SESUG Paper 157-2019

CONNECT TO vs. CONNECT USING for Security in SAS® PROC SQL

Thomas E. Billings, MUFG Union Bank, N.A., San Francisco, California

ABSTRACT

We begin with an overview of the main SQL dialects in the SAS® system, and then describe the 2 modes of operation of PROC SQL: explicit pass-through and implicit pass-through. We focus on explicit pass-through, where PROC SQL is a wrapper to pass user-written, native RDBMS SQL code to a remote system. CONNECT TO syntax is illustrated, including the common usage of database passwords in source code. Methods to mitigate this issue are discussed – PROC PWENCODE, blotting, and encrypted, compiled macros that are user-specific. Next we illustrate an alternative, CONNECT USING paired with authentication domain LIBNAMEs. Here the password and userid are hidden in metadata (using SAS Personal Login Manager), providing a higher security alternative. We end with brief comments on explicit vs. implicit pass-through, efficiency vs. portability and maintainability.

Keywords: security, PROC SQL, CONNECT, explicit pass-through, authentication domain, LIBNAME, PROC PWENCODE.
SAS products: Base SAS, SAS/SECURE, SAS Personal Login Manager.
User level: intermediate+.

BACKGROUND

The SAS® system provides 2 native SQL dialects:

- PROC SQL
- PROC FEDSQL

Both procedures are supported in Base SAS and SAS Viya®. However - in SAS Viya, PROC FEDSQL is limited to base SAS tables and CAS files. The Base SAS system includes support for 2 relevant non-SAS languages: Groovy (PROC GROOVY) and Lua (PROC LUA). Both of these languages have SQL-database interfaces, though you may need admin privileges (or admin assistance) to setup an initial interface. Both R and Python can interface with base SAS and SAS Viya; they both have numerous SQL-database interfaces.

Some of the main differences between PROC SQL and PROC FEDSQL are:

- PROC SQL supports only 2 variable types, floating point and character while PROC FEDSQL supports a long list of variable types per the ANSI standard;
- PROC FEDSQL is threaded, scalable, and can handle big data better than PROC SQL; this is a good choice for Grid and/or CAS systems
- PROC SQL supports many SAS functions that are not supported in PROC FEDSQL.

Readers interested in PROC FEDSQL are encouraged to read the extensive online documentation for the procedure. PROC SQL supports 2 processing modes:

- Explicit pass-through of native (non-SAS) SQL dialects to the target RDBMS systems, e.g., PROC SQL serves as a “wrapper” to pass Oracle SQL directly to a connected Oracle data base;

- Implicit pass-through: a user writes native SAS SQL code and PROC SQL analyzes the code then divides it into logic that can be run on the SAS server and logic that must be exported to the connected RDBMS. In this situation, PROC SQL generates the SQL sent to the RDBMS.

Code intended for explicit pass-through should be in the SQL dialect of the target RDBMS. Code for implicit pass-through is in the native SAS PROC SQL dialect and the SAS system translates that code into the SQL dialect(s) used by the target RDBMS; the generated code is passed to the connected database(s) and results are returned to SAS. Implicit PROC SQL code that operates only on SAS data sets runs only on SAS servers as there are no external RDBMS systems.

The SAS system allows users to operate on RDBMS files (and other types of files, e.g., XML) as if they were SAS data sets in the DATA step and PROCs. This is done by defining the RDBMS link via a LIBNAME.

Our primary focus here is on PROC SQL explicit pass-through (implicit will be discussed very briefly, later). We begin with PROC SQL CONNECT TO syntax.

Note: some of the text above is repurposed from Billings (2018B).

CONNECT TO SYNTAX

For explicit pass-through of user-written (or user-supplied) native, non-SAS RDBMS SQL dialect code, PROC SQL uses 2 versions of the CONNECT statement to identify the target database and initiate a connection. CONNECT TO syntax is the most common form of this statement.

An example of CONNECT TO syntax to create a SAS dataset from an Oracle source is as follows; annotations are designated by [#]:

```
proc sql …options here…;     [1]
connect to oracle(user=&userid. [2], [3]
       pw=_password_here_   [3]
       path=server_code
       …other options here…);   [4]
create table my_staging as                 [5]
       select * from connection to oracle  [5]
              (  [6]  Begin Oracle SQL
       select
              t1.variable_1,
              t1.variable_2,
              …snipped…
              t1.variable_n
       from
              schema_name.staging_table_name t1  [7]
       where
              … list of conditions, joined via and/or …

       order by t1.variable_1, t1.variable_2
              );  [6]  End Oracle SQL
disconnect from oracle;  [2]
 quit;  [1]
```

Notes:

[1] PROC SQL statement begins the code and QUIT; ends the code. Note this is QUIT; and not RUN;
[2] CONNECT TO database_name statement is paired with DISCONNECT FROM statement

[3] Userid and password are in source code here, and will show up in the log unless mitigation steps are taken (discussed below). At many sites this is considered a security issue (at least for production databases that contain confidential data).
[4] Note the) ; that closes this statement
[5] This statement directs the SAS system to create a SAS dataset, my_staging, from the results of the query.
[5,6] Note the structure: SAS CREATE TABLE … from a native Oracle SQL query (denoted by (..) with [6] above. Functionally the native Oracle SQL query is treated as a subquery of the SAS PROC SQL query, i.e., a "wrapper".
[7] The schema and table name on this line are an Oracle schema and Oracle table name.

The example above is for an external database (Oracle) query, captured in SAS. For non-query SQL, the code is slightly simpler:

```
proc sql …options here…;      [1]
connect to oracle(user=&userid. [2], [3]
       pw=_password_here_    [3]
       path=server_code
       …other options here…);    [4]
        Oracle commands here (e.g., request metadata, execute, etc.)
disconnect from oracle;   [2]
 quit;  [1][8]
```

Security. As noted above, unless mitigating measures are applied, the explicit pass-through code above will display the database password:

```
proc sql …options here…;      [1]
connect to oracle(user=&userid. [2], [3]
       pw=_password_here_    [3]
```

Passwords appearing in open code is a security issue and should be avoided. In the context of CONNECT TO syntax, the SAS system provides two mitigations, as follows.

PROC PWENCODE will take a password and create an equivalent, encoded version that SAS can decode at runtime. This keeps the actual password out of code, but the encoded version is a proxy for the password – in the SAS system – so is weak security. Examples of its usage are as follows.

To create an encoded password, use this code:

```
proc pwencode in='super_secret_value' method=sas003;   [1]
run;

%put &_pwencode.; [2]
%let _pwencode=;  [3]
%put &_pwencode.; [3]
```

[1] There are 5 encoding methods, sas001 to sas005. Methods sas003 to sas005 are available only at sites that license the SAS/SECURE product. Method sas003 conforms to the Federal Information Processing Standardization 140 (FIPS 140-2). For information on SAS compliance with FIPS, see:

https://documentation.sas.com/?cdcId=pgmsascdc&cdcVersion=9.4_3.2&docsetId=secref&docsetTarget=n0pkwquwl0843kn1lroueqj54x45.htm&locale=en

[2] The encoded string is stored in the global macro variable &_PWENCODE.
[3] If you will not use the macro variable, then it is a good idea to clear it as shown above.

When the above code is run we get (in the saslog):

```
24          proc pwencode in=XXXXXXXXXXXXXXXXXXXX method=sas003; [5]
25          run;


{SAS003}9142106A07892CA4D0A07CF7C810FC57E54CEE1EE0E5F7F6916B32F11EC2CCB50AA4  [6]

NOTE: PROCEDURE PWENCODE used (Total process time):
… snipped …

26
27          %put &_pwencode.;
{SAS003}9142106A07892CA4D0A07CF7C810FC57E54CEE1EE0E5F7F6916B32F11EC2CCB50AA4 [7]
28          %let _pwencode=;   [8]
29          %put &_pwencode.;  [8]

30
```

[5] Note that the actual password does not appear in the log; it is blotted out, the term SAS uses to refer to masking.
[6] This is the encoded version of the password
[7] Confirms that the encoded string is stored in macro variable &_PWENCODE
[8] Confirms that the macro variable can be cleared by a user (some global macro variables are protected and cannot be changed by users).

Then to use the encoded string, use it as the operand in the password= or pw= parameter:

```
 proc sql …options here…;
 connect to oracle(user=&userid.
 pw="{SAS003}9142106A07892CA4D0A07CF7C810FC57E54CEE1EE0E5F7F6916B32F11EC2CCB50AA4"
```

Double quotes are used above to encapsulate the password; single quotes can be used and quotes can be omitted in some cases. Encoded strings can be used for passwords but not for userids, and not for encryption keys.

Here is an edited example from the log from an actual run using an encoded password:

```
24        !  proc sql inobs=max;
25             connect to oracle(user="redacted"
26                    pw=XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
27                    path=redacted);
```

Blotting. Blotting refers to the masking of passwords in the saslog. To maximize the probability that the password will be blotted, structure your code so the phrase in your code:

```
password = value  OR pw=value
```

appears by itself, on a separate line (as in the examples above). The SAS documentation reports that if the password=value phrase is inside a macro it won't be blotted; this writer's experience is that (inside a macro) sometimes it is blotted, and sometimes it is not blotted. The relevant SAS 9.4 documentation is at:

https://documentation.sas.com/?docsetId=lrcon&docsetTarget=n0f79bcfsnsl82n117dahzoqrsdn.htm&docsetVersion=9.4&locale=en

The combination of PROC PWENCODE and blotting keeps the actual password out of the code and log, but the encoded string is a proxy for the password.

Encrypted, compiled function-style macros. Billings (2017A, 2018A) describes relatively simple, compiled and encrypted function-style SAS macros that return a password (or encryption key). The macros work (yield a password) only for a single, specific user, and they mask the password in the log in open code, and sometimes (but not all instances) when it is embedded inside a macro. These macros can be used in conjunction with PROC PWENCODE and blotting; having multiple layers of security is a good practice. After creating such a macro, use it as follows:

```
proc sql …options here…;
      connect to oracle(user=&userid.
            pw=%my_pswd
```

Function-style macros can also be used to hide userids, encryption keys, and other sensitive strings.

CONNECT USING CAN BE MORE SECURE

CONNECT USING syntax uses a libref (for an associated LIBNAME) for the login credentials vs. CONNECT TO where the credentials are embedded in the source code for the CONNECT TO statement. Some readers will immediately notice that this may simply shift the location of the login credentials from the PROC SQL code to the relevant LIBNAME statement IF one uses a LIBNAME like the following:

```
LIBNAME libref ORACLE  PATH="server_ID"  SCHEMA="my_schema"
              USER="my_userid" PASSWORD="my_secret" ;
```

The password and userid appear in the statement above, the same problem as when they appear in PROC SQL CONNECT TO code. The mitigations previously described can be used in the statement above; additionally the alternative methods described in Billings (2017B) can be used to issue LIBNAMEs in a secure manner. However, there is an easier and secure alternative: use authentication domain LIBNAMEs.

Authentication Domains for data source access.** This is a metadata-based method that matches login credentials with data sources. To summarize the process:

- The target data sources must be registered in metadata and an authentication domain name assigned to the source; your SAS Admins can provide this service.
- Users who wish to use this access method must have a functional userid and password for the target database/ODBC interface.
- Users enter the relevant userid and password for the target system and named authentication domain in the SAS Personal Login Manager tool.
- To access the data source, use a LIBNAME statement that specifies the named authentication domain (AUTHDOMAIN= option); example below. This method keeps userids and passwords out of source code and logs.
- The SAS system will retrieve the login credentials from secure metadata and use them for login to the target database.

Example:

```
libname mylib  oracle path=abcd schema=myschm authdomain=auth_label;
```

Authentication domains let users manage the userids and passwords required to access data sources, and provide a more granular level of database access security and control (when used in conjunction with IT management of database access). For the SAS (9.4) documentation on authentication domains, start with this URL:

- https://documentation.sas.com/?docsetId=acreldb&docsetTarget=n0aiq25zc8u8u6n1i81my0a24sd3.htm&docsetVersion=9.4&locale=en

Hemedinger (2010) discusses additional methods for keeping passwords and userids out of code and logs.

** Note: This section was adapted and updated from Billings (2018A).

Given an authentication domain LIBNAME, it can be used with CONNECT USING syntax as illustrated in the following example. First, issue the authentication domain LIBNAME; if successful, the log messages will look like the below, with *'s replaced by the relevant actual parameters:

```
NOTE:  Credential obtained from SAS metadata server.
NOTE: Libref ******* was successfully assigned as follows:
      Engine:        ORACLE
      Physical Name: *****
```

The PROC SQL code to capture (in SAS) the results of an external DMS query will look like the following example. Note that the main difference is CONNECT TO statement is replaced by CONNECT USING. Connect is in red (and highlighted) below because SAS Enterprise Guide shows it as that color even though it is correct syntax (SAS Enterprise Guide does this on occasion, i.e., flag as red code that is actually correct syntax.):

```
proc sql …options here…;
connect using libref as oracle;
create table my_staging as
       select * from connection to oracle
              (
       select
              t1.variable_1,
              t1.variable_2,
              …snipped…
              t1.variable_n
       from
              schema_name.staging_table_name t1
       where
              … list of conditions, joined via and/or …

       order by t1.variable_1, t1.variable_2
              );
disconnect from oracle;
 quit;
```

The non-query CONNECT USING case has changes (vs. CONNECT TO) similar to the above. Because CONNECT USING hides the login credentials in metadata, it is higher security than the other methods – and easier than creating macros – and it is recommended over CONNECT TO.

EXPLICIT vs. IMPLICIT PASS-THROUGH – EFFICIENCY vs. PORTABILITY

There are numerous SAS User Group papers that discuss explicit and implicit PROC SQL pass-through. The question of which is better is the subject of lively debate, and the answer is frequently context-specific. Explicit pass-through is often considered to be more efficient as it can minimize data exchange between SAS and the remote database; however implicit pass-through code is portable and can be easier to maintain. Readers interested in writing optimal SQL code can start their research with this SAS documentation:

https://documentation.sas.com/?activeCdc=pgmsascdc&cdcId=sasstudiocdc&cdcVersion=4.4&docsetId=acreldb&docsetTarget=p1f9ovbl1ifskpn1e82nky8v5bbb.htm&locale=en

SUMMARY

CONNECT TO syntax is widely used for explicit pass-through of native (non-SAS) RDBMS SQL code to external databases. However this method requires the embedding of login parameters in the PROC SQL code. There are mitigations available for this issue – PROC PWENCODE, blotting, and user-specific encrypted, compiled, function-style macros.

CONNECT USING uses an associated LIBNAME, and if that LIBNAME is an authentication domain LIBNAME, then the userid and passwords are hidden in metadata, providing a higher security approach.

Recommendation: use CONNECT USING with authentication domain LIBNAMEs instead of CONNECT TO in PROC SQL.

APPENDIX 1:
BSD 2-CLAUSE COPYRIGHT LICENSE (OPEN SOURCE)

```
* All program code in this paper is released under a Berkeley Systems
  Distribution BSD-2-Clause license, an open-source license that permits
  free reuse and republication under conditions;

/*
Copyright (c) 2019, MUFG Union Bank, N.A.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice,
this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice,
this list of conditions and the following disclaimer in the documentation
and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE
LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
POSSIBILITY OF SUCH DAMAGE.
*/
```

REFERENCES

Note: all URLs quoted or cited herein were accessed in March 2019.

Billings T (2017A). Secure Macro-Based Method to Assign LIBNAMEs for Databases. Paper to be presented at 2017 *Western Users of SAS Software Conference Proceedings.* URL: https://www.lexjansen.com/wuss/2017/22_Final_Paper_PDF.pdf

Billings T (2017B). Keeping Passwords, AES Encryption Keys, and Other Sensitive Parameters Out of Source Code and Logs. *Western Users of SAS Software Conference Proceedings.* URL: https://www.lexjansen.com/wuss/2017/21_Final_Paper_PDF.pdf

Billings T (2018A). Non-metadata Methods to Keep Passwords and Sensitive Strings out of SAS® Source Code and Logs. *SAS Global Forum Conference Proceedings.* URL: https://www.sas.com/content/dam/SAS/support/en/sas-global-forum-proceedings/2018/1736-2018.pdf

Billings T (2018B). Emulating FIRST. and LAST. SAS® DATA Step Processing in SQL? Concepts and Review. *Southeast SAS Users Group Conference Proceedings.* URL: https://www.lexjansen.com/sesug/2018/SESUG2018_Paper-192_Final_PDF.pdf

Hemedinger C (2010). Five strategies to eliminate passwords from your SAS programs. *SAS Blog: The SAS Dummy.* URL: http://blogs.sas.com/content/sasdummy/2010/11/23/five-strategies-to-eliminate-passwords-from-your-sas-programs/

ACKNOWLEDGEMENTS

Thanks to Thomas Kunselman of MUFG Union Bank, N.A.; his inquiry was the stimulus for writing this paper.

Any errors herein are solely the responsibility of the author.

CONTACT INFORMATION

A list of the author’s SAS-related papers, including URLs for free access, is available at the URL (hosted by Google Drive): https://goo.gl/uCUHoa

Note: Your enterprise web filter might prevent access to this URL from work, in which case you will need to access via a personal device.

Thomas E. Billings
MUFG Union Bank, N.A.
San Francisco, CA 94104

Remote from:
Merritt Island, FL 32952
Email: tebillings@gmail.com

Disclaimer: The contents of the paper herein are solely the author’s thoughts and opinions, which do not represent those of MUFG Union Bank N.A. The bank does not endorse, recommend, or promote any of the computing architectures, platforms, software, programming techniques or styles referenced in this paper.