SecureSQLHandler - Technical Documentation

Architecture Overview

SecureSQLHandler provides a security layer between your application and database connections. It works by intercepting, obfuscating, and securing SQL queries and their parameters before they are sent to the database engine.

Core Components

TSecureParameter

This class represents a single parameter in a SQL query.

Properties

- `Name`: Parameter name
- `Value`: Parameter value (as Variant)
- `DataType`: Parameter data type (string, integer, float, date, boolean, blob)
- `Encrypted`: Flag indicating if the parameter value is encrypted

Methods

- `Create(AName, AValue, ADataType)`: Constructor
- `AsEncrypted()`: Returns an encrypted string representation of the parameter

TSecureSQLQuery

The main class for executing secure SQL queries.

Properties

- `SQL`: The original SQL query text (getter/setter handles obfuscation)
- `Connection`: Access to the underlying FireDAC connection
- `Query`: Access to the underlying FireDAC query

Methods

- `Create(AConnectionString)`: Constructor that initializes the connection
- `AddParameter(AName, AValue, ADataType)`: Adds a parameter to the query
- `ParameterByName(AName)`: Retrieves a parameter by name
- `Execute()`: Executes a non-select query and returns affected rows

- `Open()`: Executes a select query and returns if data was found

Private Methods

- `ObfuscateSQL(ASQL)`: Transforms SQL to an encrypted form
- `DeobfuscateSQL(AObfuscatedSQL)`: Restores the original SQL
- `EncryptString(AValue)`: Encrypts a string using the instance key
- `DecryptString(AValue)`: Decrypts a string using the instance key
- `ApplyParametersToQuery()`: Applies parameters to the FireDAC query

TSecureSQLConnectionManager

Manages database connections with encrypted connection strings.

Methods

- `AddConnection(AName, AConnectionString)`: Adds and encrypts a connection
- `GetConnection(AName)`: Retrieves a connection by name
- `RemoveConnection(AName)`: Removes and frees a connection
- `EncryptConnectionString(AConnectionString)`: Encrypts a connection string
- `DecryptConnectionString(AEncryptedString)`: Decrypts a connection string

Security Implementation Details

SQL Obfuscation Algorithm

1. **Tokenization**
   - SQL keywords (SELECT, FROM, WHERE, etc.) are replaced with tokens (##SEL##, ##FRM##, etc.)
   - This preserves the structure while hiding the actual SQL commands
2. **Encryption**
   - SQL parts are split and each part (except tokens) is encrypted:
     - XOR operation with a rotating key derived from the instance's encryption key
     - Base64 encoding to ensure valid characters
3. **Integrity Protection**
   - A SHA2 hash of the obfuscated SQL + encryption key is appended
   - Before execution, this hash is verified to detect tampering

Parameter Security

Parameters are secured through:

1. Type-specific encoding based on the parameter's data type
2. Base64 encoding of the resulting value
3. Appending a hash signature for verification

Connection String Protection

Connection strings are protected by:

1. XOR encryption with a key derived from the current date and a seed phrase
2. Base64 encoding of the result
3. Daily key rotation for increased security

Execution Flow

1. **Query Creation**
   - Application creates a TSecureSQLQuery instance
   - Sets the SQL text, which is immediately obfuscated
   - Adds parameters which are stored securely
2. **Query Preparation**
   - When Execute() or Open() is called, the library:
     - Ensures the connection is open
     - Deobfuscates the SQL (internally)
     - Applies parameters to the query
3. **Query Execution**
   - The FireDAC query executes with the deobfuscated SQL
   - Results are returned to the application
   - Errors are caught and sanitized to prevent leaking SQL information

Security Considerations

Strengths

- SQL queries are never stored in plain text in memory
- Parameters are secured and only decrypted when needed
- Connection strings are encrypted
- Integrity verification prevents SQL injection through tampering

Limitations

- The library cannot prevent all forms of memory analysis

- Database server logs may still contain the executed SQL
- Advanced debugging tools might still extract information in some cases

Error Handling

The library provides sanitized error messages that do not expose the actual SQL:

- Original database errors are caught
- Generic error messages are provided to the application
- Internal errors are logged (if configured) without exposing sensitive data

Performance Optimization

To minimize the performance impact:

- SQL obfuscation occurs only once per query string
- Encryption uses fast algorithms (XOR with rotation)
- Connection strings are decrypted only when needed
- Memory usage is optimized for minimal footprint

Integration Guidelines

Integrating with Existing Applications

1. **Replace direct FireDAC usage**:

```pascal
// Before
var
  Query: TFDQuery;
begin
  Query := TFDQuery.Create(nil);
  Query.Connection := MyConnection;
  Query.SQL.Text := 'SELECT * FROM Users WHERE ID = :ID';
  Query.ParamByName('ID').AsInteger := UserID;
  Query.Open;
  // Process results...
  Query.Free;
end;

// After
var
  SecureQuery: TSecureSQLQuery;
begin
  SecureQuery := TSecureSQLQuery.Create(ConnectionString);
  SecureQuery.SQL := 'SELECT * FROM Users WHERE ID = :ID';
  SecureQuery.AddParameter('ID', UserID, pdtInteger);
  SecureQuery.Open;
  // Process results via SecureQuery.Query...
  SecureQuery.Free;
end;
```

2. **Connection Management**:

```pascal
// Store connections securely
ConnectionManager := TSecureSQLConnectionManager.Create;
ConnectionManager.AddConnection('MainDB', ConnectionString);

// Retrieve when needed
Connection := ConnectionManager.GetConnection('MainDB');
```

Example Use Cases

Secure Data Access Layer

```
pascal
```

```
unit SecureDataAccess;

interface

uses
  System.SysUtils, SecureSQLHandler;

type
  TUserRecord = record
    ID: Integer;
    Username: string;
    Email: string;
    IsActive: Boolean;
  end;

  TUserRepository = class
  private
    FConnectionString: string;
  public
    constructor Create(const AConnectionString: string);
    function GetUserByID(const AUserID: Integer): TUserRecord;
    function CreateUser(const AUsername, AEmail: string): Integer;
    function UpdateUser(const AUserRecord: TUserRecord): Boolean;
    function DeleteUser(const AUserID: Integer): Boolean;
  end;

implementation

constructor TUserRepository.Create(const AConnectionString: string);
begin
  FConnectionString := AConnectionString;
end;

function TUserRepository.GetUserByID(const AUserID: Integer): TUserRecord;
var
  SecureQuery: TSecureSQLQuery;
begin
  SecureQuery := TSecureSQLQuery.Create(FConnectionString);
  try
    SecureQuery.SQL := 'SELECT ID, Username, Email, IsActive FROM Users WHERE ID = :UserID';
    SecureQuery.AddParameter('UserID', AUserID, pdtInteger);

    if SecureQuery.Open then
    begin
      Result.ID := SecureQuery.Query.FieldByName('ID').AsInteger;
      Result.Username := SecureQuery.Query.FieldByName('Username').AsString;
      Result.Email := SecureQuery.Query.FieldByName('Email').AsString;
      Result.IsActive := SecureQuery.Query.FieldByName('IsActive').AsBoolean;
    end
    else
    begin
      Result.ID := -1; // Not found
    end;
  finally
    SecureQuery.Free;
  end;
end;

function TUserRepository.CreateUser(const AUsername, AEmail: string): Integer;
var
  SecureQuery: TSecureSQLQuery;
begin
  Result := -1;
  SecureQuery := TSecureSQLQuery.Create(FConnectionString);
  try
    SecureQuery.SQL := 'INSERT INTO Users (Username, Email, IsActive) VALUES (:Username, :Email
                        'SELECT SCOPE_IDENTITY() AS NewID';
    SecureQuery.AddParameter('Username', AUsername, pdtString);
    SecureQuery.AddParameter('Email', AEmail, pdtString);
    SecureQuery.AddParameter('IsActive', True, pdtBoolean);

    if SecureQuery.Open then
      Result := SecureQuery.Query.FieldByName('NewID').AsInteger;
  finally
    SecureQuery.Free;
  end;
end;

function TUserRepository.UpdateUser(const AUserRecord: TUserRecord): Boolean;
var
  SecureQuery: TSecureSQLQuery;
begin
  SecureQuery := TSecureSQLQuery.Create(FConnectionString);
  try
    SecureQuery.SQL := 'UPDATE Users SET Username = :Username, Email = :Email, ' +
                        'IsActive = :IsActive WHERE ID = :ID';
    SecureQuery.AddParameter('ID', AUserRecord.ID, pdtInteger);
    SecureQuery.AddParameter('Username', AUserRecord.Username, pdtString);
    SecureQuery.AddParameter('Email', AUserRecord.Email, pdtString);
    SecureQuery.AddParameter('IsActive', AUserRecord.IsActive, pdtBoolean);

    SecureQuery.Execute;
    Result := SecureQuery.Query.RowsAffected > 0;
  finally
    SecureQuery.Free;
  end;
end;

function TUserRepository.DeleteUser(const AUserID: Integer): Boolean;
var
  SecureQuery: TSecureSQLQuery;
begin
  SecureQuery := TSecureSQLQuery.Create(FConnectionString);
  try
    SecureQuery.SQL := 'DELETE FROM Users WHERE ID = :ID';
    SecureQuery.AddParameter('ID', AUserID, pdtInteger);

    SecureQuery.Execute;
    Result := SecureQuery.Query.RowsAffected > 0;
  finally
    SecureQuery.Free;
  end;
end;

end.
```

Advanced Topics

Custom Encryption Providers

The library can be extended with custom encryption providers by modifying the `EncryptString` and `DecryptString` methods. This allows for integration with hardware security modules or external encryption libraries.

Audit Logging

For regulatory compliance, you may need to add audit logging while maintaining security:

```pascal
// Add to TSecureSQLQuery
procedure LogAuditEvent(const AAction: string);
begin
  // Log the action without exposing the actual SQL
  // Store only:
  // - Action type (SELECT, INSERT, etc.)
  // - Timestamp
  // - User ID
  // - Connection name
  // - Hash of the SQL (for correlation)
end;
```

Troubleshooting

Common Issues

1. **"SQL integrity verification failed"**
   - Cause: The obfuscated SQL was tampered with
   - Solution: Ensure no middleware is modifying the SQL strings
2. **Performance Degradation**
   - Cause: Excessive creation/destruction of query objects
   - Solution: Implement a query pool or cache frequently used queries
3. **Memory Leaks**
   - Cause: Not freeing SecureQuery objects
   - Solution: Always use try/finally blocks around object creation