Secret Sequel:
Keeping Your Password Away From the LOG

Paul D Sherman, San Jose, CA
Arthur L. Carpenter, CA Occidental Consultants

ABSTRACT

Passwords are things not to be shared. As good SAS® programmers we don't hard-code our passwords in our programs, but keep them in separate files or macro variables. However this is not enough! Unfortunately the casual (or malicious) user can "turn on" macro debugging utilities like MPRINT, MLOGIC or SYMBOLGEN and literally read our password right from the LOG, in plain view. This paper presents a series of simple methods to keeping your secret password secret when accessing remote databases with PROC SQL.

Skill Level: Beginning Base/SAS, Macro, and SAS/ACCESS

KEY WORDS

PROC SQL, SQL pass through, password, SYMBOLGEN

THE PROBLEM WITH PASSWORDS

Although we generally do not need to use passwords and user id's when accessing SAS data tables, it is not at all unusual to need them when accessing data base systems such as Oracle or DB2. When the process includes SQL pass through, the user id and password must be included in the SAS program. It is the inclusion of this sensitive information in the SAS program that we are addressing in this paper.

You have been there. Time is short, you are in a hurry, and you are creating programs on-the-fly. In cases like this it is so very easy to simply hardcode the password.

```
Proc SQL noprint;
  connect to odbc(dsn=dbprod uid=mary pwd=wish2pharm);
  . . . . code not shown . . . .
```

You are hoping that no one else will see the program, but you must also remember to protect the LOG which will show:

```
pwd=wish2pharm
```

This approach simply does not protect the password.

A BETTER WAY

An alternate approach, that is often used, involves the use of macro variables. This keeps the password information out of our program. In this approach you assign (hide) the macro variable definitions in your AUTOEXEC.SAS file.

In the AUTOEXEC you define:

```
%let uid=mary;
%let pwd=wish2pharm;
```

Once defined, any program that is to access the data base automatically has these macro variables available. The code becomes:

```
Proc SQL noprint;
  connect to odbc(dsn=dbprod uid=&uid pwd=&pwd);
  . . . . code not shown . . . .
```

Using this approach the LOG now shows:

```
pwd=xxxxx
```

At first blush it appears that we have solved the problem, however if the same program is executed with the system option SYMBOLGEN turned on, the values of the macro variables are displayed for all to see! As if this was not enough, a second security issue arises because of how the macro variables are stored.

Regardless of the status of the SYMBOLGEN option, because the macro variables are necessarily stored in the global symbol table, the execution of a simple %PUT statement will expose their values.

```
%put _global_;
```

Clearly, we need a better way to hide our passwords.

THE PROBLEM WITH MACRO VARIABLES

Macro variables are very tempting because retrieving their contents is easy. All we have to do is use the ampersand operator (&) with the macro variable name. For example in a DATA step we might,

```
data _null_;
  p = "&pwd";
  run;
```

If SYMBOLGEN is turned on, however, the LOG will clearly show

```
SYMBOLGEN:  Macro variable PWD resolves to wish2pharm
```

The very presence of the ampersand (a macro facility trigger) causes SYMBOLGEN to display the resolved macro variable. When macro variable contents are sensitive, we must be very careful not to use the ampersand. So if we can't use an ampersand, how are we to resolve a macro variable?

SILENT SYMGET SOLVES THE PROBLEM

It turns out that for our purposes there is a safer and better way to deal with macro variables. The DATA step function `symget` can also be used to resolve macro variables. Since `symget` does not use the ampersand, macro variables are resolved quietly without "tickling" the SYMBOLGEN processor. In this example,

```
data _null_;
  p = symget("pwd");
  run;
```

you will see nothing in the LOG, regardless of whether SYMBOLGEN is turned on or off.

Note carefully that we have demonstrated this function using a DATA step, however the SYMGET function can also be used in WHERE statements and clauses in virtually any PROC step including PROC SQL.

In a PROC SQL step we can use a WHERE clause to subset rows. In the following example we select only rows with name = "Judy". Notice that the constant text "Judy" is quoted in the WHERE clause.

```
Proc SQL;
  (
    SELECT name, sex, age, height, weight
    FROM sashelp.class
    WHERE name eq "Judy"
  );
  quit;
```

We could also place the name of interest into a macro variable (&UID) and then recover that name using the SYMGET function.

In the following PROC SQL example SYMGET is used to recover the value of macro variable `uid` and filter the sashelp.class table. The macro variable's name is quoted because it is simply being passed as an argument to the SYMGET function, and must not be interpreted as a variable in the data set sashelp.class.

```
* EXAMPLE: secretly seeking someone *;
%let uid = Judy;
Proc SQL;
  (
    SELECT name, sex, age, height, weight
    FROM sashelp.class
    WHERE name eq symget('uid')
  );
  quit;
```

It is very important here to notice the use of quotes. UID *is* quoted because it is the name of the macro variable, however notice that SYMGET('UID') *is not* quoted even though it resolves to constant text. In the previous example the variable NAME was compared to a quoted string. Here the quotes are unncessary because the parser already knows that the resolved value of the SYMGET function will be a string.

ISSUES AND CONCEPTS

The macro language is used to generate SAS code, consequently macro language elements are treated differently than are non-macro language statements. Since macro statements are high-level scripts whose actions and progress can be shown at each step of their execution process, it is easy to reveal macro variable values in the LOG. SAS statements (PROC's and DATA's) are compiled objects and their details cannot be shown as easily unless we design them to do so.

For the purposes of concealing sensitive information we need to write our code so that we

- Avoid storing passwords in code
- Avoid storing passwords in the global symbol table
- Prevent passwords from appearing in the LOG

Of course just identifying what we should not do is just the start. How do we avoid exposing sensitive information and still have it available for our use?

STEPS TO A SOLUTION

It turns out that there are several steps that we can take that will minimize our risk of surfacing our passwords or other sensitive information in the LOG or elsewhere. These steps include:

1. Use a re-entrant PROC such as SQL or DATASETS
   1.1. Executes many (more than 1) statements without quitting
   1.2. Bring all sensitive actions within scope of this single PROC
2. Because it doesn't quit, you/someone else can't interrupt it to turn on SYMBOLGEN, MLOGIC, etc.
3. Create macro variables within this PROC, and make them LOCAL to an enclosing macro
4. Use the silent `symget()` function to fetch the macro variables - avoid the ampersand!
5. Make sure you clear the LOCAL macro variables before the PROC terminates

THE SOLUTION

First, keep sensitive information in something other than code or the global symbol table. We recommend a SAS data set or other table. This table could itself be password protected for added security, which we will discuss later, however this might be a bit of overkill. An extreme example is presented in the Appendix.

In a separate program or in a separate utility a data set is created that contains our sensitive information:

```
data mystuff.passtab;
  format dsn uid pwd $8.;
  dsn='dbprod'; uid='mary'; pwd='wish2pharm'; output;
  dsn='dbprod'; uid='john'; pwd='data4you';   output;
  dsn='dbdev';  uid='mary'; pwd='hope2pharm'; output;
  run;
%let syslast=;
```

Clearing the automatic macro variable &SYSLAST prevents someone from using the _LAST_ automatic data set name to learn the name of our password table.

Second, wrap your pass-through SQL code with a macro. Make sure to do two things: ❶ Use a local macro variable to hold the password, and ❹ use the silent `symget()` function to retrieve it.

```
%macro secretsql(dbname, username);
  %local dd uu pp; ❶
  Proc SQL noprint nofeedback;
    (
      SELECT dsn, uid, pwd into :dd, :uu, :pp ❷
      FROM mystuff.passtab ❸
      WHERE dsn eq symget('dbname')
            AND uid eq symget('username')
    );

    connect to odbc(dsn=symget('dd') uid=symget('uu') pwd=symget('pp')); ❹

    create table mytable as select * from connection to odbc(
      . . . . your pass-thru SQL statement goes here . . . .
    );

    disconnect from odbc;
    quit;
%mend secretsql;
```

❶ The macro variables that we will create in ❷ are forced onto the local symbol table. This guarantees that these macro variables will be cleared after the termination of %SECRETSQL.

❷ The data base name (DSN), user id (UID), and password (PWD) values are stored in the macro variables DD, UU, and PP respectively.

❸ The names of any data tables listed in a FROM clause are masked from the LOG, unless `_method`, `_tree`, or `feedback` options are turned on. These options are not common.

❹ We retrieve the local macro variables containing our sensitive information by calling the `symget` function. As a result the LOG does not show the value of the macro variable, even if SYMBOLGEN has been turned on.
CAVEAT: Under some combinations of OS and version of SAS the `SYMGET` function does not execute inside of the `CONNECT` statement. As a result the macro variables are not resolved correctly. If this happens, you can replace the `SYMGET` with the `%SUPERQ` macro quoting function. The `CONNECT` statement becomes:

```
connect to odbc(dsn=%superq(dd) uid=%superq(uu) pwd=%superq(pp)); ❹
```

Notice that the macro variables are not quoted inside of the `%SUPERQ` function.

When invoking your SQL query, supply the database name and user name, and rest assured that your password stays safe and private.

```
%secretsql(dbprod, mary);
```

If a curious user enables macro debugging options before submitting your code, as in

```
options mprint mlogic symbolgen;
%secretsql(dbprod, mary);
```

they will only see the following messages in the LOG. Notice that `MLOGIC` prints the values of the macro parameters `dbname` and `username`, but does not show the values of the sensitive local macro variables.

```
MLOGIC(SECRETSQL):  Beginning execution.
MLOGIC(SECRETSQL):  Parameter DBNAME has value dbprod
MLOGIC(SECRETSQL):  Parameter USERNAME has value mary
MLOGIC(SECRETSQL):  %LOCAL  DD UU PP
MPRINT(SECRETSQL):   Proc SQL noprint nofeedback;
...
MPRINT(SECRETSQL):   quit;
MLOGIC(SECRETSQL):  Ending execution.
```

} no MLOGIC info shown during compilation of the PROC

EXTENSIONS OF THE CONCEPT

There are some other things that you may want to consider, either as extensions of the solution shown above or as general security concepts. Depending on how you write your code and how strict your security needs are, you may find some of the following concepts useful as well.

PROTECTING THE PASSWORD DATA TABLE

You may wish to also password protect the data table that contains the passwords. Because SAS supports password protection this is fairly easy. In the following example the DATA step that creates the password table has been assigned a password. This password cannot be more than 8 characters, though.

```
data mystuff.passtab (password=mydspwd);
  format dsn uid pwd $8.;
  dsn='dbprod'; uid='mary'; pwd='wish2pharm'; output;
  dsn='dbprod'; uid='john'; pwd='data4you';   output;
  dsn='dbdev';  uid='mary'; pwd='hope2pharm'; output;
  run;
```

Now when this table is used in %SECRETSQL the FROM clause becomes:

```
FROM mystuff.cred(password=mydspwd)
```

Other data set options that provide additional password protection include: encrypt=, read=, write=, and alter=.

A side benefit of the use of the PASSWORD data set option is that it can be used in a DATA step and SAS will mask the password in the LOG. Unfortunately when the same option is used in a PROC SQL step the password is not automatically masked in the LOG. However for the users of the %SECRETSQL macro the password will not be revealed in the LOG as long as the macro's source code is not displayed. The solution to this problem is discussed next.

MASKING %SECRETSQL SOURCE CODE

One of the problems with the use of the %SECRETSQL macro occurs when the macro is submitted for compilation. This takes place when you submit the %MACRO to %MEND statements that form the definition of the macro. The LOG shows the source statements for the macro, and while this does not show the passwords themselves, it **will** reveal the name of the data table that holds the sensitive password information and possibly the password needed to access the data table, if one has been assigned.

Since this is only a problem when the macro code is directly submitted, you can keep the macro source code out of the LOG altogether by placing the macro in either an AUTOCALL or Compiled Stored macro library, or by temporarily disabling macro source code printing.

FORCING NOMPRINT

If the MPRINT system option has been turned on prior to calling %SECRETSQL, the source code for the macro will be revealed in the LOG when the macro is executed. This is essentially the same problem as when the macro is compiled except now the LOG shows the name of the password data table when the macro is called. To prevent this we need to make sure that NOMPRINT is in effect. We can do this in the first couple of lines of the %SECRETSQL macro. The code becomes:

```
%macro secretsql;
  %local currmprint dd uu pp;
  %let currmprint = %sysfunc(getoption(mprint));
  option nomprint;
```

The GETOPTION function grabs the current setting for the MPRINT option. We save this option so that we can reset it at the end of %SECRETSQL. Then we make sure that NOMPRINT is in effect. At the end of our macro we can reset the option back to its original setting.

```
  options &currmprint;
%mend secretsql;
```

If SAS9.2 is available, you achieve the same result by using the /SECURE option on the %MACRO statement to temporarily disable MPRINT, MLOGIC, and SYMBOLGEN during macro execution. The macro statement becomes:

```
%macro secretsql/secure;
```

Now when %SECRETSQL is executed NOMPRINT, NOSYMBOLGEN, and NOMLOGIC will be temporarily set without our setting them ourselves.

THE PASSING ACCESS PARAMETERS THROUGH THE LIBNAME

Using the secret sequel concept in the LIBNAME statement is trivial by analogy. A DATA step which quietly assigns libref mylib is shown below. By "quietly" we mean that neither the libref nor its target location will be shown in the LOG -- unless, of course, one engages the DATA step `debug` option.

```
data _null_; a = libname('mylib', 'c:\temp'); run;
```

To quietly de-assign the libref

```
data _null_; a = libname('mylib'); run;
```

The libref can also be assigned through the use of an awkward but valid SQL select statement such as

```
(select libname('mystuff', "c:\tmp") from sashelp.class (obs=1));
```

or in a macro %LET statement:

```
%let rc = %sysfunc(libname(mystuff, c:\tmp));
```

Neither of these print any NOTES in the LOG, regardless of whether or not the NOTES option is turned on.

The %SECRETSQL macro may be re-written as a totally self-contained step *without a macro*.

```
Proc SQL noprint noerrorstop nofeedback;
  (select libname('mystuff', "c:\tmp") from sashelp.class (obs=1));
  (
    select pwd into :pp
```

```
  from mystuff.passtab                                                    ❶ Fetch
  where dsn eq symget('dsn') and uid eq symget('uid')
);
(select libname('mystuff') from sashelp.class (obs=1));

connect to odbc(dsn=%superq(dsn) uid=%superq(uid) pwd=%superq(pp));
create table ... as select * from connection to odbc (
  . . . . your passthru SELECT statement goes here . . . .          } ❷ Use
);
disconnect from odbc;

(select 'xxx' into :pp from sashelp.class (obs=1));                 } ❸ Clear
quit;
```

There are three parts to this concept: Fetch, Use, and Clear. In the first part ❶ we assign the *libref*, read the password, and then clear the *libref*. Notice that we must use a "dummy" SELECT statement to invoke the LIBNAME function, because every statement of this PROC SQL step must have valid SQL syntax.

The second part ❷ is the same as in the %SECRETSQL macro above. Just remember to use the silent SYMGET function or the %SUPERQ macro function to recover the password.

Most important is the third and final part ❸. Before the PROC terminates, we overwrite some junk into the sensitive macro variable. The `noerrorstop` option guarantees that this will always happen, regardless of whether previous statements were successful. Programmers often think about this last part during the handling of exceptions, as the unequivocal "finally" section of a try-catch block.

DOES THE (SELECT LIBNAME ...) STATEMENT REALLY WORK?

The "select libname..." statements in ❶ are very awkward. Why do they work? The SELECT clause can have scalar functions which are invoked once for each observation of data returned by tables fetched in the FROM clause. The scalar function can operate on any variable of the data set. But the scalar function *is not required* to act on a variable, and furthermore, any SAS function may be used as a scalar function. Two clauses are required to form a valid SQL statement: SELECT and FROM. Since we do not really care about variables or observations, or data sets for that matter, we choose a data set which is always present on every system: sashelp.class is a good choice. The `obs=1` option guarantees there will only be one row observation returned, and therefore the SELECT clause and its scalar functions are invoked only once.

CONCLUSION

It is essential that we do all that we can to protect our passwords. When running SAS programs that require passwords, we must take steps to insure that our private information is protected. Fortunately this can be done, and done fairly easily. However we must take the necessary steps ourselves.

ABOUT THE AUTHORS

Paul D. Sherman

Having developed the equipment and created the algorithms which measure and produce data, Paul Sherman finds that there is no better tool than SAS for working with especially large quantities of data. He brings into the SAS community fresh knowledge from over a decade of experience as test equipment and process engineer in the disk drive industry. Paul has been speaking at regional and national user groups, and was recently awarded "best paper" at PharmaSUG 2006.

Arthur L. Carpenter

Art Carpenter's publications list includes four books, and numerous papers and posters presented at SUGI, SAS Global Forum, and other user group conferences. Art has been using SAS® since 1976 and has served in various leadership positions in local, regional, national, and international user groups. He is a SAS Certified Professsional™ and through California Occidental Consultants he teaches SAS courses and provides contract SAS programming support nationwide.

AUTHOR CONTACT

Paul D Sherman
335 Elan Village Lane, Apt. 424
San Jose, CA 95134

(408) 383-0471
sherman@idiom.com
www.idiom.com/~sherman/paul

Arthur L. Carpenter
California Occidental Consultants
10606 Ketch Circle
Anchorage, AK 99515

(907) 865-9167
art@caloxy.com
www.caloxy.com

REFERENCES

Carpenter, Arthur L. 2004. *Carpenter's Complete Guide to the SAS® Macro Language, Second Edition*. Cary, NC: SAS Institute Inc. Book Code number 59224.

Heaton-Wright, Lawrence. 2003. "The SAS Data step/Macro Interface," Coder's Corner section of the *Annual Pharmaceutical Industry SAS User Group Conference*, May 4-7, 2003, Miami, FL. Paper cc069. http://www.lexjansen.com/pharmasug/2003/coderscorner/cc069.pdf

Li, Leiming. "A Process for Automatically Retrieving Database Using ODBC and SAS/ACCESS SQL Procedure Pass-Through Facility," Coder's Corner section of the *Twenty-fourth Annual SAS User Group International Conference*, April 11-14, 1999, Miami, FL. Paper 089-24. http://http://www2.sas.com/proceedings/sugi24/Coders/p089-24.pdf

Open Clip Art Library. http://www.openclipart.org

SAS Technical Support. "Using PC SAS Software to Read and Write to MS Access Database Files," Cary, NC: SAS Institute, Inc. Article number TS589C. http://support.sas.com/techsup/technote/ts589c.pdf

SAS Technical Support. "How to pass additional parameters to your ODBC data source," Cary, NC: SAS Institute, Inc. http://ftp.sas.com/techsup/download/sample/unix/access/odbc/odbc_connect_string.html

Williams, Christianna. 2004. "SYMPUT and SYMGET: Getting DATA Step Variables and Macro Variables to Share," Programming and Manipulation section of the *17th Annual Northeast SAS User Group Conference*, November 14-17, 2004, Baltimore, MD. Paper pm13. http://www.nesug.org/html/Proceedings/nesug04/pm/pm13.pdf

ACHNOWLEDGEMENTS

This paper was inspired by a conversation between the authors and William Moeglich.

TRADEMARK INFORMATION

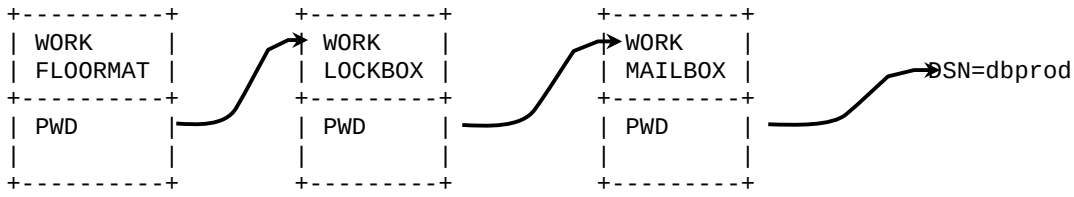
APPENDIX – PASSWORD PARADOX

For ultimate security, we need to limit access to the table of information which holds our passwords. The password table thus needs itself to have a password (and hopefully not the same one). Here lies a dilemma: How do we silently store and retrieve an access password which opens a table containing another access password?

Consider the following analogy. If you know you're not going to be home but wish to leave your house key in the mailbox for Mary, where could you safely leave the mailbox key for her? Everyone already knows to look under the floor mat and flowerpot.

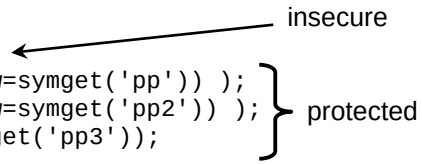
Under the public floor mat is the...	**lockbox key**	... which opens a lockbox holding a ...	**mailbox key**	... which opens a mailbox holding a ...	**database key**	... needed to access the data base.

Suppose there are three password access tables, two of which are access-protected.



The code following our "secret sequel" method described earlier is shown below. Regardless of SYMBOLGEN, etc., you will not find any passwords in the LOG. Nor will you see names of data sets.

```
%macro supersecretsequel;
  %local pp pp2 pp3;
  Proc SQL noprint nofeedback;
    ( SELECT pwd into :pp FROM work.floormat );
    ( SELECT pwd into :pp2 FROM work.lockbox (pw=symget('pp')) );
    ( SELECT pwd into :pp3 FROM work.mailbox (pw=symget('pp2')) );
    connect to odbc(dsn=&dsn. uid=&uid. pwd=symget('pp3'));
    . . . . pass-thru sql statement not shown . . . .
    disconnect from odbc;
    ( SELECT 'x','x','x' into :pp, :pp2, :pp3 FROM ... );
    quit;
%mend supersecretsequel;
```



No matter how many secure levels, layers, or mailboxes you design in to your access model, the beginning point (the floor mat) will always be insecure. The question is, how can we get a private password into the proc to look under the floor mat, without using a data set or global macro variable? In the scenario with a series of keys the real security is in knowing how each key is to be used and in which order. In the corresponding code, that knowledge has moved from our brain (the ultimate secure location) to the code itself.

At present there is no unattended batch-mode solution to the password paradox, although we believe that an approach following public key cryptography might provide useful insight.

```
%macro supersecretsequel;
  %local pp pp2 pp3;

  %window getpswd
    #2 @5 "Enter the name of the Password Data set of Interest"
    #3 @5 pwdata attr=underline required=yes
    #4 @8 '(type name and press enter)'
  ;
  %display getpswd;

  Proc SQL noprint nofeedback;
    ( SELECT pwd into :pp FROM work.floormat );
    ( SELECT pwd into :pp2 FROM work.lockbox (pw=symget('pp')) );
    ( SELECT pwd into :pp3 FROM work.mailbox (pw=symget('pp2')) );
    connect to odbc(dsn=&dsn. uid=&uid. pwd=symget('pp3'));
    (. . . . pass-thru sql statement not shown . . . .);
    disconnect from odbc;
    ( SELECT 'x','x','x' into :pp, :pp2, :pp3 FROM ... );
    quit;
%mend supersecretsequel;
```