manager

Berkeley DB uses an OpenSSL software library to provide SSL support for the Replication Manager (see "Replication Manager Overview" in the *Getting Started with Replicated Berkeley DB Applications* guide). This enables secure communication among Replication Manager group members by preventing:

- Data snooping during exchange of data between replication nodes
- Spoofing in a replication group by means of certificate-based authentication

Note:

SSL support in the Replication Manager is not built by default on Windows operating system. On Linux it is built by default only when suitable OpenSSL libraries and headers are present on UNIX. Otherwise, `–with-repmgr-ssl=yes` with suitable values for `LDFLAGS`, `LIBS` and `CFLAGS/CPPFLAGS` are required to build it.

On UNIX, providing `–with-repmgr-ssl=no` as an argument to configure, disables the build of SSL support for Replication Manager.

On Windows, removing the definition of `HAVE_REPMGR_SSL_SUPPORT` from `db_config.h` in `build_windows`, disables SSL.

Once built, SSL support in the Replication Manager is enabled by default. You can, however, disable this by setting the Replication Manager flag `DB_REPMGR_CONF_DISABLE_SSL` using `DB_ENV->rep_set_config()` before starting the Replication Manager.

Features

1. SSL support exists for both Linux and Windows operating systems. However, it is your responsibility to install and ensure availability of OpenSSL header files and libraries while building Berkeley DB. You can do that by following the instructions mentioned in *OpenSSL Installation* document.

   Note that you must use OpenSSL-library versions greater than or equal to 1.0.1 that includes thread support. Unless this condition is met, the build will fail.

2. A replication node without SSL support cannot join a replication group with SSL and vice versa. For more information on building Berkeley DB with SSL support, see the following sections of the *Oracle Berkeley DB Installation and Build Guide* :

   - "Building Secure Sockets Layer (SSL) Support for the Replication Manager" in Chapter "Building Berkeley DB for Windows"of the *Oracle Berkeley DB Installation and Build Guide*
   - "--with-replication-ssl" in Chapter "Building Berkeley DB for UNIX/POSIX" of the *Oracle Berkeley DB Installation and Build Guide*

3. Transport Layer Security (TLSv1.2) protocol is used along with ciphers with the following criteria:

   a. Ciphers should have a key length of 128 bits or more.

   b. Ciphers should not use Secure Hash Algorithm 1 (SHA1).

   c. Ciphers should not use Data Encryption Service (DES) and Triple Data Encryption Service (3DES).

4. You are responsible for creating the certificates and their corresponding keys. The certificates and keys provided by you are used for authentication/verification as well as for establishing a secure SSL connection. Note that, all certificates must be in Privacy Enhanced Mail Computing (.pem) format. It is assumed that private keys and corresponding passwords will not be compromised.

   You can use the `DB_ENV->repmgr_set_ssl_config()` method for specifying the location of the certificates and keys. You use this method to set the value of the following SSL configuration options:

   - DB_REPMGR_SSL_CA_CERT

     Location of CA certificate or CA chain certificate for verification.
   - DB_REPMGR_SSL_CA_DIR

     Location of directory containing all CA /Intermediate CA certificates for verification.
   - DB_REPMGR_SSL_REPNODE_CERT

     Location of certificate presented by this node to peers for authentication.
   - DB_REPMGR_SSL_REPNODE_PRIVATE_KEY

     Location of Private Key corresponding to the RepNode certificate.
   - DB_REPMGR_SSL_REPNODE_KEY_PASSWD

     Password protecting the aforementioned Private Key.
   - DB_REPMGR_SSL_VERIFY_DEPTH

     Number of levels of verification allowed for peer certificate verification.

5. The Replication Manager provides support for authentication of Certificate Authority (CA) signed certificates as well as of Intermediate CA signed certificates. You can specify the location of CA certificates required for the necessary authentication.

Limitations

- The Replication Manager does not allow you to provide multiple files or directories for CA certificate lookup.
- The Replication Manager does not support selective revocation of suspect certificates. In case you suspect that one or more certificates have been compromised, you have to restart the entire replication group with a new set of certificates.
- The Replication Manager does not support automatic renegotiation of session keys after certain number of bytes are transferred or after certain amount of time has elapsed.

Related Topics

- Replication Manager Overview
- with-repmgr-ssl
- Building Secure Sockets Layer (SSL) Support for the Replication Manager
- OpenSSL Installation
- Key Lengths
- Secure Hash Algorithm 1
- Data Encryption Standard
- Triple DES
- Privacy-enhanced Electronic Mail
- DB_REPMGR_CONF_DISABLE_SSL
- API Methods

4

Authenticating Berkeley DB SQLite User

The interaction with the Berkeley DB SQL interface is almost identical to SQLite. You use the same APIs, the same command shell environment, the same SQL statements, and the same PRAGMAs to work with the database created by the Berkeley DB SQL interface as you would if you were using SQLite.

If you are a SQLite user who is using the Berkeley DB SQL interface, see the following:

Related Topics

- Authenticating Berkeley DB SQL Keystore-Based User
- Authenticating Berkeley DB SQL User (without Keystore)
- Miscellaneous Differences

Authenticating Berkeley DB SQL Keystore-Based User

Berkeley DB SQL provides keystore-based user authentication to allow the user to work easily with encryption and user authentication together.

In keystore-based user authentication, encryption becomes mandatory if you enable user authentication. You can then work just with the user authentication API and without the knowledge of an encryption key. You can do this by storing the encryption key into a keystore file.

The encryption key stored in the keystore file is encrypted. The keystore file, name ending with"`.ks`", is placed under the same directory as the database environment. Each authenticated user has one entry in this keystore file. The entry contains the user's name and the encryption key. When `sqlite3_user_authenticate()` is called, if the encryption key is not applied to the database connection yet, Berkeley DB SQL will find the user's entry in the keystore file, compute the database encryption key with the user's password, and then apply the encryption key to the database connection.

To provide security for the Berkeley DB SQL API, turn on keystore-based user authentication. You can enable keystore-based user authentication by adding the `-DBDBSQL_USER_AUTHENTICATION_KEYSTORE` compile option. For further details, read the sections below.

Interface

Keystore user authentication APIs work the same way as non-keystore user authentication. For more information, see *Authenticating Berkeley DB SQL User (without Keystore)*.

Bootstrap

A no-authentication-required database becomes an authentication-required database after the first user is added to the Berkeley DB database. This is called user authentication bootstrap. In bootstrap, you must set the `isAdmin` parameter of the `sqlite3_user_add()` to `true`. After bootstrap, the first added user is logged into the database connection. There are several cases related to the encryption key when doing keystore-based user authentication bootstrap:

- If the database file does not exist yet and the user does not provide his/her encryption key, a random generated key will be applied to the database and be stored into the keystore file. The call sequence is:

  ```
  sqlite3_open_v2();
  sqlite3_user_add();
  ```

  > **Note:**
  >
  > It is recommended to back up the keystore file, especially when using a generated random key.

- If the database file does not exist yet and the user provides the encryption key, the encryption key provided will be applied to the database and be stored into the keystore file. The call sequence is:

  ```
  sqlite3_open_v2();
  sqlite3_key_v2();
  sqlite3_user_add();
  ```

- If the database file already exists, the database is encrypted, and the user provides the correct encryption key, the encryption key provided will be stored into the keystore file. The call sequence is:

  ```
  sqlite3_open_v2();
  sqlite3_key_v2();
  sqlite3_user_add();
  ```

The bootstrap fails if:

- The database file already exists, the database is encrypted, and the user provides an incorrect encryption key.
- The database file already exists but the database is not encrypted.

User Login

As the encryption key is already in the keystore file, you only need to provide the username/password details to work with the encrypted database. Berkeley DB SQL will compute the encryption key from the keystore file and apply it to the database connection. The call sequence is:

```
sqlite3_open_v2();
sqlite3_user_authenticate();
/* Database is now usable */
```

You can also provide the encryption key before the `sqlite3_user_authenticate()` call. In this case, Berkeley DB SQL will not visit the keystore file for the encryption key. The call sequence is:

```
sqlite3_open_v2();
sqlite3_key_v2();
sqlite3_user_authenticate();
/* Database is now usable */
```

Transaction

Berkeley DB user authentication APIs `sqlite3_user_add()/sqlite3_user_change()/sqlite3_user_delete()` work in their own transaction. Calling these APIs inside a transaction results in an error.

The Lock File

Berkeley DB keystore user authentication uses a locking file to ensure it behaves correctly in a multi-threaded environment. In rare cases, if a system or application crash occurs while updating the keystore file, the locking file may not be cleaned and the next `sqlite3_user_authenticate()` call will be rejected. In this case, you need to clean the `.lck` file under the database environment.

Related Topics

- Authenticating Berkeley DB SQL User (without Keystore)

Authenticating Berkeley DB SQL User (without Keystore)

With the Berkeley DB user authentication extension, a database can be marked as requiring authentication.

To view an authentication-required Berkeley DB database, an authenticated user must be logged into the database connection first. Once you mark a Berkeley DB database as authentication-required, it cannot be converted into a no-authentication-required database. Encryption is mandatory if you activate user authentication.

By default a database does not require authentication. You can add the -`DBDBSQL_USER_AUTHENTICATION` compile-time option to activate the Berkeley DB user authentication module.

The client application must add `lang/sql/generated/sqlite3.h` to work with Berkeley DB user authentication. Berkeley DB user authentication is based on SQLite user authentication. For further details, read the sections below.

The Interface

You can use the following three ways to work with Berkeley DB user authentication:

- via C APIs

```
int sqlite3_user_authenticate(
 sqlite3 *db, /* The database connection */
 const char *zUsername, /* Username */
 const char *aPW, /* Password or credentials */
 int nPW /* Number of bytes in aPW[] */
 );
```

```
int sqlite3_user_add(
sqlite3 *db, /* Database connection */
const char *zUsername, /* Username to be added */
const char *aPW, /* Password or credentials */
int nPW, /* Number of bytes in aPW[] */
int isAdmin /* True to give new user admin privilege */
);
int sqlite3_user_change(
sqlite3 *db, /* Database connection */
const char *zUsername, /* Username to change */
const void *aPW, /* Modified password or credentials */
int nPW, /* Number of bytes in aPW[] */
int isAdmin /* Modified admin privilege for the user */
);
int sqlite3_user_delete(
sqlite3 *db, /* Database connection */
const char *zUsername /* Username to remove */
);
```

- Via the Berkeley DB SQL user authentication PRAGMAs below:
  - `PRAGMA bdbsql_user_login="{USER_NAME}:{USER_PWD}";`
  - `PRAGMA bdbsql_user_add="{USER_NAME}:{USER_PWD}:{IS_ADMIN}";`
  - `PRAGMA bdbsql_user_edit="{USER_NAME}:{USER_PWD}:{IS_ADMIN}";`
  - `PRAGMA bdbsql_user_delete="{USER_NAME}";`
- Via the Berkeley DB SQL shell commands as below:
  - `.user login {USER_NAME} {USER_PWD}`
  - `.user add {USER_NAME} {USER_PWD} {IS_ADMIN}`
  - `.user edit {USER_NAME} {USER_PWD} {IS_ADMIN}`
  - `.user delete {USER_NAME}`

You can use the:

- `sqlite3_user_authenticate()` interface to log a user into the database connection. Calling `sqlite3_user_authenticate()` in a no-authentication-required database connection results in an error. This is different from original SQLite behavior.
- `sqlite3_user_add()`/`sqlite3_user_delete()` interfaces to add/delete a user. Calling `sqlite3_user_add()`/`sqlite3_user_delete()` in an authentication-required database connection without an administrative user logged in results in an error. The currently logged-in user cannot be deleted.
- `sqlite3_user_change()` interface to change a user's login credentials or admin privilege. Any user can change their own password, but no user can change their

own administrative privilege setting. Only an administrative user can change another user's login credentials or administrative privilege setting.

Modify the `sqlite3_set_authorizer()` callback to take a 7th parameter, which is the username of the currently logged in user, or NULL for a no-authentication-required database.

When attaching new database files to a connection, each newly attached database (that is an authentication-required database) is checked using the same username and password as provided in the main database. If that check fails, then the `ATTACH` command fails with a `SQLITE_AUTH` error.

Bootstrap

A no-authentication-required database becomes an authentication-required database when the first user is added into the Berkeley DB database. This is called user authentication bootstrap. In bootstrap, you must set the `isAdmin` parameter of the `sqlite3_user_add()` to true. After bootstrap, the first added user is logged into the database connection.

Transaction

Berkeley DB user authentication APIs `sqlite3_user_add()/sqlite3_user_change()/ sqlite3_user_delete()` work in their own transaction. Calling these APIs inside a transaction results in an error.

Security Considerations

A Berkeley DB database is not considered to be secure if it has only Berkeley DB user authentication applied status. The security issues are as follows:

- Anyone with access to the device can just open the database file in binary editor to see and modify the data.
- An authentication-required Berkeley DB database requires no authentication if opened by a version of Berkeley DB that omits the user authentication compile-time option.

Due to these issues, Berkeley DB requires encryption as a prerequisite for user authentication. Call `sqlite3_key_v2()` first with the correct decryption key prior to invoking `sqlite3_user_authenticate()/ sqlite3_user_add()`.

1. To open an existing, encrypted, authentication-required database, the call sequence is:

   ```
   sqlite3_open_v2();
   sqlite3_key_v2();
   sqlite3_user_authenticate();
   /* Database is now usable */
   ```

2. To create a new, encrypted, authentication-required database, the call sequence is:

   ```
   sqlite3_open_v2();
   sqlite3_key_v2();
   sqlite3_user_add();
   ```

Related Topics

- Authenticating Berkeley DB SQL Keystore-Based User

5 Configuring Security for Distributed Applications with Berkeley DB

In addition to being an embedded database, Berkeley DB also supports client-server architecture by providing a stand-alone server program and client driver APIs. The server program offers remote access to DB features.

The client driver APIs provide building blocks for applications that communicate with a database server. Multiple client applications can communicate with a single server simultaneously. For more information, see the *Getting Started with Distributed Berkeley DB Applications* guide.

You can enable Secure Sockets Layer (SSL) to secure communications between clients and servers. Both uni- and bi-directional authentications are supported. Java keystores are used to manage private and public keys for SSL authentication. Depending on whether uni- or bi-directional authentication is used, keystores and/or trust stores must be configured properly on both client and server machines.

Related Topics

- Configuring the Server
- Enabling SSL for the Server
- Connecting to a Server with SSL
- Getting Started with Distributed Berkeley DB Applications

Using Berkeley DB Server Configuration Files

Berkeley DB server uses two configuration files, one for `log4j 2`, and the other for the server itself. After a change to a configuration file, you must restart the server to pick up the change.

log4j 2 Configuration File

Berkeley DB server uses `log4j 2`, and thus requires a `log4j 2` configuration file. Berkeley DB server accepts configuration files in XML only. By default, it looks for a `log4j2.xml` file under the current working directory. The `-log-config` command-line argument overrides this when you start the server.

Here's an example of a `log4j 2` configuration file for a Berkeley DB server:

```
<?xml version="1.0" encoding="UTF-8"?>

<Configuration status="INFO">
 <Properties>
  <Property name="filename">path-to-log-file/logs/db.log<Property/>
 </Properties>
 <Appenders>
  <RollingFile name="RollingFile" fileName="${filename}"
             filePattern="logs/$${date:yyyy-MM}/%d{yyyy-MM-dd}-%i.log.gz">
   <PatternLayout pattern="%d{yyyy-MM-dd 'at' HH:mm:ss z} %p %c{10}
%class{36} %L %M - %msg%xEx%n"/>
   <SizeBasedTriggeringPolicy size="20MB"/>
   <DefaultRolloverStrategy max="20"/>
  </RollingFile>
 </Appenders>

 <Loggers>
  <Root>
   <AppenderRef ref="RollingFile"/>
  </Root>
 </Loggers>
</Configuration>
```

Server Configuration File

Berkeley DB server also uses a Java properties file to control its own behavior. By default, the server looks for a `bdb.properties` file under the current working directory. The `-config-file` command-line argument overrides this when you start the server.

Note:

Because this file may contain sensitive information, its permissions should be limited to the minimum. If the file is accessible globally (globally readable, writable or executable), the server would not start.

Here's an example of a configuration file for a Berkeley DB server:

```
###################################
# Server configuration
###################################
# The server's listening port
port=8080
# If SSL is enabled, this indicates the maximum number of concurrent
# client connections.
#
# If SSL is disabled, there is no limit on the number of concurrent
# connections, and this parameter indicates the maximum number of
# requests that can be processed concurrently.
workers=10
# Configure the host name of this server. SSL is enabled if ssl.host
# is set.
#ssl.host=server-host-name
# Configure the keystore for SSL.
#ssl.keyStore=path-to-key-store
#ssl.keyStore.password=key-store-password
#ssl.keyStore.type=jks
#ssl.keyStore.manager=SunX509
# Configure the trust store for SSL.
```

```
#ssl.trustStore=path-to-trust-store
#ssl.trustStore.password=trust-store-password
#ssl.trustStore.type=jks
#ssl.trustStore.manager=PKIX

###################################
# Database service configurations
###################################
# The root directory for all environments, the home directory for an
# environment with name "<env>" is $root.home/
<env>
root.home=path-to-environment-root-directory
# The root directory for all data directories, the data directory
# for an environment with name "<env>" is $root.data/
<env>
root.data=path-to-data-root-directory
# The root directory for all blob directories, the blob directory
# for an environment with name "<env>" is $root.blob/
<env>
root.blob=path-to-blob-root-directory
# The root directory for all log directories, the log directory for
# an environment with name "<env>" is $root.log/<env>

root.log=path-to-db-log-root-directory
# Configure the timeout value for open handles. The timeout value
# is set in seconds.
#
# For example, if the timeout is set to 600 seconds (10 minutes),
# and a handle has not been accessed for longer than 600 seconds,
# then this handle will be closed automatically when the cleanup
# worker runs the next time.
handle.timeout=600
# The frequency the cleanup worker runs. The frequency is set in
# seconds.
#
# For example, if the frequency is set to 300 seconds, the
# cleanup worker runs every 300 seconds.
cleanup.interval=300
```

Related Topics

- Configuring the Server

Configuring the Server

There are several options available for you to configure the server.

- `port`

  Specifies the port number the server listens for client connections. The default value is `8080`.

- `workers`

  Specifies the number of worker threads used for handling client requests. The default value is `20`.

If SSL is not enabled, client requests are handled by a shared pool of worker threads. A free worker thread is chosen to handle each client request, and the worker thread becomes free again after handling the request. The number of worker threads determines the maximum number of client requests that can be handled concurrently. If the number of concurrent requests exceeds the number of worker threads, later requests will be blocked until a free worker thread is available.

> **Note:**
>
> The number of concurrent client connections is limited only by system resources. For example, the server can serve `200` concurrent client connections even if the number of workers is set to `20`.

If SSL is enabled, each client is handled by a dedicated worker thread. Therefore, the number of worker threads determines the maximum number of concurrent client connections. For example, if workers is set to `100`, the server can serve a maximum of `100` concurrent client connections. If more clients try to connect to the server, they are blocked until some client disconnects from the server or the connection times out (the timeout is specified by the user).

- `ssl.host`

  Specifies the host name of the server. SSL is enabled if this property is specified. Clients should use this name when connecting to the server. There is no default value, and SSL is disabled by default.

- `ssl.keyStore`

  Specifies the path to the Java keystore file that manages the server's private key. There is no default value.

  > **Note:**
  >
  > This file maintains sensitive information. If the file is accessible globally (globally readable, writable, or executable), the server would not start.

- `ssl.keyStore.password`

  Specifies the password of the keystore file specified in `ssl.keyStore`. There is no default value.

- `ssl.keyStore.type`

  Specifies the type of the keystore file specified in `ssl.keyStore`. There is no default value.

- `ssl.keyStore.manager`

  Specifies the algorithm name of the key manager factory. There is no default value.

- `ssl.trustStore`

  Specifies the path to the Java trust store file that manages the server's trusted public certificates. There is no default value.

> **Note:**
>
> This file maintains sensitive information. If the file is accessible globally (globally readable, writable, or executable), the server would not start.

- `ssl.trustStore.password`

  Specifies the password of the trust store file specified in `ssl.trustStore`. There is no default value.
- `ssl.trustStore.type`

  Specifies the type of the trust store file specified in `ssl.trustStore`. There is no default value.
- `ssl.trustStore.manager`

  Specifies the algorithm name of the trust manager factory. There is no default value.
- `root.home`

  Specifies the root directory for all environments' home directories. For example, if the environment home specified by a client is `env_home`, the actual home directory on the server is `/env_home`. The default value is the current working directory.

  > **Note:**
  >
  > The environment home specified by a client cannot escape the specified root directory. Absolute paths or relative paths that escape the root directory are not allowed.
- `root.data`

  Specifies the root directory for all environments' data directories. For example, if the environment home specified by a client is `env_home`, the actual directory for access method database files of this environment is `/env_home`.

  If this property is not specified, `root.home` is used.
- `root.blob`

  Specifies the root directory for all environments' blob directories. For example, if the environment home specified by a client is `env_home`, the actual directory for blob files of this environment is `/env_home`.

  If this property is not specified, `root.home` is used.
- `root.log`

  Specifies the root directory for all environments' log directories. For example, if the environment home specified by a client is `env_home`, the actual directory for log files of this environment is `/env_home`.

  If this property is not specified, `root.home` is used.
- `handle.timeout`

Specifies the timeout for open handles on the server, in seconds. The server scans all open handles periodically, and closes those handles that have not been accessed for the last `handle.timeout` seconds. The default value is `600`.

- `cleanup.interval`

  Specifies the interval between consecutive scans of open handles, in seconds. The default value is `300`.

Related Topics

- Using Berkeley DB Server Configuration Files

Enabling SSL for the Server

To enable SSL for a server, you need to set up the appropriate Java keystore and/or trust store files and then configure the server to use these keystores. If you want to authenticate the server so that clients know that they are connecting to the correct server, set up a keystore with the server's private key on the server. For example, the following command creates a keystore `keystore.jks` containing a generated private/ public key pair.

```
keytool -genkeypair -alias certificatekey -keyalg RSA \
-validity 7 -keystore keystore.jks
```

If you want to authenticate clients, set up a trust store with trusted clients' public keys on the server. For more information, see "Connecting to a Server with SSL".

Once the keystore and/or trust store are set up, you should list them in the server configuration file. For example:

```
ssl.host=localhost

# Configure the keystore for SSL.
ssl.keyStore=keystore.jks
ssl.keyStore.password=<password>
# Configure the trust store for SSL.
#ssl.trustStore=truststore.jks
#ssl.trustStore.password=<password>
```

Related Topics

- Using Berkeley DB Server Configuration Files
- Connecting to a Server with SSL

Connecting to a Server with SSL

To connect to a server with SSL, you need to set up the trust store files and configure the client to use them through `SslConfig`. To continue from the example of authenticating the server found in "Enabling SSL for the Server", to set up the trust store files and connect to a server with SSL, do the following:

1. Export the certificate containing the server's public key from the keystore on the server using the following command:

   ```
   keytool -export -alias certificatekey \
   -keystore keystore.jks -rfc -file cert.cer
   ```

2. Create a trust store on the client machine and import the certificate to it using the following command:

   ```
   keytool -import -alias certificatekey \
   -file cert.cer -keystore truststore.jks
   ```

3. Connect to the server by using `BdbServerConnection.connectSsl` and specifying the host name, port and a `SslConfig`:

   ```
   SslConfig sslConfig = new SslConfig()
   .setTrustStore("truststore.jks", "password");
   BdbServerConnection conn =
   BdbServerConnection.connectSsl("localhost", 8080, sslConfig);
   ```

Related Topics

- Enabling SSL for the Server