Paper 103-27

Simplifying SAS® Security

Derek Morgan, Washington University Medical School, St. Louis, MO
Michael Province, Washington University Medical School, St. Louis, MO

Abstract

An increased emphasis on the security of data within the organization has led to the use of the security built into the SAS® System. The encryption and passwording of the datasets used for this project was relatively easy to accomplish. However, day-to-day uses of this data range from analysis to data entry to ad hoc query. How do we allow all the different users access without hard coding the passwords, or forcing them to continually type in passwords? By using macro variables to carry the passwords.

The Method

Creating a SAS system table with the built in security is easy. There are three levels of password protection: read, write, and alter. You can employ one, two, or all three on any given dataset. In addition, you can encrypt a dataset with the simple dataset option ENCRYPT=YES. The datasets for this project are encrypted, and use all three levels of passwords.

Creating an Encrypted Dataset with Passwords

```
DATA foo.bar (ENCRYPT=YES READ=foo
              WRITE=bar alter=foobar);
LENGTH id 5 v1-v40 3;
STOP;
RUN;
```

Any time a dataset needs to be opened for access, the password for the desired level of access needs to be specified. Only the maximum level of access necessary needs to be specified for a given dataset request.

Accessing a Dataset with Passwords

```
PROC MEANS DATA = foo.bar (READ=foo);
RUN;

PROC FSEDIT DATA=foo.bar (WRITE=bar)
             SCREEN=project.screencat.foo;
RUN;

DATA foo.bar (ENCRYPT=YES READ=foo
              WRITE=bar ALTER=foobar);
UPDATE foo.bar (READ=foo) newfoobar;
BY ID;
RUN;

PROC SORT DATA=foo.bar (ALTER=foobar);
RUN;
```

A protected/encrypted dataset will always need to have a password associated with it in a PROC or DATA step. While it is possible to make non-encrypted work copies from the master datasets, sometimes this will take more work or more resources than are available. No matter what, you will still need to open the original dataset with its read password.

MACROing Your Way to Security

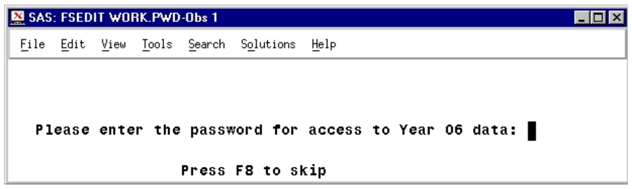
First, each project has its own set of passwords. All the datasets for a given project use the same passwords. This allows us to use macro variables to represent each of the passwords. Now, instead of being forced to type the passwords in our SAS programs (where anyone who gains access to that ASCII file will have the passwords to the datasets in that program) we use:

&READPASS – Read-level password.
&WRITPASS – Write-level password.
&ALTRPASS – Alter-level password.

In this way, any programmer can write code without knowing the specific password for any dataset, and the passwords are not immediately obvious. However, if you put %LET READPASS=foo; at the top of your programs, any security improvements using macro variables will be negated. The key is in how these macro variables are assigned. We have several choices for this.

Let's take the case of a programmer developing a program interactively on the SAS System under UNIX or for Windows. For them, it would be ideal if they didn't have to worry about continually typing passwords each time a dataset was accessed. However, using "%LET READPASS=foo;" at the top of a program does not yield much in the way of increased security.

The approach we have chosen is to execute an interactive application inside of the AUTOEXEC.SAS file associated with the project. This specific application uses the FSEDIT component of the SAS/FSP® product, but any interactive facility within the SAS System will do. Whenever the SAS System is initialized, this is what comes up on the screen:


SAS: FSEDIT WORK.PWD-Obs 1
File Edit View Tools Search Solutions Help

```
Please enter the password for access to Year 06 data:

                Press F8 to skip
```

There are actually two fields on this screen; there is an "invisible" message field named "MSG" above the line where the user enters a password. The pad character for that line is set to blank, and the color is red. The user is prompted to enter an access password (which is NOT the same as any of the passwords for the data tables.) The code below will detail how the password application functions from this point.

SCL for Password verification

```
1   init:
2   control label;
3   cursor passwd;
4   rc = rc;
5   return;

6   main:
7   return;

8   passwd:
9   if passwd ne " " then do;
10     fn = "secure.t4b5d9l3 (read=" ||
             passwd || ")";
11     pwfil = open(fn);
12     if pwfil then do;
13        rc = fetch(pwfil);
14        call symput('readpass',
                      getvarc(pwfil,1));
15        call symput('writpass',
                      getvarc(pwfil,2);
16        call symput('altrpass',
                      getvarc(pwfil,3);
17        rc = close(pwfil);
18        call execcmd('end');
19        return;
20     end;
21     msg =
     "Password incorrect... restart SAS session";
22     call wait(3);
23     call execcmd('end');
24     return;
25  end;
26  msg = "Please enter a password";
27  cursor passwd;
28  return;
29
30  term:
31  return;
```

When the user presses the ENTER key, the application checks to make sure that they have entered something in the password field. If the password field is not blank, then the SCL program attempts to open the password table, which is itself encrypted and therefore password-protected. The passwords for the password table are not the same as those for the project data.

The password table can only be opened if the password entered in the field is correct. If it is incorrect, the message line will display a message (line 21), there will be a pause, and the password application will terminate. At that point, the user is returned to the interactive SAS System session, but the macro variables that should contain the passwords for access to the data are blank.

If the table can be opened, the password record is loaded into the program data vector, and CALL SYMPUT is used to fill each macro variable with its corresponding password. The password table is then closed, the password application is terminated (line 18), and the user is returned to the interactive SAS System session.

Now, in order to access the project data from code,all that is required is to substitute the macro variable name for the password. Using the original example on page 1, we will now have:

Accessing a Dataset with Macro Passwords

```
PROC MEANS DATA = foo.bar (READ=&READPASS);
RUN;

PROC FSEDIT DATA=foo.bar (WRITE=&WRITPASS)
                  SCREEN=project.screencat.foo;
RUN;

DATA foo.bar (ENCRYPT=YES READ=&READPASS
              WRITE=&WRITPASS
              ALTER=&ALTRPASS);
UPDATE foo.bar (READ=&READPASS) newfoobar;
BY ID;
RUN;

PROC SORT DATA=foo.bar (ALTER=&ALTRPASS);
RUN;
```

As you can see, now there are no exposed passwords in the code. This works well for any SAS System process that is started manually. Since we do not have any automatic batch jobs, this is sufficient for our purposes. The closest we have actually come to needing an automated method for password implementation are in our UNIX script jobs. This situation is handled by invoking the script using the password table's password as a parameter.

>monthly.report rosebud

The AUTOEXEC.SAS checks for the presence of a UNIX parameter using %SYSGET. If the %SYSGET parameter exists, then the AUTOEXEC.SAS will execute this code instead of the interactive application:

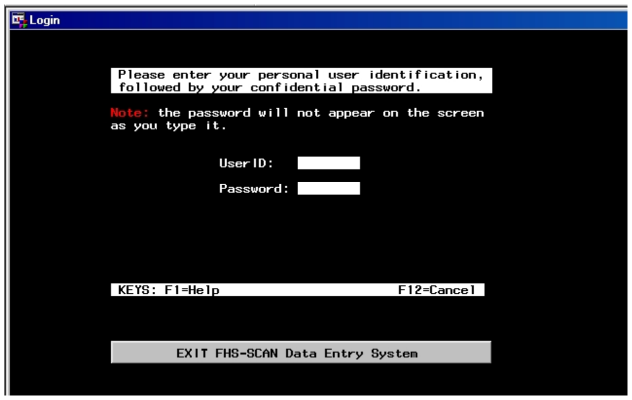
```
%LET batchvar=%SYSGET(parm1);
DATA _NULL_;
SET secure.t4b5d9l3 (READ=&batchvar);
CALL SYMPUT('readpass',ereaaede);
CALL SYMPUT('writpass',iwiriiitiei);
CALL SYMPUT('altrpass',aalataeara);
RUN;
```

This way, each time a new SAS System job starts, the user is not queried for the password.

Security Within A SAS System Application

Another situation where we need security is within our data entry and management applications. These are generally Windows-based SAS/AF® and SAS/FSP applications, but the methods can be extended to other platforms.

Again, we use macro variables to carry the passwords through the system while it is in use; however, there are several lines of defense against unauthorized data access. The first is the operating system security itself, with user identification and authorization allowing access to the computer. Next, the application has its own user identification and authorization facility. This program module is the entry point into the application.



Please enter your personal user identification, followed by your confidential password.

Note: the password will not appear on the screen as you type it.

UserID:

Password:

KEYS: F1=Help F12=Cancel

EXIT FHS-SCAN Data Entry System

The relevant SCL for this login screen is as follows:

```
INIT:
call setcr('stay','noreturn','modify');
control enter error allcmds label;
length passwd $8;
nerror=0;
errlim=3;  /* Set maximum # of tries */
passwd=_blank_;
msg=_blank_;
slevel=.;
return;

MAIN:
erroroff _all_;
cmd=upcase(lastcmd());
if cmd eq 'END' then
   call nextcmd();
if cmd eq 'CANCEL'
then userid='CANCEL';
if (upcase(userid) eq 'CANCEL') or
   (upcase(pass) eq 'CANCEL') then do;
if secure ne . then
   call close(secure);
status_='H';
return;
end;
return;

USERID:
msg=_blank_;
cursor pass;
return;

PASS: /* check the id and password */
secure=open('gcx.secure(read=sasisfun)','i');
call set(secure);
lrc=locatec(secure,varnum(secure,'userid'),
            userid);
call close(secure);
secure=.;
```

```
      /* userid and password valid?  */
if (lrc ne 0 and pass eq passwd) then do;
   call sympput('userid',userid);
   call symputn('slevel',slevel);
   call sympput('x1pass',mangia');
   call sympput('x2pass','molto');
   call sympput('x3pass','bene');
          /*--proceed to main menu  */
   call display('gc.programs._main.program');
 /* on return, clear the macros set above */
   call sympput('userid','');
   call symputn('slevel',.);
   call sympput('x1pass',' ');
   call sympput('x2pass',' ');
   call sympput('x3pass',' ');
   msg = _blank_;
   userid=_blank_;
   pass=_blank_;
   call nextcmd();
   cursor userid;
   return;
end;
else do;  /* invalid userid or password */
   alarm;
   erroron msg;
   nerror=nerror+1;
/* send retry message to userid and/or exit */
   userid=_blank_;
   pass=_blank_;
   cursor userid;
   if (nerror eq errlim-1) then
      msg='One more try before exiting.';
   else
      msg='Userid or Password is invalid.';
   if (nerror ge errlim) then do;
      call execcmd('cancel');
      return;
   end;
end;
return;

QUIT:
if secure ne . then
   call close(secure);
_status_='H';
return;

TERM:
if secure ne . then
   call close(secure);
return;
```

Here's How The SCL Works

Line 5 sets the maximum number of tries before the login screen automatically terminates, and the user will have to re-start the application from the desktop. Line 29 opens the security dataset where the userids and passwords are stored. This dataset is encrypted, and the passwords are not the same as the passwords for the other datasets. The userid/password combination entered by the user is compared to the values stored in the security database. If they match (lines 34-39), the macro variables are loaded with their correct values, and the main menu for the system is displayed (line 40). When the user closes the main menu to exit the application, all of the security macro variables are set to blank (lines 41-45), and the application terminates.

If the userid/password combination does not match the values stored in the security dataset (lines 53-68), the system clears the userid and password fields, prompts the user to try again, and counts an incorrect try (line 56). If the user has one more try left before the application closes, the program displays a message to that effect. If the maximum number of tries has been reached (lines 64-67), then the program (and the application) is halted. Otherwise, the user is allowed to re-enter a userid/password combination.

This is the only location in the application where the data passwords are hard-coded. The security dataset passwords are also hard-coded in the modules that allow a user to change his or her password, and those the system manager uses to add or remove other users. What's to prevent someone from opening up these program modules and getting the passwords that way? These modules are distributed without the source code. If desired, the entire application could be distributed without any source, although that may cause problems while debugging at the client site. You can remove the source code from program modules using the following:

```
PROC BUILD CAT=libname.nocodecatalog BATCH;
MERGE CAT=libname.codecatalog NOSOURCE;
RUN;
```

This will copy the contents of the modules in CODECATALOG to NOCODECATALOG without the source code.

Accessing Encrypted Datasets Within the Application

When a dataset needs to be opened inside the application, any passwords are attached by using the SYMGET() function.

```
dsn = "gcx.register (read=" || symget('x1pass')
       || ")";
register = open(dsn,'i');
```

This example stores a full SAS dataset name with options in the variable DSN, which is then used in the OPEN() function.

Using macro passwords inside of a SUBMIT block is a little more complicated, requiring a separate macro variable to hold the password code:

```
call symput('klooge',"(read=" ||
             symget('x1pass') || ")");
submit continue sql;
create table address as
select a.id, a.name, b.*
from tempy as A, gc1.hs2 &klooge as B
where a.id=b.id;
endsubmit;
```

The macro variable &KLOOGE is used to store the syntactically correct password option, so that it is resolved when it is passed to the SAS System during the execution of the SUBMIT block. Of course, for full security, &KLOOGE is blanked immediately after it has been used.

There is an important reason why the SYMGET() function is used instead of just the simple macro reference in the code. A simple macro reference will be resolved at the time the program module is compiled, not at program execution. If you have all the macro variables correctly assigned in the SAS session while you are compiling these modules, then the code will work. Let's use the access example from earlier, but instead of the SYMGET() function, let's just put the macro reference directly in the code.

```
dsn = "gcx.register (read=&x1pass)";
register = open(dsn,'i');
```

If the macro variables are blank, DSN will have the value "gcx.register (read=)" because &X1PASS is blank; therefore, the OPEN() function will fail. SYMGET() works during program execution, so as long as the macro variables are defined during program execution, the code will work.

What about %PUT _ALL_?

Yes, this will dump all the macro variables, along with all of their values to the SAS System log. However, as long as the application is not active, none of the password macro variables are defined, and while their names may be revealed, their values will not. You still need to log into the application to gain access to these passwords via this method. The security dataset passwords are hard-coded in source code that is missing, so it becomes a matter of hacking the userid/password combination, not cracking the macro variable password scheme.

Summary

Leaving passwords to SAS System data tables in open code negates any security enhancement that using passwords and/or encryption provides. Macro variables can be substituted in the code for the passwords, and there are different methods for assigning the correct values to the macro variables. An interactive SCL-based application is one way of handling this assignment, or you can use the less secure method of passing an environment variable to the program. Either way, the program code itself will only have references to macro variables instead of passwords.

Within SAS System-based applications, the same macro method of providing passwords can be used. In order for this to work, access to the application itself should be guarded by a combination of userid and password. These are stored in a separate encrypted dataset with its own set passwords. Once access to the application has been granted, the passwords are assigned to macro variables, and these macro variables are used whenever dataset access is required. The SYMGET() function insures that the resolution of macro references takes place during the execution of the application, not during the compilation of the macros.

While this method can be easily defeated with “%PUT _ALL_;”, it does make it more difficult for an unauthorized person to gain access to data.

This work was partially supported by NHLBI grants
HL 54473 and HL 47317.

Further inquiries are welcome to:

Derek Morgan
Division of Biostatistics
Washington University Medical School
Box 8067, 660 S. Euclid Ave.
St. Louis, MO 63110
Phone: (314) 362-3685   FAX: (314) 362-2693
E-mail: derek@wubios.wustl.edu

This and other SAS System examples and papers can be found on the World Wide Web at:

http://www.biostat.wustl.edu/~derek/sasindex.html