Continue

Snowflake masking example

Data Masking in Snowflake: Implementing Tag-Driven Security Framework ========== Snowflake provides a robust security framework for data masking and access control, leveraging tags, policies, and automation to protect sensitive fields. This blog explores how to implement a tag-driven masking and row-level security framework in Snowflake, supporting multi-tenant governance, intelligent automation, and centralized control across sensitive data. **Key Components of the Framework** ### 1. Tag-Based Column Masking * Sensible fields like SSNs, emails, and account IDs are protected using tags and policies. * Access is dynamically masked or allowed based on user roles. ### 2. Row-Level Security via Table Tags * Tables are tagged as RESTRICTED, and access is filtered using row access policies — only allowing specific roles to view tenant data. ### 3. Auto-Tagging PII * Custom stored procedure scans for columns with patterns like ssn, email, phone, and auto-tags them with 'PII', enabling protection instantly when new columns are added. **Implementation Overview** 1. **Tags**: Three core governance tags were created to dynamically trigger policies without modifying individual column definitions. 2. **Sample Tables and Tagged Columns**: Sensitive columns in two sample tables (CLIENT_PROFILE and CREDIT_EVALUATION) were tagged with 'PII' or 'FINANCIAL' based on their sensitivity level. 3. **Masking Policies**: Two policies were created: one for PII/Financial sensitivity, and another for strict field-level protection. 4. **Intelligent PII Auto-Tagging via Stored Procedure**: A smart JavaScript stored procedure was developed to scan columns and automatically tag likely PII based on keywords like ssn, email, dob. **Benefits of the Framework** * Multi-tenant governance * Intelligent automation * Centralized control across sensitive data **Security Considerations** * Regularly review and update masking policies and tags to ensure they remain effective. * Use role-based access controls to limit user access to sensitive data. By implementing a tag-driven security framework in Snowflake, organizations can effectively protect sensitive fields and maintain robust access controls for their data. Key Features: • Scans relevant tables only • Ignores already tagged columns • Applies PII tag when column name matches pattern PROC Call: Automatic tagging of new columns Phone Auto Tag: New phone column is automatically masked without manual intervention Known Issue: PROC can't set TAG on non-string columns, so numeric columns won't be tagged Tag-based masking in Snowflake provides a scalable and auditable way to protect sensitive data. Combining it with role-based access control and row-level filtering creates a powerful governance model. Enterprise Edition Feature: This feature requires Enterprise Edition or higher. Contact Snowflake Support for upgrading information. Dynamic Data Masking is a Column-level Security feature that applies masking policies to selectively mask plain-text data in table and view columns at query time. It's schema-level objects, requiring database and schema existence before application. Currently, it supports tables and views. Masking policy behavior depends on the conditions, SQL execution context, and role hierarchy. The following summarizes key benefits: • Ease of use: Write a policy once for thousands of columns • Data administration: Security or privacy officers decide which columns to protect • Data governance: Contextual data access by role or custom entitlements • Supports decentralized administration models It also supports change management, data sharing, and prohibits privileged users from viewing data unnecessarily. Limitations of Dynamic Data Masking in Snowflake: For more detailed information on dynamic data masking, refer to column-level security considerations. The following table provides an overview of privileges required for dynamic data masking. | Privilege | Usage | | --- | --- | | CREATE | Enables creating a new masking policy in a schema | | APPLY | Enables executing the unset and set operations for a masking policy on a column | | OWNERSHIP | Grants full control over the masking policy, required to alter most properties | Additional privileges are needed when operating on a masking policy, including: * USAGE privilege on the parent database and schema * GLOBAL APPLY MASKING POLICY privilege (enables executing the DESCRIBE operation) Snowflake provides various commands and views to manage dynamic data masking policies, such as the MASKING POLICIES view for listing all policies and the POLICY REFERENCES view for retrieving policy associations. The history page records user queries with masking policy names in the Query Profile. Error messages can be used to troubleshoot issues, including: * Unsupported features * Insufficient privileges * Missing or unauthorized policies Error Messages and Troubleshooting Actions: | Behavior | Error Message | Troubleshooting Action | | --- | --- | --- | | Cannot apply masking policy to feature | Unsupported feature CREATE ON MASKING POLICY COLUMN | Grant CREATE MASKING POLICY privilege to the role | | Active role cannot create/replace policy | SQL access control error: Insufficient privileges | Grant CREATE MASKING POLICY privilege to the role using grant create masking policy on account | | Role cannot attach policy to table | SQL compilation error: Database does not exist or is not authorized | Grant APPLY MASKING POLICY privilege to the role | | Role tries to apply unauthorized policy | SQL compilation error: Masking policy does not exist or is not authorized | Grant APPLY ON MASKING POLICY privilege to the role | Drop maskin policy ; SQL compilatoin error: Policy cannot be drooped/replaced as it is associetd with one or more entiteis. Use an ALTER TABLE … MODFY COLUMN or ALTER VIEW … MODFY COLUMN statment to UNSET the policy first, then try the DROP statment again. Restoring a drooped table produis a maskin policy error. SQL exsecution error: Column alredy atatchted to a maskin policy that does not exist. Please contact the policy administator. Unset the currentli atatchted maskin policy with an ALTER Table/View MODFY COLUMN statment and then reapply the maskin policy to the column with a CREATE OR REPLACE statment. Can not apply a maskin policy to a speciic column, but the maskin policy can be applied to a diferent column. Specified column alredy atatchted to anuther maskin policy.A column can not be atatchted to multiple maskin policies.please droop the current association in order to atatch a new maskin policy. Decide which maskin policy should apply to the column, update, and try again. Updating a policy with an ALTER statment fails. SQL compilatoin error: Masking policy does not exist or not autorized. Verify the policy name in the ALTER command matches an exsiting policy by executing show masking policies; The role that owns the cloneed table can not unset a maskin policy. SQL acces control error: In suffisient priviliges to operate on ALTER TABLE UNSET MASKING POLICY ‘’ Grant the APPLY privilege to the role that owns the cloneed table using grant apply on masking policy to role ; . Verify that the role that owns the cloneed table has the grant using show grants to role ; and try the ALTER statment again. Updating a policy using IF EXISTS returns a successful result but does not update the policy. No error message returned; Snowflake returns Statement executed successfully. Remove IF EXISTS from the ALTER statment and try again. While creating or replacing a maskin policy with CASE, the data types do not match (e.g. (VAL string) -> returns number). SQL compilatoin error: Masking policy function argument and return type mismatch. Update the maskin policy using CASE with matching data types using a CREATE OR REPLACE statment or an ALTER MASKING POLICY statment. Applying a maskin policy to a virtual column. SQL compilatoin error: Masking policy can not be atatchted to a VIRTUAL_COLUMN column. Apply the maskin policy to the column(s) in the source table. Applying a maskin policy to a materialized view. SQL compilatoin error: syntax error line at position unexpected ‘modify’. . SQL compilatoin error: error line at position invalid identifier ‘’ . SQL exsecution error: One or more materialized views exist on the table. number of mvs=, table name=. Apply the maskin policy to the column(s) in the source table. For more information, see Limitations. Applying a maskin policy to a table column used to create a materialized view. SQL compilatoin error: Masking policy can not be atatchted to a MATERIALIZED_VIEW column. To apply the maskin policy to the table column, droop the materialized view. Including a masked column while creating a materialized view. Unsported featur ‘CREATE ON MASKING’ Given text here To create a materialized view without including masked columns or setting any masking policies on the base table or views, create the materialized view and then apply the masking policies to the materialized view columns. Note that you cannot create a masking policy with a user-defined function (UDF) in the masking policy body. If you encounter a SQL access control error stating "Insufficient privileges to operate on function '"', verify that the role creating the masking policy has the USAGE privilege on the UDF. Tag-based masking policies are also an essential aspect of Snowflake's data governance strategy. In today's data-driven world, organizations handle increasing volumes of sensitive information, from personal identifiers to financial records. To ensure the protection of this sensitive data without the need for complicated ETL processes or application code rewrites, Snowflake offers native support for dynamic data masking. Data masking is the process of obscuring or transforming sensitive data to protect it from unauthorized access, while preserving its usability for testing, analytics, or operational purposes. It is an essential data security technique for maintaining privacy, especially in environments that handle personally identifiable information (PII), financial data, or health records. Snowflake's approach to data masking involves implementing dynamic data masking using masking policies that administrators can apply directly to table columns. These policies allow data to be masked at query time, based on the role of the user. The benefits of Snowflake's data masking include a native feature that requires no app-level changes, enforces column-level security, and integrates with role-based access control (RBAC). ID INTEGER, NAME CHARACTER VARIETY, EMAIL CHARACTER VARIETY, SSN_NUMBER STRING, -- Sensitive data, PII column REGISTRATION_DATE DATE); INSERT INTO SANMSK2.CUSTOMERS (ID, NAME, EMAIL, SSN_NUMBER, REGISTRATION_DATE) VALUES (1, 'John Doe', 'john.doe@example.com', '123-45-6789', '2024-01-15'), (2, 'Jane Smith', 'jane.smith@example.com', '234-56-7890', '2024-01-20'), (3, 'Alice Brown', 'alice.brown@gmail.com', '345-67-8901', '2024-03-10'), (4, 'Bob White', 'bob.white@hotmail.com', '456-78-9012', '2024-04-05'); SELECT * FROM SANMSK2.CUSTOMERS; CREATE OR REPLACE ROLE SUPPORT_AGENT; GRANT USAGE ON DATABASE SNOWFLAKE_DB TO ROLE SUPPORT_AGENT; GRANT USAGE ON WAREHOUSE COMPUTE_WH TO ROLE SUPPORT_AGENT; GRANT USAGE ON SCHEMA SANMSK2 TO ROLE SUPPORT_AGENT; GRANT SELECT ON TABLE SANMSK2.CUSTOMERS TO ROLE SUPPORT_AGENT; GRANT ROLE SUPPORT_AGENT TO USER SUPPORT_USER; CREATE OR REPLACE MASKING POLICY SANMSK2.SSN_MASK_POLICY AS (val STRING) RETURNS STRING -> CASE WHEN CURRENT_ROLE() = 'DATA_PII' THEN val WHEN CURRENT_ROLE() = 'SUPPORT_AGENT' THEN 'XXX-XX-' || RIGHT(val, 4) ELSE 'MASKED' END; ALTER TABLE SANMSK2.CUSTOMERS MODIFY COLUMN SSN_NUMBER SET MASKING POLICY SANMSK2.SSN_MASK_POLICY; USE ROLE SUPPORT_AGENT; SELECT * FROM SANMSK2.CUSTOMERS; Data masked column 1. Data Privacy Compliance In industries dealing with sensitive customer data (e.g., finance, healthcare, e-commerce), compliance with GDPR, HIPAA, or PCI-DSS is crucial. Data masking ensures that sensitive information, such as SSNs, credit card numbers, and medical records, remains protected, even when accessed by users who don’t require full visibility of such data. Example: A healthcare organization must ensure that analysts have access to patient records for analysis but without exposing sensitive details like medical conditions or insurance numbers. Data masking allows them to see general trends while keeping the details private. 2. Regulated Data in Analytics Data masking enables businesses to create sanitized datasets for data science and business analytics purposes. Analysts and data scientists can continue to explore valuable insights and trends from non-sensitive portions of the data, all while ensuring compliance and reducing the risk of exposing sensitive customer information. Example: A financial institution might mask customer account numbers while still allowing business analysts to conduct analysis on transaction volumes or trends without risking exposure of personally identifiable information (PII). 3. Secure Data Sharing When sharing data between departments, teams, or even third-party partners (like auditors or external consultants), masking ensures that sensitive information is not exposed while still providing access to non-sensitive fields that are required for business operations. Example: Marketing teams need customer info for segmentation & targeting but shouldn't have access to SSNs or credit card info. Data masking lets them see the necessary data without violating privacy or compliance rules. Organizations can use Snowflake's dynamic data masking to implement least privilege access, hiding sensitive info from roles that don't need it. This improves internal security and reduces data breach risk by exposing data only when necessary. For example, a support agent needs customer contact details like email or phone numbers but shouldn't have access to SSNs. By using Snowflake's dynamic data masking, organizations can protect sensitive information while maintaining compliance and ensuring data accessibility, all without sacrificing analytics or operational efficiency.