Paper #1736-2018

Non-Metadata Methods to Keep Passwords and Sensitive Strings out of SAS® Source Code and Logs

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

ABSTRACT

The SAS® system provides metadata-based methods to keep most passwords and encryption keys out of source code and logs. Here we provide a brief overview of these methods and then describe alternative methods that do not require or depend on SAS system-provided metadata features. The first method is the simplest: function-style SAS macros that supply sensitive/confidential strings. For security, these macros must be written in a certain way and the executable file for the compiled macro must be vetted to make sure that no sensitive data are visible. This method is relatively secure when used in open code, but may not be secure if the invocation of the compiled macro is inside a wrapper macro with MPRINT enabled. The second method is an extension of work by Sherman and Carpenter (2009): a secure, compiled, encrypted macro that assigns LIBNAMEs to relational databases and ODBC. The macro uses an encrypted SAS data set that contains the login parameters. The user of the macro does not know, and cannot acquire, the userid & password for the target database. This is a very complex macro with extensive error checking. A table that summarizes the various metadata and non-metadata approaches is also provided herein. SAS products: Base SAS, SAS Business Intelligence products, SAS Management Console. User level: intermediate & advanced.

MOTIVATION: SECURITY

SAS® code and/or logs can potentially contain sensitive information in a number of contexts and applications:

- Passwords and userids in LIBNAMEs for relational databases and ODBC
- Passwords and userids in PROC SQL CONNECT TO syntax for relational databases and ODBC
- Encryption keys for SAS data sets: AES encryption and/or SAS-proprietary encryption keys
- Passwords and userids in LIBNAMEs for PCFILES server, EXCEL, and select other data engines
- Passwords and userids in FILENAMEs for select external data sources
- PDF file passwords (created using ODS), and
- other applications.

The presence of sensitive or confidential information like passwords, encryption keys, and (in some contexts/applications) userids in production source code/logs is a security risk and in many enterprises is a security violation. If auditors find sensitive information in your code or logs, it can produce unfavorable audit findings. Similar remarks may apply to systems that are subject to external/regulatory review for security.

Unfortunately, some programmers still include passwords in their SAS source code because it is the fastest solution to an immediate problem and/or they are not aware of the alternatives available. Programmers working in non-metadata environments have fewer options (compared to metadata environments) to protect passwords in their code and logs.

In the sections that follow we provide a brief overview of the relevant metadata-based options to hide passwords and encryption keys, describe alternative non-metadata methods to accomplish the same purpose, and end with a table that summarizes the applications and options available.

SAS SYSTEM: METADATA-BASED METHODS

The SAS system provides a number of metadata based methods to keep passwords, userids, and encryption keys out of source code and logs.

Metadata-bound libraries to hide encryption keys. A library (directory that contains SAS files) can be defined as metadata-bound and encrypted using SAS-proprietary encryption or, in SAS 9.4+, the stronger AES-encryption. Metadata-bound libraries enforce metadata permissions and security on a physical library and the files it contains, covering all the access methods supported in SAS. This feature enhances the security of data in the SAS system.

In most applications, metadata-bound libraries are the optimal way to manage encrypted libraries. When an encrypted, metadata-bound library is accessed by an authorized user, the SAS system verifies that the user is allowed to access the file and automatically supplies the key needed to decrypt or encrypt the target file. For libraries that are not metadata-bound, and for all encrypted file/library access in non-metadata environments, the user must supply the encryption key on all/nearly all accesses.

Non-metadata methods to keep encryption keys out of the source code/logs are presented later herein, but readers are cautioned that supplying encryption keys on every access can be a real hassle. This especially applies when one is accessing encrypted files (that are not metadata-bound) interactively using SAS Enterprise Guide, SAS Studio, and other tools: you have to manually enter the encryption key every time you open the file to look at the data. Keyboard macros can make this easier, but that may violate security rules at some sites. In contrast, in these same applications the user is not prompted for an encryption key for metadata-bound libraries that the user is authorized to access; instead, the key is automatically supplied by the SAS system.

Metadata-bound libraries are the most user-friendly approach to hide encryption keys, and non-metadata methods are suggested only for non-metadata environments and/or special high-security applications in metadata environments. Metadata -bound libraries can be created:

- using PROC AUTHLIB,
- using SAS Management Console; ask your SAS Admins to setup the libraries.

For more information on this topic, refer to: *SAS® 9.4 Guide to Metadata-Bound Libraries;* URL:

- http://support.sas.com/documentation/cdl/en/seclibag/66930/HTML/default/viewer.htm#titlepage.htm

Meta-LIBNAMEs for data source access. A meta-LIBNAME is a LIBNAME to a data source that is defined in metadata, with the associated SAS-user accesses also managed in metadata. It is possible to hide userids and passwords in metadata by using meta-LIBNAMEs; the default userids and passwords defined in metadata do not appear in code or logs that are accessible to the user. The SAS system provides multiple ways to setup and use meta-LIBNAMEs.

Data sources that are referenced in meta-LIBNAMEs must first be registered in metadata; this is usually done by SAS Admins but some users of SAS Data Integration Studio also have this capability. For most users, if a meta-LIBNAME does not exist for a required data source, you should ask your SAS Admins to register the source and setup a meta-LIBNAME. The metadata engine also lets your SAS Admins control access to data sources registered in metadata.

Once they are setup, meta-LIBNAMEs for sources to which **all** users should have access can be included in an autoexec file and – for example – in SAS Enterprise Guide, the LIBNAME may be automatically active for all jobs or may require the user to click to manually assign the library. Pre-assigned LIBNAMEs are very convenient and user-friendly, but they violate security policies at some sites (discussed below).

Another way to use meta-LIBNAMEs is for the user to issue an individual LIBNAME that specifies the meta engine and the libref assigned in the metadata server for the target data source. For example:

```
LIBNAME mymeta META LIBRARY=”Oracle_library_name”;
```

Meta-LIBNAMEs hide passwords and userids when the default userid and password stored in metadata are used; this of course is the easiest usage. It is possible to override these parameters in user-written meta-LIBNAME statements by supplying the values of these parameters. However, if this is done, then the non-metadata methods described in this paper should be used to hide the passwords and userids.

Meta-LIBNAMEs can be setup so that, e.g., a group of SAS Enterprise Guide users have automatic LIBNAME access to multiple databases. Service accounts are often used for this application, and that can be problematic from the IT security view. Many enterprises require that individual database access must be assigned on a by-user, by-database basis, all database access requests must be approved by the DBA or IT other IT staff (i.e., SAS Admins might not have this authority), and service accounts are restricted only to production programs that are under change control. The next metadata-based method discussed (Authentication Domains) may provide a better alternative for sites that face this restriction.

For more information on meta-LIBNAMEs, refer to: *SAS® 9.4 Language Interfaces to Metadata, Third Edition.* Section: Introduction to the Metadata LIBNAME Engine; URL:

- https://support.sas.com/documentation/cdl/en/lrmeta/70119/HTML/default/viewer.htm#n1u5a581j3yiexn12aigv9g0jt5x.htm

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 system.
- 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.

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 more information on Authentication Domains, refer to these SAS documentation URLs:

- https://support.sas.com/documentation/cdl/en/bisecag/69827/HTML/default/viewer.htm#p1du6ccnyjmlkdn1pwc9q11m088w.htm
- http://documentation.sas.com/?docsetId=acreldb&docsetTarget=n0aiq25zc8u8u6n1i81my0a24sd3.htm&docsetVersion=9.4&locale=ja

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

SAS SYSTEM: NON-METADATA METHODS

PROC PWENCODE; a proxy for database passwords. PROC PWENCODE is a long-standing Base SAS procedure that, given a password, will return an encoded string that can be used in place of the password in SAS code. Technically, this does keep database passwords out of source code but the encoded string is a proxy for the password and can be used to access the target database in SAS. Your local IT security group may regard the security provided by PWENCODE as better than showing passwords in plain text, but less than satisfactory.

There are constraints on where passwords encoded by PWENCODE can be used; details are provided in later sections herein. Despite these limitations, PROC PWENCODE provides an additional layer of security, and it should be used in supported applications.

For more information on PROC PWENCODE, refer to the SAS documentation URL:

- http://support.sas.com/documentation/cdl/en/proc/70377/HTML/default/viewer.htm#n0r32dnp2ulgvtn1ilx3iwwy9elf.htm

Blotting passwords and encryption keys in SAS logs. Base SAS provides some methods to keep passwords and encryption keys out of logs. These methods are limited: they do not work for userids, do not work inside macros, and do not blot or hide sensitive parameters in source code. For more information, refer to the URL:

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

ALTERNATIVE NON-METADATA METHODS

In the sections that follow, we describe macro-based methods to keep passwords, encryption keys, userids, and other sensitive character strings out of SAS code and logs. Most of the material below was previously published and presented at the 2017 *Western Users of SAS Software* conference; paper citations and links to full text are provided in the references section. The methods are:

- Compiled, encrypted function-style SAS macro, written in a certain way and also vetted for security. This is a decentralized method, 1 macro per parameter to be hidden. This method is relatively secure in open code, but not when invoked inside a wrapper macro. (Billings 2017B).
- Complex, compiled, encrypted SAS macro that securely and silently assigns LIBNAMEs to relational databases and ODBC. This is a centralized method as it uses a common parameter file. (Billings 2017A, a paper inspired by Sherman and Carpenter 2009).

We will also observe that SAS metadata-based methods cover most – but not all – instances where a password or userid may occur in SAS source code/logs, and the non-metadata methods described here can be used to cover –in some instances - the cases not supported by metadata methods.

DECENTRALIZED METHOD: COMPILED, ENCRYPTED FUNCTION-STYLE MACROS

This method was derived as a side effect of the work described in Billings (2017A). The objective was to find a way to hide SAS-proprietary and AES encryption keys in SAS code and logs, and in particular when the target code (to access an encrypted file) is not embedded in a compiled, encrypted macro. We begin with a brief overview of the encryption methods supported in SAS.

The SAS® system supports encryption of native SAS data sets using 2 different encryption methods:

- a SAS proprietary method, and
- AES-256 encryption.

AES is short for Advanced Encryption Standard, and 256 refers to the use of a 256-bit key for encryption. AES encryption is widely used and has been adopted as a standard by the U.S. and other governments. AES encryption is new in SAS 9.4 and is not available in earlier releases of the SAS system.

Encryption can be applied at the data set level using data set options and via metadata-bound meta-LIBNAMEs (or in some contexts, standard LIBNAMEs). To encrypt a file using data set options, use code like the following when creating the file.

For SAS proprietary encryption:

- `data mylib.myfile (encrypt=yes pw=password);`
- `data mylib.myfile (encrypt=yes write=password);`
- `data mylib.myfile (encrypt=yes alter=password);`
- `data mylib.myfile (encrypt=yes read=password);`

Using PW= is recommended as it sets ALTER=, WRITE=, and READ= passwords in a single option; it is also simpler. SAS proprietary encryption uses relatively weak passwords, i.e., the passwords must be a valid SAS name, maximum of 8 characters in length, and are case-insensitive.

For AES encryption (SAS 9.4+):

- `data mylib.myfile (encrypt=aes encryptkey=key);   /*no quotes*/`
- `data mylib.myfile (encrypt=aes encryptkey="key"); /*double quotes*/`
- `data mylib.myfile (encrypt=aes encryptkey='key'); /*single quotes*/`

The rules for what constitutes a valid key vary somewhat depending on whether the no quotes, double quotes, or single quotes format is used; consult the SAS documentation for details (check reference section for relevant URL).

To use an encrypted file in code, reference the file using the relevant data set option:

```
mylib.myfile (read=password)
mylib.myfile (encryptkey=*)   and * = one of: key, "key", 'key'
```

The requirement to supply the password or key with the data set name in code is problematic as best practice is to keep sensitive or confidential information out of source code and SAS logs.

Challenges and constraints when using encryption keys

When you create an AES-encrypted file in SAS, the SAS system issues this note of advice:

```
NOTE: If you lose or forget the ENCRYPTKEY, there will be no way to open the
file and recover the data.
```

The above note should be taken very seriously, because:

data loss caused by a missing encryption key could have a major and negative impact on your employer and/or your career.

Caution and due diligence should be exercised in keeping a secure record of keys used. Safeguarding the keys should be a critical part of data governance in enterprises.

Format of the key is important. As described above, the AES encryption key can be in one of 3 different formats – a plain string (no quotes), in double quotes, or in single quotes. One might think (naively) that if a string is used as key in one format, then that same key would work in the other formats, after adding or subtracting quotes. However, this is only partially true; the table below shows how a file saved with an AES key in 1 format can throw an error when a read is attempted using the same key in a different format:

SAS data set created using AES encryption key in format:	**Data set access method:**		
	No quotes	**Double quotes (")**	**Single quotes (')**
No quotes	Works	Throws error	Throws error
Double quotes (")	Throws error	Works	Works
Single quotes (')	Throws error	Works; macro quoting constraint	Works

The reason for the errors above is structural: an AES key that is valid in single or double quotes may throw a syntax error when used without quotes.

> **Note:** the encryption key string (by itself, no quotes) can be supplied via a macro/macro variable in the code for the no-quotes and double quotes formats, but not the single quotes option as single quotes prevent macro expansion. To clarify this point: if we create a file with the key "xyz" (double quotes) then we can supply the key as a macro/macro variable in any of the forms: 1) "&key." or "%key" so long as &key, %key resolve to xyz (no quotes), or 2) &key., %key so long as these resolve to "xyz", i.e., xyz embedded in double quotes. Form 2 in the preceding is recommended as being more secure; this is discussed in a later section.

Limited flexibility in specifying passwords or keys. A number of techniques that are sometimes used to mask or hide strings do **not** work as operands for PW=, ENCRYPTKEY=:

- SYMGET("macro_variable_name")
- RESOLVE('&macro_variable_name')
- Passwords or keys encoded using PROC PWENCODE

Other methods to hide the code might be N/A. System options NOMPRINT, NOSYMBOLGEN, NOMLOGIC can be specified to keep information out of the log (from macro-run code), but this approach can be cumbersome as it requires encapsulating code in macros and additional coding. Additionally, the system options NOSOURCE, NOSOURCE2, and NONOTES may be needed to keep sensitive information out of the log.

Sensitive parameters can be held in macro variables but the %LET statement will show in source code, even if the appropriate options are set to suppress the log. Also, sensitive data stored in global macro variables can be exposed by a simple %PUT _ALL_; statement after the fact (in the same SAS session).

Function-style, compiled, encrypted macros

The difficulties in keeping passwords and keys out of source code and the log are described above. However, there is a method that works: make the sensitive information, whether a plain text string with no quotes or a string denoted by single/double quotes, the output produced by a function-style compiled, encrypted macro. To make the macro secure, it should include tests on &SYSUSERID, i.e., the macro can be made to work for 1 and only 1 userid if desired.

Macros are needed for PW= and each of the 3 forms of ENCRYPTKEY=. The code below works; additional discussion follows the text. All code in this paper is released under the BSD 2-clause open source copyright license which allows free reuse under conditions; see Appendix 1 for details. The data sets used below contained artificial data, and the encryption key and password are also artificial hence can be public.

For SAS proprietary passwords:

```
%macro mypswd/ store secure;
%*macro mypswd;
      %local saspswd myuserid;
      %let myuserid=redacted;
      %let saspswd=X1Y_59KM;
      %if (&sysuserid. = &myuserid.) %then %superq(saspswd);
            %let saspswd=;
            %let myuserid=;
%mend;
```

For AES encryption key; key has no quotes:

```
%macro myaes1/ store secure;
%*macro myaes;
      %local EK myuserid;
      %let myuserid=redacted;
      %let
EK=XTPzMEd8pvIDBmlXKfjrwvNk0fAfTXiwmUNYniT0fOhKibzs3dEhVvCCQeyF8U2K;
      %if (&sysuserid. = &myuserid.) %then %superq(EK);
            %let EK=;
            %let myuserid=;
%mend;
```

For AES encryption key; key in double quotes:

```
%macro myaes2/ store secure;
%*macro myaes;
      %local EK myuserid;
      %let myuserid=redacted;
      %let
EK="XTPzMEd8pvIDBmlXKfjrwvNk0fAfTXiwmUNYniT0fOhKibzs3dEhVvCCQeyF8U2K";
      %if (&sysuserid. = &myuserid.) %then %unquote(&EK.);
            %let EK=;
            %let myuserid=;
%mend;
```

For AES encryption key; key in single quotes:

```
%macro myaes3/ store secure;
%*macro myaes;
      %local EK myuserid;
      %let myuserid=redacted;
      %let
EK='XTPzMEd8pvIDBmlXKfjrwvNk0fAfTXiwmUNYniT0fOhKibzs3dEhVvCCQeyF8U2K';
      %if (&sysuserid. = &myuserid.) %then %unquote(&EK.);
            %let EK=;
            %let myuserid=;
%mend;
```

Notes/details:

1. **Creating compiled macro libraries.** The macros need to be stored in a compiled, encrypted macro library, per the /STORE SECURE option on the %MACRO statement. Autocall libraries are not appropriate because they contain unencrypted source code.
2. **Using compiled macro libraries.** To use compiled macros, the system options MSTORED and SASMSTORE= must be specified; the latter specifies a libref or catref for the stored macro catalog/library. The macro library should be ACCESS=READONLY in LIBNAME statements in normal usage; the only exception being jobs that create compiled macros. Compiled macro libraries can be concatenated if necessary.
3. **Security: delete logs, change code after compilations.** The code used to compile the macros and associated log will contain sensitive information. These logs should be deleted and the macro source code modified to remove sensitive information, before the file gets copied to nightly backup or to a source code repository. A secure record of the passwords/keys should be kept, possibly offline.
4. **Security: the macro works for only 1 userid.** This code in the macro: `%if (&sysuserid. = &myuserid.) %then` limits the macro to work only for a single userid, and provides security. This can be modified to include other userids, or omitted. (Warning: If this check is omitted, then the macro is **not** secure!) If a list of userids must be supported, the code can be modified to accommodate multiple users, but take note: that induces a maintenance requirement.
5. **Security: complementary with PROC PWENCODE**. This method also works to deliver database passwords encoded using PROC PWENCODE, for additional security. However, recall that passwords/keys encoded with PWENCODE do not work for encrypted SAS data set access.
6. **Security: operating system and metadata permissions.** Access to the directory containing the executable form of the compiled macro can be restricted using operating system and/or SAS metadata permissions. Having multiple layers of security is a "best practice".
7. **Special characters and macro quoting.** The single/double quote macro versions here were tested using keys/passwords that had no special characters. Ideally, the macros should not throw an error because of special characters. To avoid this, for AES encryption keys, use the no quotes form for simplicity as it disallows special characters. If possible, avoid use of special characters in strings (passwords or userids) to be protected with these macros. If special characters must be supported, you may need to modify the macro code to use the appropriate macro quoting functions (e.g., %SUPERQ) and/or mask any unmatched single or double quotes or parentheses in strings defined using %STR or %NRSTR. For more information, see the References section for a link to SAS documentation on macro quoting functions.
8. **Function style macros: potentially fragile code.** To work correctly, function-style macros must return a character string with no semicolons. The code won't work correctly if:
   - ➢ A null statement (; only or blanks plus a ;) is inserted in the macro
   - ➢ Comments of the form: * comment ; are inserted in the code
9. **Constraints on macro names.** Testing during development found that macro names like %mymacro(num=1) will throw an error when used as SAS data set passwords or encryption keys; it seems that they do not parse correctly in some of the contexts tested. Suggestion: keep the macro names short (max. 8 characters including % character), and don't have any macro parameters. (Note, however, that longer macro names – with no arguments - did work with tests for relational database passwords.)
10. There should be 1 macro for each parameter.
11. For sample code to generate random AES keys, see Billings (2017A).

Constraint: compiled, encrypted macros are not secure. The use of compiled, encrypted macros does **not** guarantee security. Opening a compiled macro file in a text editor like notepad or vi reveals that some/many SAS macro statements are masked in the compiled file, but some SAS code may appear as plain text on the editor screen. In some cases, macro statements may be visible in the compiled file. This is a security issue if sensitive information is visible in the compiled macro file.

The macros above uses the method of encapsulating sensitive parameters in %LOCAL macro variables, with a %LET statement in the code, which is cleared after the macro runs. This method is described in Billings (2017A) and it appears to work when only 1 macro is compiled.

Adding an additional macro to the compile, a stub macro, is worth doing: in tests done for this paper, the code for the last macro (of n>1 macros) compiled may be visible in the executable macro file, but code for the earlier macros (compiled in the same run) was hidden. If a stub macro is compiled last, its code may be exposed (and that does not matter) but the code for the preceding macros may be hidden.

In this context, a stub macro is very simple: a %MACRO statement with a name, %RETURN , and %MEND. Note: This information is observational and cannot be guaranteed. Always check the compiled macro executable file to make sure that sensitive information is not visible! If the hacks mentioned here do not work to hide sensitive information, then you may need to develop other hacks.

Of course if you are the **only** user who can access the compiled macro catalog file as enforced by operating system or metadata permissions, then you may be able to skip some of the security checks above.

The macros work and keep passwords/keys out of the code/logs (but see constraint below). The macros above were compiled and the test code below was run successfully against data sets containing artificial data. Data sets test_1, 2, 3 are aes-encrypted files whose key has no quotes, double quotes, and single quotes respectively. Data set text_4 is encrypted using SAS proprietary encryption.

In the log below, the associated macro is invoked in the code in place of the sensitive parameters. Note that macro %myaes1 outputs a string without quotes and can be used successfully inside double quotes (3rd PROC PRINT below). The macros %myaes2, %myaes3 respectively output a string in double/single quotes, and %mypswd outputs a SAS proprietary [password.

```
25         options mprint symbolgen mlogic macrogen;

27         libname stdmcr "/redacted/" access=readonly;
NOTE: Libref STDMCR was successfully assigned as follows:
      Engine:        V9
      Physical Name: /redacted/
28         options mstored sasmstore=stdmcr;
29         options nocenter dtreset;

31         libname CT "/redacted/;
NOTE: Libref CT was successfully assigned as follows:
      Engine:        V9
      Physical Name: /redacted/

33         proc print data=ct.test_1 (encryptkey=%myaes1);
34         run;

NOTE: There were 100 observations read from the data set CT.TEST_1.

36         proc print data=ct.test_2 (encryptkey = %myaes2);
37         run;

NOTE: There were 100 observations read from the data set CT.TEST_2.

39         proc print data=ct.test_2 (encryptkey = "%myaes1");
40         run;

NOTE: There were 100 observations read from the data set CT.TEST_2.
```

```
42          proc print data=ct.test_3 (encryptkey = %myaes3);
43          run;

NOTE: There were 100 observations read from the data set CT.TEST_3.

45          proc print data=ct.test_4 (PW=%mypswd);
46          run;

NOTE: There were 100 observations read from the data set CT.TEST_4.
```

The test code/log above shows that the approach of using encrypted, compiled function-style macros (coded as described here) to provide the password/encryption key in data set options works. Furthermore, the values of the password/encryption key do **not** appear in the source code or log, despite the use of SAS system options that normally disclose the internal operations/code generated by macros. When the macro code includes the test on &sysuserid as done here, then the macro will not work for anyone but the authorized userid as &sysuserid is system-set and cannot be reset by a user.

Additionally, the common macro hacks:

`%put %macro_name;` and/or

```
%let secret=%macro_name;
%put &secret.;
```

do **not** work with these compiled macros for userid(s) other than the authorized userid(s).

Additional applications of this method

The method works to assign parameters for the ODS pdf password system option statement. It also works with LIBNAME parameters, using macros with output that has no quotes, double quotes, and single-quotes. These macros can be used in the Excel DDE FILENAME approach to keep Excel file passwords out of the log/code, but at the same time feeding them to DDE. (Test macros were used with some of the code of Hao (2013) and the output string verified by visual inspection.) The macros also work to supply passwords & userids for PROC SQL CONNECT syntax.

In the applications above, no sensitive parameters appeared in the code or log, despite the invocation of system options MPRINT, SYMBOLGEN, MLOGIC, and the obsolete MACROGEN. Given the preceding, it follows that the use of compiled, encrypted macros to supply sensitive strings is not limited to PW=, READ=, ALTER=, WRITE=, or ENCRYPTKEY= fields in data set options. Presumably it can be used wherever single-token strings need to be protected in SAS code.

Important: constraints of this method. Additional testing revealed limitations of this approach. First, if you have a parameter that must be in quotes or that you choose to supply in quotes, then you should – for more security – use the form %mypswd returns “my_password” and not the unquoted string: password. To clarify, instead of, say:

`SERVERPASS = "%mypswd"`, use the form:
`SERVERPASS = %mypswd` (i.e., no quotes, and the macro output includes the quotes).

Second, compiled, encrypted macros automatically turn off MPRINT, SYMBOLGEN, MLOGIC, and that helps this method work in open code, i.e. code that is not encased in a macro. However, when a compiled, encrypted macro is embedded inside an outer, wrapper macro, and if MPRINT is enabled for the outside (wrapper) macro, then the sensitive information may appear in the log anyway.

Third, and this is well-known: notes and the log can be suppressed but error messages are difficult to suppress. If you use this approach in an open-code statement that produces an error message, then sensitive information may be revealed in the error messages.

Mitigations for this include:

- Turn off MPRINT, SYMBOLGEN, MLOGIC in the wrapper macro, at least for strategic portions of the code that include sensitive information,
- If feasible, modify the logic so invocations of the compiled, encrypted macros are in open code and not encased in a macro (this may be easy or difficult, depending on the application).

This is an important constraint to keep in mind when using this approach. During testing for production programs, it is wise to double check and make user that no sensitive information appears in the log.

Note: another user cannot take your compiled macro, embed it in a wrapper macro, and use it to acquire your password. The test on &sysuserid in the macro code means that however/wherever the macro is called, it returns a non-null result only to your userid.

Implementation issues

Test first. ALWAYS test your macro before using it in code. To test the compiled macro, use code like the following, run under your userid:

```
%let myvar=%mymacro;
%put &myvar.;
```

Per the preceding section, the code above will **not** reveal your password if run under a userid other than yours. The macro code shown here includes a commented-out %*MACRO statement specifically to make it easy to create a version of the macro that is not stored/encrypted, for testing.

Possible compiled macro library access issues. If you are working in SAS Enterprise Guide and you compile a macro and then try to use it in the very same run, you may encounter access issues even if you issue LIBNAMEs to clear the relevant libref and then reassign it. When this happens, simply save the project/code, exit, then restart SAS Enterprise Guide and use the just-compiled library. For more options to try in this situation, see Hemedinger (2016).

CENTRALIZED METHOD: COMPILED, ENCRYPTED MACRO TO ISSUE LIBNAMES FOR DATABASES, ODBC

Objective and requirements. Develop a method in Base SAS® to securely and silently assign LIBNAMEs/librefs to relational database systems (RDBMS). Secure & silent means the programmer can assign the LIBNAME but does not know and cannot acquire, sensitive database connection parameters (userids and passwords). The method should **not** protect database path, DSN (for ODBC), or schema names as programmers need this information.

If database connection parameters are saved in files, those files must be encrypted and the location of those files must not be revealed in the log. Both SAS-proprietary and AES encryption should be supported for parameter files.

The method should not **require** metadata but should be able to interact with metadata if needed, i.e., it should support the ability (option) to replace user-defined LIBNAMEs with meta-LIBNAMEs or standard production LIBNAMEs (when appropriate). The method should be able to support multiple database parameter files, e.g., a main or central file controlled by admins, and user-controlled files.

To the maximum extent possible, the method should be secure:

- If implemented via a macro, then SAS options that disclose the inner workings of macros (e.g., MPRINT) should not reveal any sensitive information,
- known hacks like the following should not disclose sensitive information:
  - %PUT %macro_name_here;
  - %PUT _ALL_; and
  - LIBNAME _ALL_ LIST;
- the method should work with SAS proprietary encryption and also with AES encryption (SAS 9.4+; not available in earlier releases).

To clarify security requirements, the following are to be protected and kept secret from users:

- Userids for target databases,
- Passwords for target databases,
- Location/name of (encrypted) database parameter files,
- Passwords or encryption keys for encrypted database parameter files.

Source code for the method is not secret, so long as the above items are protected.

Background

Sherman & Carpenter (2009) method. The paper by Sherman & Carpenter (2009) presents a secure method to access databases that does not require metadata and works in any environment. It is a 3-step SQL-based approach; in a single PROC SQL invocation, their method is to:

1. Pull database login parameters from a SAS file (unencrypted in early versions of their paper), and save in macro variables
2. Use the database connect parameters with PROC SQL CONNECT TO syntax, to perform the target data pull
3. Follow-up with another SQL SELECT statement to clean up (erase) the relevant macro parameters.

They suggest using step 2 to issue a LIBNAME function call via an SQL SELECT statement. Unfortunately, their method does not work with LIBNAMEs for relational databases: the LIBNAME function cannot have null arguments in PROC SQL and one argument to the function call must be null for databases (i.e., the physical directory associated with the LIBNAME). This contradiction prevents use of their excellent method for the task of assigning LIBNAMEs to relational databases. (Their method will work for LIBNAMEs for SAS files, but simpler methods are available for that application.)

Their paper/method, along with the resultant challenge to securely assign LIBNAMEs to RDBMS systems, inspired and motivated the development of the centralized method in this paper. Additionally, the requirement to protect database login parameters in an encrypted SAS file created additional challenges, i.e., the need to make sure the encryption keys/passwords for files are kept secret.

Additional security and operational issues

We modified the approach of Sherman & Carpenter (2009), replacing some of their SQL with DATA _NULL_ steps. The code for the method is encapsulated in an encrypted, compiled macro. Our approach induces additional operational design features and requirements.

Compiled, encrypted SAS macros are not secure. The use of a compiled, encrypted macro does **not** guarantee security for the source code. Opening a compiled macro file in a text editor like notepad or vi may reveal that some of the SAS code is masked in the compiled file, but some code may appear as plain text on the editor screen. Mitigation methods for this issue were previously described in the discussion for the decentralized method.

Constraints on SAS-proprietary passwords and AES encryption keys in SAS code. The code used to access an encrypted file must include the relevant AES encryption key or password as data set options. This topic was covered in a preceding section, and the comments here apply to the similar implementation in the secure centralized approach.

SAS provides 3 ways to supply AES encryption keys, which can be up to 64 characters in length:

```
1. ENCRYPTKEY=string or ENCRYPTKEY=&my_string
2. ENCRYPTKEY="string" or ENCRYPTKEY="&my_string"
3. ENCRYPTKEY='string'.
```

Options 2 & 3 allow blanks and special characters which can complicate their use in macros (especially trailing blanks). Experimentation led us to select/use option 1 with no special characters; it is the weakest key of the 3, but it is still highly secure, avoids the problems of the other options, and works best in operation.

Notes can be suppressed but warnings and error messages are difficult to suppress. Notes and the display of source code in the log can be suppressed, but it is more difficult to suppress error messages and warnings. Because of this, the macro has extensive error checking and handling, to prevent disclosure of sensitive information when errors occur.

As a general principle, the level and degree of error checking done should reflect the importance of the program and the underlying requirements. The sensitive parameters here are slowly changing dimensions that are under control of admins and/or programmers. Admins should monitor changes in database login credentials; a test program that checks every database connection, 1X every day, is recommended for operations.

Database parameter files: requirement to support multiple databases and environments. Database parameter files are used to contain the database login parameters. Each database & environment is a separate row in the parameter file. Two variables uniquely identify each row/connection:

- source_name = label for or name of database
- env = intended to be the run environment, i.e., dev, test, prod, but can be anything you want.

The database parameter file also contains in each row, variables that specify:

- A default libref to be assigned (used when no libref is supplied by the user or an invalid libref is specified),
- Userid and password for the remote database
- SAS data engine to use – Oracle, ODBC, etc.
- Schema name for the remote database
- Path (set to missing for ODBC connections)
- DSN (ODBC only; set to missing for non-ODBC)
- An override indicator, described below.

Variable lengths as used in macro development are:

```
length source_name env $30.;
length default_libn $8. db_user engine $30.;
length pass $300.;
length path schema dsn $40. override $5.;
```

Libref override. Two operational situations need to be accommodated:

- Programmers use their own librefs during development (and they should be macro variables for portability) but a uniform libref is to be used when the programs go into production, and/or
- Over time, a data source may be assigned a meta-LIBNAME or standard production LIBNAME/libref, and programs should use that LIBNAME/libref.

The above is supported as follows:

- Each row in the database parameter file has a variable for a default libref and an override indicator. When the override indicator is set (to yes), the user-supplied libref is ignored and the default libref is used for the macro processing. If no libref is supplied by the user, the default value in the parameter file is used.
- The macro returns the assigned libref in a global macro variable, and any associated error indicators
- For the above to work optimally in practice, users should write their own macro to invoke the main macro and check the resultant error indicators and use the assigned libref as returned OR abort the run if appropriate. This is not absolutely necessary but is recommended.
- The above will encourage those who use hard-coded, non-macro-variable librefs to change their code.
- In the database parameter file, if the variable override is set to 'no' and the user supplies a valid libref, that libref will be used by the macro.

Need to create encrypted parameter file. The use of an encrypted database parameter file means that:

- The admin and/or users who wish to have their own parameter file will need to create an encrypted file that provides the relevant database connection parameters. To maintain security, database connection parameters should be tested before (and after) the information is added to the parameter file.
- The LOGs for programs that create or update database parameter files should be deleted, for the obvious security reasons.
- For AES encryption, a program that creates a random encryption key (user-selected length; 64 characters is recommended) was developed. Use of this program is optional; see Appendix 2 in Billings (2017A) for the program code.

Admin-owned & user-owned parameter files. The requirement to support multiple database parameter files (admin-owned and/or user-owned) is supported as follows.

- The location/name of the main or default database parameter file is specified in the main macro.
- Users who wish to provide/use their own database parameter file will need to write a macro that provides the location/name of their file. The macro must have a pre-set name: %PA_LIBR, and it too must be complied and encrypted – but in a separate, user-supplied library (code for the macro is below).
- A "stub" version of the %PA_LIBR macro is stored in the compiled macro library that contains the main macro. This stub version returns nothing, and is used as a signal in controlling when the default parameter file location is used.
- Library/catalog concatenation is used to cause the user-defined version of PA_LIBR to be invoked instead of the stub version:
  - ➢ The user must specify libnames pointing to both macro libraries, #1 that contains their version of %PA_LIBR and #2 the main library with the stub version of the macro,
  - ➢ then specify a 3rd libname or catname for the concatenated libraries (#1 and #2).
  - ➢ The concatenated libref or catref is then specified with OPTIONS MSTORED SASMSTORE=libref or catref.
  - ➢ If the user-defined libref is the 1st library in the concatenation, then the user-defined version of %PA_LIBR will be invoked/used instead of the stub/default version

Sample SAS code to concatenate compiled macro libraries: in the code below, the user library is listed first in the concatenation, so a user-defined version of %PA_LIBR will be invoked instead of the stub version.

```
libname stdmcr "path_to_main_library"  access=readonly;
libname stdmcr2 "path_to_user_library" access=readonly;
libname mcrolib (stdmcr2 stdmcr) access=readonly;
options mstored sasmstore=mcrolib;
```

Users who do not need to provide their own database parameter file can ignore the above and instead rely on the default database parameter file.

Sample macro to supply location information for user-owned database parameter file

The code is below; fill in the location information as needed (i.e., macro variables at top of program). Notice the test of &sysuserid against the %local macro variable value &usr. As written, this macro (deliberately) works for 1 and only 1 userid. Of course it can be modified to support multiple userids if needed; the SAS macro language supports an IN operator if the MINOPERATOR system option is selected (also see related option MINDELIMITER=).

Comments in the macro code provide an explanation of variables. The multiple %let statements contain the sensitive parameters. If a password or any of the parameters below contains a special character that can throw errors in the macro language, you will need to modify your code to use the appropriate macro quoting functions. Also note use of %superq function - that might not be required in some cases, but is suggested to reduce the risk of run-time errors in macro execution. The macro variables are cleared on exit, to prevent disclosure by looking in memory after execution.

Note: The SAS code below is released under a *BSD 2-clause open source copyright license*; see Appendix 1 for the license text.

```
%macro pa_libr(nbr) / store secure;
      %local usr pth fnm psw ekey;
      %let usr=*****;     /* fill in:  authorized userid */
      %let pth=*****;     /* fill in:  path to file */
      %let fnm==*****;    /* fill in:  SAS data set name */
      %let psw==*****;    /* fill in:  SAS-proprietary password */
      %let ekey=*****;    /* fill in:  AES encryption key for file */

      %if %length(&nbr.) = 0 %then
            %goto exit;

      %if ((&nbr. ne 1) and (&nbr. ne 2) and (&nbr. ne 3) and (&nbr. ne 4))
%then
            %goto exit;

      %if (&sysuserid. = &usr.) %then
            %do;
                  %if (&nbr. = 1) %then
                        %superq(pth);

                  %if (&nbr. = 2) %then
                        &fnm.;

                  %if (&nbr. = 3) %then
                        %superq(psw);
```

```
                    %if (&nbr. = 4) %then
                          %superq(ekey);
              %end;

%exit:
      %let usr=;
      %let pth=;
      %let fnm=;
      %let psw=;
      %let ekey=;
%mend;
```

The stub version of the macro is as follows:

```
%macro pa_libr(nbr) / store secure;
%*macro pa_libr(nbr);
      %* stub macro, returns nothing;
      %return;
%mend;
```

Readers may note the similarity of the code above, compared to the simpler decentralized macro presented in earlier sections. The macro above inspired the simpler, decentralized approach.

Main macro to assign LIBNAMES to RDBMS systems

The source code for the main macro is presented in full text, with annotation, in Billings (2017A). Due to page length limits, the code is not listed here. Instead, we present an outline of the code and processing, as follows:

1. LIBNAMEs and OPTIONS to access stored macro library.
2. Startup: capture values of relevant system options and save in macro variables, then turn off all system options that reveal information about the workings of the macro.
3. Use support macro %PA_LIBR to get location of database parameter file; the file may be user-supplied or default.
4. Logic to check the values of user-supplied macro variables; use default values if no user-supplied values.
5. If user-supplied libref (for target database), check for validity using NVALID function inside a DATA step (this function should not be used with %sysfunc or %qsysfunc).
6. We need a libref for the database parameter file. Generate a random libref string by concatenating a character (%LOCAL variable inside the macro) with a random number produced using the RAND function, then converted to character with PUT.
7. Define LIBNAME that points to the database parameter file; check that target SAS file is present in directory; error handling for exceptions.
8. Use SQL to pull the desired database connection parameters from the encrypted parameter file. For security, clear the libref after the pull, even if it threw an error.
9. The SQL pull of database connection parameters may throw an error or return nothing. Check and pursue an error exit if warranted.
10. Determine the proper libref name (for target database) per database parameter file and user-supplied libref. Check if libref already assigned; terminate if necessary.
11. Issue LIBNAME for target database. Check for successful connection and set macro variables for results. The output libref is in a global macro variable.
12. Normal and error exits:
    - clear %LOCAL macro variables
    - DATA _NULL_ step to clear anything visible via function SYSMSG()
    - print messages to user

- restore system options to previous values.

Implementation

The ordering below is not strict; some things can be done in parallel (only main steps are listed).

1. If not already in-hand, get access to and valid/tested login parameters for at least some of the target relational database environments of interest.
2. Create AES encryption key (see Appendix 2) or determine SAS proprietary password
3. Create an encrypted database parameter file. Set/check relevant operating system (and/or metadata) permissions on the file.
4. Migrate a copy of the macro source code to a directory/file or source code management repository, e.g., Git, that is restricted access/use.
5. Modify the macro program code to add sensitive parameters (SAS proprietary passwords, encryption keys, database parameter file path/name). Compile the macros. (Main macro and %PA_LIBR supplementary macro if needed.) Verify that sensitive data cannot be seen in the compiled file.
6. Test that the macro works -- access to target databases
7. Delete logs from the macro compilation step(s) as those contain sensitive information. If you are using a tool like SAS Enterprise Guide that saves logs in the .egp file by default, you may need to take additional steps to delete the logs.
8. Write wrapper macro to invoke the main macro to setup LIBNAMEs (optional; recommended)
9. Change relevant programs – if necessary – to invoke the wrapper macro and use the librefs setup by the secure LIBNAME macro.

Implementation has a cost in terms of setup effort, but once running, the process should require only limited maintenance as database parameters are usually (very) slowly changing dimensions.

Implementation issues

Debug is a challenge. Because the macro is compiled and produces no log, no notes (other than macro-generated messages), the macro cannot be debugged as-is when errors occur. Instead, for debug, create a new version of the macro that:

- Is **not** a stored, compiled macro
- Change the code so MPRINT, MLOGIC, SYMBOLGEN work normally.
- Rerun the uncompiled version to try to reproduce the error message(s).

After debugging, create a revised version of the stored, encrypted, compiled macro, and delete the uncompiled version.

Encrypted files are more difficult to work with:

- To open an encrypted file in most SAS tools or code, you must enter the encryption key or password (or include it in code), depending on the type of encryption used.
- If you don't have a password or encryption key, you can't do much with encrypted files.
- Any SAS logs that show encryption keys or passwords should be deleted. Similarly, sensitive parameters should be deleted from source code files as well (can delete after runs).

The exception here is meta-LIBNAME files that are metadata-bound and encrypted. For these, encrypted file access is handled automatically by the SAS system – no need to enter passwords or keys if you have the proper metadata access. However if the database parameter file is encrypted this way, then anyone who uses the macro has read access to the database parameter file – not a good idea for security!

How to control access to the macro?

- Use your operating system permissions (for the compiled macro file and/or database parameter files)

- Use metadata permissions in a client-server environment (control access to database file)
- Modify the main macro to check &sysuserid against a list of authorized users. Constraint: this approach requires ongoing maintenance of an authorized user list, something to be avoided when possible.

Future considerations

- In an email exchange, Art Carpenter suggested replacing some of the DATA _NULL_ steps used to invoke the LIBNAME function with macro code that uses %SYSFUNC paired with %SUPERQ. Readers may wish to experiment with this; it might yield cleaner code.
- The DATA _NULL_ steps in the macro are not compiled and hence do not use the SOURCE=NOSAVE or SOURCE=ENCRYPT options as it is not necessary for our application. Others may find these potentially useful and/or might be used to prevent disclosure of source code. (These options work only with compiled DATA steps whereas the methods described above presumably work with any SAS code in a compiled, encrypted macro.)
- The macro was developed for and tested with database applications that are read-only. Changes may be needed for applications that require write access, e.g., additional options may need to be supplied in the LIBNAME function calls.

CONCLUSIONS

We have presented 2 non-metadata approaches to keeping userids, passwords, encryption keys, and other sensitive/confidential strings out of SAS source code and logs:

- a **decentralized** macro-based approach to supply individual parameters (tokens) like passwords, encryption keys, userids, and
- a **centralized** macro-based approach to securely assign LIBNAMEs to relational databases using a database parameter file.

The decentralized method is much easier to apply than the centralized method and is applicable to more applications; however for security it should be used only in open code and not inside wrapper macros. Appendix 2 (below) provides a table that compares the metadata and non-metadata approaches for keeping passwords out of source code and logs.

The table shows that there are a few applications in SAS where passwords or userids are required, but metadata-based methods are not available for those instances. The table also shows that the methods described in this paper, especially the decentralized approach, can – in some instances - support these applications and hence can be used to fill in the gaps in coverage of metadata-based approaches.

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 copyright license, an open-source license
  that permits free reuse and republication under conditions;
```

APPENDIX 2:
METHODS FOR HIDING SENSITIVE PARAMETERS IN SAS: SOURCE CODE & LOGS

Methods for Hiding Sensitive Parameters in SAS: Source Code & Logs

Type of sensitive information	**Context**	**Metadata-based methods**	**Non-metadata methods**
AES encryption keys, SAS proprietary passwords	Data set options, specialized LIBNAMEs	Metadata-bound libraries	Billings (2017B) macro method
DBMS userids, passwords	Accessing databases via LIBNAMEs	Authentication domains; meta-LIBNAMES using default parameters	Billings (2017A, B) macro methods
DBMS userids, passwords	Accessing databases using CONNECT TO syntax	Authentication domains work, but this is not clear in SAS documentation. Alternate syntax: CONNECT USING libref.	Sherman & Carpenter (2009) method; Billings (2017B) method
Userids, passwords for most external file interfaces	FILENAMEs: ftp, sftp, URL, WebDAV, etc.	Authentication domains	Billings (2017B) macro method
Userids, passwords for files	Authentication domains NOT supported: LIBNAME for data engines PCFILES server, EXCEL, ACCESS; PROC IMPORT, EXPORT, etc.	N/A; use non-metadata method.	Billings (2017B) macro method
PDF, Excel passwords (output files)	Writing password-protected pdf, Excel files in SAS code	N/A; use non-metadata method	Billings (2017B) macro method

Note: some of the non-metadata methods are limited to open code, i.e., are not secure when used inside a wrapper macro unless MPRINT and similar options are turned off in the wrapper macro.

REFERENCES

Note: all URLs quoted or cited herein were accessed in May 2017.

Billings T (2017A). Secure Macro-Based Method to Assign LIBNAMEs for Databases. Paper to be presented at 2017 *Western Users of SAS Software.*

Billings T (2017B). Keeping Passwords, AES Encryption Keys, and Other Sensitive Parameters Out of Source Code and Logs. Paper to be presented at 2017 *Western Users of SAS Software.*

Hao, J (2013). SAS Automation - From Password Protected Excel Raw Data to Professional-Looking PowerPoint Report. *Midwest SAS Users Group Conference Proceedings.* URL: http://www.mwsug.org/proceedings/2013/FS/MWSUG-2013-FS04.pdf

Hemedinger C (2016). Tip: How to close all data sets in SAS Enterprise Guide. *SAS Blog: The SAS Dummy.* URL: http://blogs.sas.com/content/sasdummy/2016/10/18/close-data-sas-eg/

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/

SAS Institute, Inc. online documentation:

- Blotting Passwords and Encryption Key Values. *SAS® 9.4 Language Reference: Concepts, Fifth Edition. URL*:http://support.sas.com/documentation/cdl/en/lrcon/68089/HTML/default/viewer.htm#n0f79bcfsnsl82n117dahzoqrsdn.htm
- ENCRYPTKEY= Data Set Option. *SAS® 9.4 Data Set Options: Reference.* URL: https://support.sas.com/documentation/cdl/en/ledsoptsref/68025/HTML/default/viewer.htm#n0yzx049gh8pn3n1v7yrzagard3a.htm
- Macro Quoting. *SAS® 9.4 Macro Language: Reference, Fifth Edition.* URL: http://support.sas.com/documentation/cdl/en/mcrolref/69726/HTML/default/viewer.htm#p07u5itr1teq0dn1bx0lli1ri5dy.htm
- *SAS® 9.4 Guide to Metadata-Bound Libraries, Second Edition.* URL: http://support.sas.com/documentation/cdl/en/seclibag/66930/HTML/default/viewer.htm#titlepage.htm
- Usage Note 38204: Using the AUTHDOMAIN= option with SAS/ACCESS® 9.2 engines. *SAMPLES & SAS NOTES.* URL: http://support.sas.com/kb/38/204.html

Sherman P, Carpenter A (2009). Secret Sequel: Keeping Your Password Away from the LOG. *SAS Global Forum Proceedings.*
URL: http://sascommunity.org/wiki/Secret_Sequel:_Keeping_Your_Password_Away_from_the_LOG

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

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.

Remote from:
Merritt Island, Florida 32952

Phone: 321-453-5694
Email: tebillings@gmail.com