Protecting Macros and Macro Variables: It Is All About Control

Eric Sun, sanofi-aventis, Bridgewater, NJ
Arthur L. Carpenter, CALOXY, Anchorage, AK

ABSTRACT

In a regulated environment it is crucially important that we are able to control, which version of a macro, and which version of a macro variable, is being used at any given time. As crucial as this is, in the SAS® macro language this is not something that is easily accomplished. We can write an application that calls a macro that we have validated, but the user of the application can write and force the use their user written un-validated version of that same macro. Values of macro variables that we have populated can be 'accidentally' replaced by user written assignments. How can you guarantee that the end results are accurate if you cannot guarantee that the proper programs have been executed?

Although our tools are limited, there are a few options available that can be used to help us control our macro execution environment. These, along with management programs, can give the application developer *better* control, and *greater* protection, during the execution of the application.

For a successful macro language application, it is all about CONTROL!!

KEYWORDS

Reverse engineering, User Macro, Macro Compilation, Stored compiled Macro Libraries, Macro Protection, Macro variable collisions, MSTORED, MREPLACE, MCOMPILE, SECURE

INTRODUCTION

In the SAS programming world, a great deal of pleasure and creative satisfaction can be derived from the use of the SAS Macro Language. SAS itself provides various utilities to enrich the flavor and sophistication of SAS programming, especially SAS based application development. As part of the SAS programming language, a SAS macro is compiled and protected under its configuration and installation, but at a loose side so maximum flexibility to the users can be preserved. However the SAS application creation in the pharmaceutical industry requires a more robust design and stricter control to meet the rigid regulations, especially those used for drug submission to regional and national authorities.

When control related issues are discussed by experienced SAS programmers, common topics include:

- What version of a macro is being executed?
- How can we control for the correct version?
- How do we avoid macro variable collisions; and how do we protect our macro variables, compiled macros, and the source code?

Some of these issues become the roadblocks that tend to discouraged application developers. On-the-other-hand these same challenges have spawned a number of SAS conference papers, with a number of innovative solutions. In this paper we would like to focus on these points and examine some practical approaches and solutions. Some remain unresolved and we invite you to contribute to the conversation.

In this paper, the following points are discussed and shared:

- How macros are compiled.
- How Macro (library) search order is defined and how it can be protected.
- What is a macro variable collision?
- How can the version of the macro be managed and secured?
- How can we avoid macro reverse engineering?

Control Issues – What We Need

When executing applications in a controlled environment, we need to know which macros are being executed, and even more importantly, which version of the macros and in what version of SAS, if running in a multi-version platform. We need to be able to certify that when a given macro is called, the version that we have validated is the one being used. Sometimes because of regulation or business SOP, we may also need to verify that the user is qualified to execute a specific version of a macro. Since our users are also capable of writing their own macros, how do we know, truly know, that they have not replaced our macro with one of their own, intentionally or not?

Using stored compiled macro libraries is a part of the solution, but only a part. The source code of the macro must also be stored and protected, and we may also need to make sure that the macro itself cannot be re-engineered if not for all, at least the core portion that handles the CONTROL.

Similarly during the execution of our application, we generate macro variables with values that will be needed by subsequent operations within the application. How do we ensure that the value of that macro variable has not been altered by a user macro variable assignment? Or if we are unable to prevent macro variable value reassignment, how can we have our macro memorize its values and even restore them as needed?

How do we control our macro definitions and our macro variable value assignments?

The first level of control is trust, trust in professionalism. Unfortunately this is not simply a matter of professionalism, or of controlling those with nefarious objectives. Mistakes and accidents do happen, and we cannot simply trust that they will not happen to the users of our application.

Macro Compilation and Macro Library Review

There are two primary types of macro libraries, Autocall and Stored Compiled Macro Libraries. By default the autocall macro facility is available to most companies and groups that utilize SAS. Stored Compiled Macro Libraries, on-the-other-hand, are only available when turned on with the MSTORED system option. The code at the left turns on and allocates, both an autocall library and a stored compiled macro library. These two simple libraries will be used throughout this paper. Your libraries can be much more complex with multiple levels of locations for both types of libraries. Although the SASMSTORE option accepts only one *libref,* the library associated with that *libref* can be a concatenated or composite library.

```
* Specify the autocall library;
filename autolib "&path\autocallmacros";
options mautosource
        sasautos=(autolib sasautos);

* Specify the stored compiled macro library;
libname complib "&path\storedmacros";
options mstored
        sasmstore=complib;
```

A macro is defined through the use of the %MACRO and %MEND statements. When these statements are executed, the macro facility performs a macro compilation. This is not really a true compilation, and is little more than a check on macro syntax. The compiled macro definition is then written to a SAS catalog with an entry type of MACRO. The default catalog is WORK.SASMACR and the entry name will be the name of the macro itself. If a stored compiled macro library is available, the /STORE option on the %MACRO statement can be used to direct the compiled macro to a permanent SASMACR catalog with a different *libref* from WORK.

```
%macro abc;
   %let dsn = 5;
%mend abc;
```

When a macro, as an example we can use the %ABC macro, is called, SAS must search for the macro's definition. SAS first looks for the ABC.MACRO entry in the WORK.SASMACR catalog. Then, assuming that it is not found in the WORK catalog, and if stored compiled macro libraries are turned on, a search is made for the ABC.MACRO entry in each SASMACR catalog in the *libref* designated by the SASMSTORE system option. Finally if a compiled entry has not yet been found, SAS starts a search in the autocall library locations for a program with the name of ABC. This process and the use of macro libraries has been described in detail in Carpenter (2001 and 2004).

Understanding this search order is crucial to understanding the dilemmas associated with protecting and utilizing a specific version of a SAS Macro. In summary the search order is:

1. WORK.SASMACR
2. stored compiled macro libraries   COMPLIB.SASMACR
3. autocall macro libraries

Moreover, the name of the compiled SAS catalog, **SASMACR,** is the 'only' catalog name will be recognized in the specified SASMSTORE library or libraries. That means in a multi-version maintained macro application, certain macro calls (catalog entries) with the same names across versions cannot be preloaded until later determination from a version control mechanism. In that case, the correct compiled 'Version Macro' of the application should be loaded into WORK library (the loading of stored compiled macros into the WORK.SASMACR catalog is discussed below).

Macro Variable Collision Review

A macro variable collision occurs when a macro variable is written and its value unintentionally overwrites an existing value. Usually this happens when we create a macro variable within a macro and it is written to a higher symbol table.

```
%let dsn = mydata.clinics; ❶
%macro abc;
%let dsn = 5; ❷
%mend abc;
%abc
```

In the example to the left, the macro variable &DSN ❶ will be written to the global symbol table. Generally when a macro variable is created within a macro ❷ it will be written to the local symbol table, however since &DSN already exists in the higher table, there will be no entry for &DSN in the local symbol table for %ABC, and instead the value 5 will replace the data set name in the value for &DSN in the global table. The two variable values have collided.

If our application uses macro variables to pass information the values of those macro variables must be protected. See Carpenter (2005) for more detail on macro variable collisions.

CONTROLLING MACRO LIBRARIES

One of the keys to controlling macro execution is controlling the macro libraries and how they are accessed. Let us suppose that the macro %DEF is very important to our application. It has been tested and validated and we want to make sure that the users of the application will have access to our version of the %DEF macro. To make it available we place it in our stored compiled macro library using the /store option on the %MACRO statement.

```
%macro def / store;
%put Stored compiled Version of DEF;
%mend def;
```

The %DEF macro is now ready for use, but what are our guarantees and what might go wrong? A user could, by either mistake or calculation circumvent the use of our macro by writing their own version of the %DEF macro and:

- using the /STORE option, overwrite your version in the stored compiled library (this is a worst case scenario, because it affects everyone else using the library).
- place it's definition in the autocall library
- execute the definition without the /STORE option placing the compiled macro in the WORK.SASMACR catalog.

The problem has been compounded by the search order for the macro definition. WORK.SASMACR is searched first and a user defined macro will be found there before our version is seen in either of the libraries.

What We Have

There are a few system options that provide some control and protection for our macros. These options are a combination of system and %MACRO statement options. Along with these options there are some techniques that we can apply to bolster our control and confidence.

Protecting Source Code

Macro source code can be surfaced in a number of ways. The most common is through the use of the MPRINT system option. Even when NOMPRINT is in effect, a technique attributed to Ian Whitlock, which uses the %PUT and a macro quoting function, will write the inner workings of a macro to the LOG.

```
%put %quote(%abc);
```

Using the /source option on the %MACRO statement, it is possible to store your macro's definition, the code itself, in the catalog along with the compiled macro. This code can then be reclaimed from the catalog using the %COPY statement. Of course if your users can see the source code for your macro, they can then re-engineer your macro.

```
%macro def/store source;
```

Regardless of whether or not the /SOURCE option was used on the %MACRO statement it is still to some extent possible to reclaim some of the original macro definition. A DATA _NULL_ step can be used to write the hex codes of the macro's definition to the LOG. The /SECURE option on the %MACRO statement prevents even this partial recovery of the code by encrypting the compiled macro definition.

```
filename maccat catalog
        'complib.sasmacr.def.macro';
data _null_;
   infile maccat;
   input;
   list;
   run;
```

```
%macro def/store secure;
```

Use of the /SECURE option (with or without the /STORE option) causes the system option values of NOMPRINT, NOMLOGIC, NOSYMBOLGEN to be temporarily set during the execution of the macro. The /SECURE option in the example to the left resets the MPRINT system option to NOMPRINT during the execution of %DTEST. After %DTEST has completed execution, the value of MPRINT is restored as the system option value.

```
%macro dtest/secure;
proc print data=sashelp.class;
   run;
%mend dtest;
options mprint symbolgen mlogic;
%dtest
```

The /SECURE option helps us to keep the source code of our validated macro out of the hands of those who may wish to re-engineer our code.

Unfortunately this is not the only way that the use of our version of the macro can be circumvented.

In a short article written for the Views newsletter Murphy Choy (Choy, 2011) suggests the use of an encrypted SAS data set as the storage mechanism for macro code.

Macro Compilation Options – A Couple of Big Hammers

Although we can conceal the source code for our compiled macro, the user could still circumvent the use of our version of the macro by writing his/her own macro from scratch, and then force its use in preference to the validated version, using one of the following techniques:

❸ In the program being executed, use the %MACRO to %MEND without the /STORE option. This writes the macro definition to WORK.SASMACR, where it will be selected first.

❹ By placing a dummy macro definition (%GHISNEAKY), in the autocall library, but with the twist that the code contains a second definition of a controlled macro. When the dummy macro is called, it will not yet have been compiled and will not yet exist in WORK.SASMACR. Consequently the autocall library code is executed. Since it contains the definitions of two macros, both will be compiled. The second controlled macro (%GHI) definition will replace our validated version of %GHI and all subsequent calls to %GHI will use the new version.

```
*ghisneaky.sas;
%macro ghisneaky;
%put this is a sneaky macro;
%mend ghisneaky;

%macro ghi;❹
%put modified ghi from sneakyghi;
%mend ghi;
```

❺ If the %MACRO statement in the user's macro is written with the /STORE option, the user's version will overwrite the validated version of the macro in the stored compiled library.

```
options nomcompile;

* attempt to compile an autocall macro;
* The macro compiles!!!; ❹
%ghi

* Macro MYGHI does not compile; ❸
%macro myghi;
%put compile from within program;
%mend myghi;
%myghi

* Included definitions do not compile; ❸
%inc "&path\autocallmacros\ghi.sas";
%ghi

* The macro is not stored or compiled; ❺
%macro storeghi / store;
%put macro was compiled and stored;
%mend storeghi;
%storeghi
```

The system options NOMCOMPILE and NOMREPLACE are partial solutions to these circumventions, however they are not without side effects, and of course like other system options, protection of the option's value itself is outside of our control. NOMCOMPILE prevents the compilation of new macro definitions and NOMREPLACE restricts the storage of a new compiled version of a macro that has already been compiled.

```
options nomreplace;

* Compile the autocall macro GHI;
* Definition is stored in WORK.SASMACR;
* (entry GHI.MACRO does not already exist);
%ghi

* Unauthorized version of GHI
* does not replace version from the
* autocall library; ❹
%macro ghi;
%put Unauthorized version of GHI;
%mend ghi;
%ghi
```

NOMCOMPILE prevents the compilation of new macros, but does not prevent the use macros that are already stored in a library. This option is a solution to ❸ and ❺. However it also prevents the user from compiling new macro definitions. Notice also that macro definitions stored in the autocall library are not affected by NOMCOMPILE.

The NOMREPLACE system option is used to prevent the replacement of a compiled macro in WORK.SASMACR. It has no affect on stored compiled macro libraries, and therefore offers us no protection against the scenario where the user overwrites the official version of the macro in the stored compiled macro library ❺.

When used together these two system options give us some protection for our authorized macro versions from the three cases shown above. Unfortunately the use of NOMCOMPILE can severely limit the use of user written macros.

```
%macro copysasmacr/store;
proc datasets nolist;
   copy in=complib
        out=work
        mtype=cat;
      select sasmacr;
   quit;
%mend copysasmacr;
```

Since one of the biggest issues is that the WORK.SASMACR catalog is searched first, this is the catalog on which we need to focus. If we were to copy all the macro definitions in our stored compiled macro library to WORK.SASMACR at the very start of the program/application, the NOMREPLACE option would be protecting our authorized macro definitions! The macro COPYSASMACR can be placed in our library and then called in the autoexec.sas program, before WORK.SASMACR exists. With the NOMREPLACE option (starting in SAS9.2) in effect, but not NOMCOMPILE, users will be free to create their own macros, but not macros with the same name as any of our validated macros.

This catalog copy using PROC DATASETS will fail if the WORK.SASMACR catalog already exists, however PROC CATALOG has some other capabilities that will allow us to copy all or part of a stored compiled library to the WORK.SASMACR catalog even if it already exists (see the %VERCOPY macro below).

Detecting Macro Location

Another issue associated with version control is the macro location. Since the definition of a given macro can reside in any of a number of locations:

- the calling program
- a program that already stored the compiled macro in WORK.SASMACR
- an autocall library
- a program that stored the compiled macro definition in the stored compiled macro library.

To further complicate the issue, the autocall library can include multiple locations and the stored compiled macro library can be a concatenated catalog. The definition of a given macro can even reside in multiple locations, and SAS will execute the first version encountered. When we execute the %JKL macro how do we know which version was used?

When the MAUTOLOCDISPLAY system option is turned on, the physical location of any macro, which has a definition derived from the autocall library, is printed to the LOG for each macro execution. Only macros originating in the autocall library will have their location printed. The location of macros executed from either WORK.SASMACR or the stored compiled macro library will not be shown. There is no option for showing the library location for compiled macros.

There is a partial work around for our situation of needing a stored compiled macro library *and* the desire to print the location of the macro definition in the LOG. First establish a level of the autocall library that is reserved for the definitions of the validated macros. You may wish to restrict write access to this location. Then make sure that each %MACRO statement in the autocall library has the /STORE option. Once these macro definitions have been compiled, they will be stored in the stored compiled macro library. When the macro executes, the MAUTOLOCDISPLAY will write the appropriate autocall location to the LOG. Any location, other than the one containing your validated macro definitions, will indicate a version of a macro that has not been approved. The caveat is that the MAUTOLOCDISPLAY option will not display the location of a macro compiled in a previous session. It will, however reveal the location compiled in the current session, and if that macro is one of your controlled macros, but with an incorrect location.

Putting It All Together

The following steps summarize the above discussion to provide maximum version control over the macros that you have written.

When building the macro definitions:

- Establish the autocall and stored compiled macro libraries.
- Store your macro definitions in a secure level of the autocall library including the use of the /STORE and SECURE %MACRO statement options.
- Compile each of these macros into the stored compiled macro library
- You may wish to restrict write access to the stored compiled macro library location at the OS level.

When the user of the application is going to access your controlled macros:

- Establish the autocall and stored compiled macro libraries.
- In the autoexec.sas copy all of your controlled macro definitions from the stored compiled macro library into WORK.SASMACR
- Turn on the NOMREPLACE and MAUTOLOCDISPLAY system options.
- You may wish to write a program to harvest the definition locations written to the LOG by MAUTOLOCDISPLAY

What to protect and what to hide

In the real world of SAS Macro programming, there are situations where a Macro Library is created for use by an expert user community. Over protection will hurt the understanding and confidence of the user community to use the macro tool, especially as its use becomes more complicated or difficult. Therefore the purpose, implementation, scope, and control of a macro application should be carefully defined. Some application design principles might include:

- Fully utilize any available security mechanisms and privileged application account provided at the Operating System level (e.g. Unix, Win, Open VMS) to protect the environmental configurations including OPTIONS/GOPTIONS.
- Critical configuration values, such as those mentioned above, should be assigned in a macro in the compiled catalog. Modifications to the standard values can then be determined through the use of tools like SASHELP.VOPTION or DICTIONARY.OPTIONS. This allows the application to perform cross checking of the current values versus what are needed for the normal execution of the macro application.

You may also need to hide control codes or settings used by your application. The protection of passwords and userids is discussed by Sherman and Carpenter (2007). Many of the macro contents can be revealed at execution using combinations of the system options MPRINT, MLOGIC, and SYMBOLGEN. Since these options write to the LOG, one solution is to remove portions of the LOG with a mixture of PROC PRINTTO and a temporary catalog.

```
%macro prep ( );
       :
       :
/* Checking SAS License info from User Machine */
proc printto log=work.tmp.dump.source❻;   run;
       proc setinit; run;
proc printto;  run;

filename SelfKill catalog "work.tmp.dump.source" ;
data _null_; infile SelfKill length=long ;
     <<<Parsing SAS Version, SITE info etc.>>>
     <<<is written to macro vars           >>>
    run ;

filename SelfKikll  clear ;
proc datasets lib=work memtype=catalog nolist ❼;
      delete tmp ;
run ;
quit;
       :
       :
%mend prep;
```

Here PROC PRINTTO is used, while the control values are being set, to write the LOG to a SAS catalog in the WORK location ❻,. Later when it is no longer needed the catalog containing the LOG is deleted ❼. Using this technique, you can hide sensitive macro code, and therefore reduce the vulnerability of your macro application.

Design Considerations for a Protected Macro Application

When building your own macro library or application, you will need to think about the levels and kinds of protection that your application will require. The following are some practical solutions that can be used to construct a self protected system with expandable architecture and intelligence that will control what the non-developer will be able to see.

Version Control: To achieve further protection from user defined macros that are written to the WORK.SASMACR which will supersedes our macro definition in the SASMSTORE library, we can implement a Master/Slave hierarchical library so that the Master will act as the Control Console. Once logical branching is identified, the Slave (version) macros will be copied to the WORK catalog. Any previously loaded user macro with the same name in WORK will then be overwritten, therefore the Slave Macro Library is protected.

We can make sure that our version of the macro is executed by making sure that the correct version is in the WORK.SASMACR catalog. Here ❽ a user has purposely created and compiled their version of the macro %CLEANUP, which will thereafter supersede our verified version in the Slave macro catalog (stored compiled macro library). They should not be able to do this if your version of %CLEANUP already exists in the WORK.SASMACR catalog, and the NOMREPLACE option is set, however remember we cannot protect this system option.

```
%macro cleanup; ❽
  %put User macro CLEANUP;
%mend cleanup;
```

For this example let's assume that the %BIGSTEP macro calls the %CLEANUP macro. Although the %BIGSTEP macro is the correct version, its call to %CLEANUP will result in the use of an unverified macro. We get around this by using %BIGSTEP as the master that makes sure that all of its macro calls are correct. The first few lines of %BIGSTEP include a call to the macro %VERCOPY ❾, which was mentioned earlier in this paper.

```
%macro bigstep;
   %vercopy(verlist=cleanup ghi)  ❾
   %cleanup ❿
   %* do other things;
   %ghi  ❿
%mend bigstep;
```

The %VERCOPY macro copies one or more slave macros from the stored compiled macro library to the WORK.SASMACR library, thus guaranteeing that the correct version of %CLEANUP and %GHI will be utilized ❿.

```
%macro vercopy(verlist=)/store;
proc catalog c=complib.sasmacr
             force
             et=macro;
    copy out=work.sasmacr ;
    select &verlist;
    quit;
%mend vercopy;
```

It is also possible to delete specific macro definitions from the WORK.SASMACR catalog. The macro %PURGEWORK can be used to delete one or more macros from the WORK.SASMACR catalog. This allows you to delete any user defined macros that will interfere with one of your validated macros. Once deleted from WORK.SASMACR the macro facility will need to search for the macro definition in either the stored compiled macro library or in the autocall macro library, either of which will contain the valid version of the macro. While the technique utilized by %PURGEWORK should work fine, the %VERCOPY approach is probably more robust. The developers of the SAS Macro Language would remind us at this point that ***neither*** of these two approaches have been tested by SAS. They would be concerned that there may be traces of a macro that lingers in memory even after it has been deleted from WORK.SASMACR. In SAS 9.3 this problem is solved more satisfactorily by the introduction of the %SYSMACDELETE statement, which is designed to delete a macro from the WORK.SASMACR catalog.

```
%macro purgework(macname=);
proc catalog cat=work.sasmacr
             entrytype=macro;
   delete &macname;
   run;
%mend purgework;
```

Multi -macro library identification and linkage: The Master macro can be designed with an 'invisible activation key' to execute the corresponding Slave call. If there is a user macro call with the same name as the Master call's and placed in the WORK; without the KEY, no further action will happen.

Work environment detection and settings: For example the code block displayed above can be used for OS (Unix, Windows, etc.), SAS License version detection and some other desirable features like user access and version control, etc.).

Regulating needed SAS Options: The ability to modify the values of SAS options is always available to all users; some options are needed for user programs and some may be critical for the macro application for standardized outcome and even for the better performance. **Options that are critical to the success of a macro or application should be set within the macro and not solely instantiated during the initialization of the SAS session.**

What We Do NOT Have

The majority of the techniques in the previous sections were designed to provide us control that we only minimally can obtain through options. Even using the techniques described above there are loop holes that prevent full control. The following are some things that would be very helpful – things on our wish list.

Library Search Order Control
An option is needed to control macro library search order; much like FMTSEARCH is for formats searches. This option would allow us to put a compiled macro library first in the search order, thus avoiding the need to copy the library to the WORK.SASMACR. As an alternative the INSERT option could be altered to apply to macro library searches.

Password Protection for Catalogs
Similar to the password data set options, password protection at the catalog level, or better yet at the entry level, would allow us to protect and control individual macro versions in the stored compiled macro library.

Macro Location
An expansion of the MAUTOLOCDISPLAY option that shows the location of each macro executed. Ideally the returned value could be captured in a macro variable or a data set, rather than just being written to the LOG. A similar option to identify the library location for compiled macros would be helpful.

PROTECTING MACRO VARIABLES AND AVOIDING MACRO VARIABLE COLLISIONS

Macro variable collisions can play havoc with an application that passes information through macro variables, or uses macro variables to control its processes. Usually these collisions are inadvertent, and usually they are a result of a poor understanding of symbol table assignment rules when a macro variable is defined. But we should not be restricted by these arcane rules! What can we do to protect the macro variables that are key to our application?

The rule for your users might be; "Always write your macro variables to the local symbol table. Never use the global symbol table or any symbol table higher than the local table to pass information out of your macro." This will be a hard rule for your users to follow. Not only does it require diligence, but a higher level of macro programming skills than is commonly encountered.

What We Have

If every macro author always explicitly assigned every macro variable to the intended symbol table, the problem would go away. Of course we are not always able to make that assignment explicitly. Very often the macro variable is intended to be placed in the local symbol table and the %LOCAL statement can be used. However it is not always that easy. In each of the following macros (%DATALIST and %SQLLIST) the list of student names in the data set SASHELP.CLASS are to be stored in a list of macro variables (&NAME1, &NAME2, etc.). The variable names are intended to be local, but, because we do not know the names of the macro variables until macro execution, a %LOCAL statement cannot be used.

```
%macro datalist(dsn=);
%local i;
data _null_;
   set &dsn;
   call symput('name'||left(put(_n_,3.)),name);
   call symput('nname', left(put(_n_,3.)));
   run;

%do i = 1 %to &nname;
   %put &&name&i;
%end;
%mend datalist;

%datalist(dsn=sashelp.class)
```

```
%macro sqllist(dsn=);
%local i;
proc sql noprint;
   select name
      into :name1-:name9999
         from &dsn;
   quit;

%do i = 1 %to &sqlobs;
   %put &&name&i;
%end;
%mend sqllist;

%sqllist(dsn=sashelp.class)
```

Using CALL SYMPUTX to Force Macro Variables to the Local Symbol Table

Unlike SYMPUT, the newer SYMPUTX routine has the ability to force a macro onto the local symbol table. By setting the optional third argument to 'L', the assignment of the macro variable is made to the local table. The SYMPUTX routine has the added advantage over the SYMPUT routing of not expecting the second argument to be character. The `left(put(_n_,3.))` is no longer needed.

```
call symputx('nname', _n_,'l');
```

The %LOCAL statement and the use of SYMPUTX with the third argument set to 'L' will force macro variables into the local symbol table. Ultimately this makes little difference to the macro itself, but it can make a huge difference to the calling macros. Placing macro variables in the local symbol table protects macro variables in the higher tables. There is no similar tool for use in a SQL step.

Forcing Macro Variables to the Local Symbol Table in the SQL Step

```
proc sql noprint;
   select count(name)
      into :num
         from &dsn; quit;

%do i= 1 %to &num; %local name&i; %end;

proc sql noprint;
   select name
      into :name1-:name&num
         from &dsn; quit;
```

Unlike the SYMPUTX routine in the DATA step, the SQL step does not have a way to force a macro variable onto the local table. In this SQL step a series of macro variables are created: &NAME1, &NAME2, … . There is no way however, to know for sure what symbol table they will be written too! If this step was to be used inside of a macro, these macro variables might not be written to the local table, unless those same variables were *already* on the table. We could use a %LOCAL statement if we knew how many variables were to be created.

Here in this program fragment we count the names and then use a %DO loop, which creates a series of %LOCAL statements. This approach works, because the variable names are numbered.

Naming Conventions
It may be possible to establish a macro variable naming convention that will help protect your macros. You could start all your vulnerable macro variable names with a standard set of characters, and then train your users are not to use those characters. The rule might be; “Never start your macro variable names with ‘SQL’, ‘SYS’, or ‘APPL’. If you do, bad things may happen.” Of course once again you are relying on the good will, training, and professionalism of other folks.

Protecting Global Macro Variables
Although it is vulnerable to macro variable collisions, the global symbol table can be very useful for maintaining control values utilized by the application. Since these values can be easily altered by the user, it is important to be able to monitor and reset their values. This can be facilitated by one or more of the following possible techniques:

- Centralize all of the needed global macro variable definitions in one macro with appropriate naming conventions.
- Retrieve or reestablish values stored in a compiled macro as noted in the previous item
- Utilize access controlled dataset (Read/Write/Alter) to preserve dynamic changed values needed for later retrieval after user macro execution
- Pass all control values as macro parameters (thus avoiding the global symbol table).
- Store and restore defined (g)options before and after the execution of protected macros

What We Do NOT Have
Unfortunately there is little that we can do to protect the macro variables in a given symbol table from changes that might be made by macros that we call. The author of the macro that we call can protect us, but we cannot protect our own table if the author of that macro we call does not choose to protect us. This of course makes our macro variables vulnerable to collisions.

The problem of macro variable collisions could be alleviated with a few tools and options:

- An option to protect a macro variable or even an entire symbol table. This might take the form of making the variable or symbol table read only or password protected.
- An equivalent of the SYMPUTX routine’s third argument for the SQL step
- The ability to place a macro variable in a specific symbol table when it is created. We need something more flexible than just the %GLOBAL and %LOCAL statements.

PREVENTING MACRO REVERSE ENGINEERING
Reverse engineering of macro code generally takes place when someone copies, changes, and then executes the macro that we have otherwise verified and validated. To prevent this from happening we need to be able to protect our source code and to prevent its extraction. Generally speaking, code extraction is often based on the LOG generated with the MPRINT option turned on. As was mentioned earlier, the source code compilation /SECURE option can turn off the MPRINT, MLOGIC and SYMBOLGEN, however the notes generated from the internal logics of the macro will still appear in the LOG.

Another common approach used by those wishing to reverse engineer our macros is to use the MFILE and MPRINT system options. These options can be used to save the macro’s resultant code. Of course the macro logic is not recorded, nor are any other macro language references. Still some information can be revealed.

The macro %DUMPCODE will start routing resultant code to a file (TARGET.SAS), which will be a SAS program. If our user wishes to see the code generated by the macro %GHI, the macro call would be preceded by a call to the %DUMPCODE macro.

```
%macro dumpcode;
filename mprint "c:\temp\target.sas";
options mprint mfile;
%mend dumpcode;
```

```
%dumpcode
%ghi
options nomfile;
```

The MFILE system option requires that the MPRINT option has also been turned on. Fortunately for us, as long as we remember to use the /SECURE option on the %MACRO statement, MPRINT is temporarily turned off. In this example TARGET.SAS will not be created, as long as, the %MACRO statement has included the /SECURE option.

Another information source that we may need to hide is the value of macro variables. The SYMBOLGEN option reveals the values of resolved macro variables in the LOG. If you need to prevent the surfacing of these values in the LOG, the SYMGET function can be useful. Because it is a DATA step function, although it can be used in a SQL step as well, SYMBOLGEN does not apply to its returned value.

What We Have

If we understand the techniques that can be used to obtain our macro's source code, we can take several precautions to mitigate the efforts of those interested in the reverse engineering of our macros:

- Turn off the MPRINT function within the target macro by using either the NOMPRINT system option or the /SECURE option on the %MACRO statement
- Hide any crucial macro variables, such as control codes, passwords, and user ids (Sherman and Carpenter, 2007).
- To prevent critical codes from displaying in the LOG, use PROC PRINTTO to reroute the LOG to a temporary file.
- Use the SYMGET function in the DATA and SQL steps to prevent the surfacing of macro variable values in the LOG.

CONCLUSION

While we do not have all the tools that we would like to have for the protection of our macros and macro variables, there is still a lot that we can do. We do have to be aware of the issues and we need to be careful in the implementation of our applications.

As we implement these techniques, we need to always keep our audience in mind. There is little value in a fancy sophisticated scheme if it is too cumbersome to use. There is also very little value in spending the time and effort to build some of these tools, if our users are sophisticated enough and sufficiently trained to avoid the pitfalls and traps from which we are trying to protect them.

REFERENCES

Carpenter, Arthur L., 2001, "Building and Using Macro Libraries", presented at the: 9th Western Users of SAS Software Conference (September, 2001), Midwest SAS Users Group Conference (November, 2001), and Pacific Northwest SAS Users Group (November, 2005). The paper was published in the proceedings for these conferences. http://caloxy.com/papers/45-p17-27.pdf

Carpenter, Arthur L., 2004, *Carpenter's Complete Guide to the SAS® Macro Language, 2nd Edition*, Cary, NC: SAS Institute Inc. http://www.sas.com/apps/pubscat/bookdetails.jsp?catid=1&pc=59224

Carpenter, Arthur L., 2005, "Make 'em %LOCAL: Avoiding Macro Variable Collisions", proceedings of the Pharmaceutical SAS User Group Conference (PharmaSUG), 2005, Cary, NC: SAS Institute Inc., paper TT04. Also published in the proceedings of the 13th Annual Western Users of SAS Software, Inc. Users Group Conference (WUSS), Cary, NC: SAS Institute Inc., paper SOL_Make_em_local_avoiding. http://caloxy.com/papers/62_TT04.pdf

Choy, Murphy, 2011,"Protecting SAS Programs with SAS Data", article in Views Issue 54. http://www.sascommunity.org/mwiki/images/7/75/VIEWS_News_Issue54.pdf

Sherman, Paul D. and Arthur L. Carpenter, 2007, "Secret Sequel: Keeping Your Password Away From the LOG", proceedings of the Pharmaceutical SAS User Group Conference (PharmaSUG), 2007, Cary, NC: SAS Institute Inc., paper TT07. Also presented in 2007 at the 15th Annual Western Users of SAS Software, Inc. Users Group Conference (WUSS), San Francisco, CA. http://caloxy.com/papers/74Secret.pdf

ABOUT THE AUTHORS

Eric Sun is a SAS Reporting Specialist who has been leading multiple worldwide SAS and eSubmission projects at sanofi-aventis since 2003. Since 1990, Eric has been in pharmaceutical and medical device industries; providing clinical SAS® programming and system development for companies including J&J, Novatis, Pfizer, CR Bard Inc. Throughout his various roles and career growth, Eric has presented multiple papers at SUGI, PharmaSUG and regional user groups. Lately he has specialized in SAS reporting in multiple languages like English, Japanese, Chinese, and in trial public disclosure to regional and local authorities.

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 1977 and has served in various leadership positions in local, regional, national, and international user groups. He is a SAS Certified Advanced Professional programmer, and through California Occidental Consultants he teaches SAS courses and provides contract SAS programming support nationwide.

AUTHOR CONTACT INFORMATION

Your comments and questions are valued and encouraged. Please contact the authors at:

Author Name:	Eric Sun	Art Carpenter
Company:	sanofi-aventis U.S. Inc.	CALOXY
Address	200 Crossing Boulevard,	10606 Ketch Circle
City state ZIP	Bridgewater, NJ 08807	Anchorage, AK 99515
Work Phone:	(908) 304-6681	(907) 865-9167
E-mail:	eric.sun@sanofi-aventis.com	art@caloxy.com

ACKNOWLEDGEMENTS

We would like to offer a special thanks to Russ Tyndall a Technical Support Analyst with the DATA Step and Macro Language Team for providing a technical review of the issues brought up in this paper.

We also want to thank Mr. Makdad Sebai, s-a Paris site based, for his contributions to some practical cases presented in this paper.

TRADEMARK NOTICE