SESUG 2025 - Paper IACS-108

SAS® Code Hidden in Plain Sight*

Bartosz Jabłoński - yabwon / Warsaw University of Technology

ABSTRACT

Sometimes there is a need to share a SAS® macro in a way that the execution of the macro is possible, but at the same time the source code is not "readable" (e.g., the code is a proprietary solution). In this article, a programming-based solution utilizing encrypted user-written functions (FCMP) and SAS data sets, which gives requested effect, will be explained. Furthermore, the solution (in contrary to operating-system-dependent SAS catalogs) allows for code exchange between different operating systems! As a cherry on top, the GSM (Generate Secure Macros) package, which allows users to create secured macros stored in SAS Proc FCMP functions and share them between different operating systems without showing their code, will be presented.

Table of contents

INTRODUCTION AND TWO QUOTES

Many SAS users and developers gather around various community groups to discuss, share ideas, and solve problems. To enumerate some of those, there is (the most popular) `communities.sas.com`. Next in line (my favorite) is the `SAS-L` discussion mailing list, and of course Stack Overflow, and Reddit (r/sas) or Slack forum. There exists also one more, a little bit less popular, gathered around the `www.sasensei.com` quiz, discussion group. Late June 2021, Anamaria Dinu asked there the following question:

> *"Is there any way you can run a code, without seeing the program ? I am asking if needed to share the code with someone, we can add options to not display the source code to the log, but can there be any restrictions on the program, like password protected?"*

Basically it was about: can, independently of the operating system, a SAS macro be executed on client site but without showing the source code, in other words can we have an encrypted SAS code that works, but is not human readable (a code hidden in plain sight)?

The discussion began. During the discussion, I posted several proposals which were elegantly "invalidated" by Lex Jansen, who also took part in discussion and to whom I am very grateful for his comments which eventually led me to the idea described in this article.

As I wrote, in that discussion, after first few iterations it seemed to be impossible to get a solution satisfying those requirements but, as a quote (attributed to Linus Torvalds) says:

> *"If you know the system well enough, you can do things that aren't supposed to be possible."*

Eventually the problem was resolved and in the next few chapters, I am trying to explain how to get it done.

*The text was originally published in WUSS 2023 conference proceedings (see [**Jablonski(2) 2023**]). This is a "republish".

MACROS AND (WHY NOT) CATALOGS

We start with a macro and let us assume we have the following one:

code: a macro

```
%macro notSecretMacro(question);
  data _null_;
    x = rank("*");
    put "The answer for &question. is: " x;
  run;
%mend;
```

We can easily assume that we start with a macro, because even if we had a code which is not a macro, e.g., pure 4GL, IML, or SQL, we can get a macro by adding three lines of a macro wrapper:

code: a wrapper

```
%macro wrapper();
  <... the code goes here ...>
%mend;
%wrapper()
```

Let's continue with compiling the macro. The process does not leave too many tracks in the log, but when we look into the WORK library we see that the `sasmacr` catalog (that is for Base SAS session, if we are working with SAS EG or Studio it will be rather `sasmac1` or something similar) contains the compiled macro. The final test that compilation succeeded is to run the following:

code: run macro

```
%notSecretMacro('question about life, universe, and all the rest')
```

In the LOG we see (for color convention check Appendix A - code coloring guide):

the log

```
1    %notSecretMacro('question about life, universe, and all the rest')

The answer for 'question about life, universe, and all the rest' is: 42
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds
```

The macro works fine, and answers our question, but it has one flaw - the macro source is almost plain text visible, which is not the intention here[1]. First of all, enabling:

code: macro printing options

```
options mprint mlogic symbolgen;
```

reveals the following in the log:

the log

```
1    %notSecretMacro('question about life, universe, and all the rest')
MLOGIC(NOTSECRETMACRO):  Beginning execution.
MLOGIC(NOTSECRETMACRO):  Parameter QUESTION has value
                         'question about life, universe, and all the rest'
MPRINT(NOTSECRETMACRO):   data _null_;
MPRINT(NOTSECRETMACRO):   x = rank("*");
SYMBOLGEN:  Macro variable QUESTION resolves to
            'question about life, universe, and all the rest'
MPRINT(NOTSECRETMACRO):   put "The answer for 'question about life, universe,
                          and all the rest' is: " x;
```

[1] To be clear, the author is a proponent of the open source code solutions, but sometimes the "business case" requires some protection and that is why author plays "devil's advocate" and the article discussion takes place.

```
MPRINT(NOTSECRETMACRO):   run;

The answer for 'question about life, universe, and all the rest' is: 42
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds

MLOGIC(NOTSECRETMACRO):  Ending execution.
```

and, basically, we see every line of 4GL code which was inside the macro.

Similarly, if we run the FILENAME statement pointing to the catalog with the macro and we extract the content with a "plain text reading DATA STEP":

code: a plain text reading DATA STEP

```
filename ns catalog 'work.sasmacr.notSecretMacro.macro';
data _null_;
  infile ns;
  input;
  put _INFILE_;
run;
```

We can see that the LOG, even without running the macro itself, presents quite a number of details:

the log

```
NOTE: The infile NS is:
      Catalog Name=WORK.SASMACR.NOTSECRETMACRO.MACRO,
      [...]
      File Size (bytes)=5120

┌♠ NOTSECRETMACRO ∓Z 9.4 ┌+∞•▲¶-
&┌┌
¬┌⧈ QUESTION
╫¬│
└╚ data _null_;  x = rank("*");  put "The answer for &question. is: " x;  run;
&-▲
┘
╫•+∞¶--
NOTE: 8 records were read from the infile NS.
      The minimum record length was 4.
      The maximum record length was 116.
NOTE: DATA statement used (Total process time):
      real time           0.05 seconds
      cpu time            0.00 seconds
```

The printout is a bit obfuscated, but still not enough so we could not tell the essence of the code.

Various articles (see [**Sun & Carpenter 2011**]) or notes (see [**Usage Note 23210**]) provide help and high-quality discussion on how to increase macro "secrecy". Also someone who at least once read the %MACRO statement documentation (see [**Macro Statement**]) finds the answer to the question about how to improve macro "secrecy" almost obvious.

The SECURE option added to a macro definition provides encryption of that macro source code. The documentation states:

> *"[The SECURE option] causes the contents of a macro to be encrypted when stored in a stored compiled macro library.*

> *The SECURE option enables you to write secure macros that prevent access to the macro source code, which helps to protect intellectual property contained in the code. For macros that are compiled with the SECURE option, the %COPY macro statement cannot recover the macro source code. When executing macros that are compiled with the SECURE option, system options MPRINT and MLOGIC are disabled so that the logical decisions made and the SAS code generated by the secure macro are no longer written to the SAS log."*

Hence, by adding the option to the macro definition, we are able to add a security layer which takes us one step closer to the solution of the problem stated in the Introduction. Why it is just one step closer and not the final solution we discuss in a moment. Now let us take a look at the result of adding the SECURE option. The difference in the macro definition is almost not noticeable, which is why we will highlight modified code:

code: a secure macro

```
%macro secretMacro(question) / SECURE ;
  data _null_;
    x = rank("*");
    put "The answer for &question. is: " x;
  run;
%mend;
```

The encrypted macro, even if executed with "printing options" enabled, provides no information about its source:

code: run macro

```
options mprint mlogic symbolgen;
%secretMacro('question about life, universe, and all the rest')
```

In the LOG, we can see only what the developer wants us to see:

the log

```
1    options mprint mlogic symbolgen;
2    %secretMacro('question about life, universe, and all the rest')

The answer for 'question about life, universe, and all the rest' is: 42
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds
```

(*Note:* A side note, but worth mentioning, is if we add the NONOTES and NOSOURCE options inside the macro definition, even less info will be printed in the log e.g., all "data/proc statement used" notes will disappear.)

When we try to repeat the trick with the FILENAME statement to print out catalog content, we will see a lot of unreadable symbols:

the log

```
1    filename s catalog 'work.sasmacr.secretMacro.macro';
2    data _null_;
3      infile s;
4      input;
5      put _INFILE_;
6    run;

NOTE: The infile S is:
      Catalog Name=WORK.SASMACR.SECRETMACRO.MACRO,
      [...]
      File Size (bytes)=5120

┌♠ SECRETMACRO ∓Z 9.4 ┌+∞•▲¶-
```

```
⊗⌐●∓╫♠╫∟+●⏁ ⌟⊞↗♦
⋇╫
■⌐∞⊥¶●●¬::♦⏁ ♦◈│ │ │⏁ ∞⊥∞↗⊞⊡⊗⏁ ⊥⊞ᛕ⏁ ♦◈⊥¬♦ⵣ⏁ ♠ⵣ⊥⊥↗♦+::■ⵣ●●+ⵣᛕ●╫╫⏁ ¶ᛕ⨯::↗⨯⊞ⵣ
⌟♦♠¬⏁ ⊥⏁ ⊡↗◈╫⏁ ⏁ ⋇∟╫■
⋇⊞◈⌟●⌟↗♠♦⊞↗♠╫¶¶⊗∟⨯◈ᛕ▲⌐⋇∟
⊡▲⊡∞ⵣ♦ⵣ⊡⌟⏁ +¶ᛕ∓ᛕⵣ♦∟⊞│⊥⨯⏁ ⊗∓⨯¬⋇◈■¬ᛕ◈●♠+ⵣ⊞ᛕᛕ⊡∞⨯¬♦⊞■⌟+▲⊞⏁
■+⏁ ↗⊥⌐⋇│♠●↗⌟╫⊥ᛕ¶■∟⌟●¬♦+ᛕⵣ
∓
⌐+⊗⏁ +⊞ᛕ│ │⌐::∞∓+¶⨯¶−
NOTE: 8 records were read from the infile NS.
      The minimum record length was 12.
      The maximum record length was 136.
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.00 seconds
```

Even the `%PUT` statement fails (with a spectacular error):

the log

```
1    %put %secretMacro('question about life, universe, and all the rest');
ERROR: Attempt to execute the /SECURE macro SECRETMACRO within a %PUT statement.
ERROR: The macro SECRETMACRO will stop executing.
```

Now the question is: Are we done? Only partially... And why is that? Well, there are two reasons. The first, SAS catalogs with tons of their practical use cases have a small drawback. SAS catalogs are Operating System dependent. When we generate secure macro in a SAS catalog under Windows OS, we are not able to use it under Linux OS; even 32-bit and 64-bit sessions on the same OS cause problems. A copy of the `sasmacr` catalog moved from Windows results in the following printout in the Enterprise Guide SAS Session running on Linux:

the log

```
1          libname LinuxOS "/home/user/SAS/TEST";
NOTE: Libref LINUXOS was successfully assigned as follows:
      Engine:        V9
      Physical Name: /home/user/SAS/TEST
2          options mstored sasmstore=LinuxOS;
ERROR: File LINUXOS.SASMACR.CATALOG was created for a different OS.
NOTE: The SAS System was unable to open the macro library referenced
      by the SASMSTORE = libref LINUXOS.
WARNING: Apparent invocation of macro SECRETMACRO not resolved.
3
4          %secretMacro('question about life, universe, and all the rest')

           _
           180
ERROR 180-322: Statement is not valid or it is used out of proper order.
```

The `SECURE` option has some serious limitations, which are consequences of how the macro language works. The detailed discussion about it requires a dedicated section and can be found in LIMITATIONS.

Since we are not one hundred percent satisfied with the obtained result, let us pause the "macro path" for a minute and refresh our memory. Where else in SAS programming did we encountered the encrypted code? The answer is presented in the next section.

FUNCTIONS AND (WHY YES) DATA SETS

Now let's have a look at the `FCMP` procedure and user-defined functions.

To create a user-defined function in SAS we use the FCMP procedure. The source code is wrapped in between FUNCTION and ENDSUB keywords. Function accepts zero or more numeric or character arguments, and returns a value with the RETURN statement. An example of such a function accepting character input and returning a numerical result can be seen here:

code: a function

```
proc FCMP outlib=work.notSecret.p;
  function notSecret(question $);
    x = rank("*");
    return(x);
  endsub;
run;
```

The FCMP procedure, "equipped" with the OUTLIB= option, stores the created function in the work.notSecret data set. The third argument (p) in the name may be considered as a "special grouping parameter", in the FCMP procedure nomenclature is called a package (but in totally different meaning than word "package" used in up-coming sections!)

A function created this way can be easily used in a SAS DATA STEP code, or even the SQL procedure query. The following example:

code: call a function

```
options cmplib = work.secret;

data _null_;
  rcDS = notSecret('question about life, universe, and all the rest');
  put rcDS=;
run;

proc sql;
  select distinct
    notSecret('question about life, universe, and all the rest') as rcSQL
  from
    sashelp.class(obs=1)
  ;
quit;
```

prints out (for DATA STEP) in the LOG

the log

```
rcDS=42
```

and in the output window (for SQL):

rcSQL
42

The data set that is created by the FCMP procedure has a special structure, a special type assigned, and a dedicated index created, but even with those all "specialties", it is still a data set, and the source code of the function can be easily previewed without any tricks, just by using the PRINT procedure:

code: proc print

```
proc print data=work.notSecret noobs;
  where NValue;
  var Type Subtype Name Value;
run;
```

The output window reveals all the gory details of the definition:

Type	Subtype	Name	Value
Statement Source	Executable	FUNCTION	function notSecret(question $);
Statement Source	Executable	ASSIGN	x = rank("*");
Statement Source	Executable	RETURN	return(x);
Statement Source	Executable	ENDSUB	endsub;

Once again, to someone who at least once read the FCMP procedure documentation (see [**FCMP Procedure**]), the answer to the question about how to improve function "secrecy" is almost obvious. The ENCRYPT option (aka HIDE) added to the FCMP procedure call provides encryption of the source code of the function. The documentation states (though a bit briefly):

> *"[The ENCRYPT option] specifies to encode the source code in a data set."*

Hence, by adding the option to the function definition in the FCMP procedure header, we are able to add a security layer which (one more time) moves us one step closer to the solution of the problem stated in the Introduction. Why is it, again, just only one step closer and not the final solution we discuss in a moment? Now let us take a look at the result of adding the ENCRYPT option. The difference in the procedure call is almost not noticeable. That is why the modification in the code is highlighted:

code: a secure function

```
proc FCMP outlib=work.secret.p / ENCRYPT ;
  function secret(question $);
    x = rank("*");
    return(x);
  endsub;
run;
```

When we try to repeat the PRINT procedure experiment, the encrypted function only produces a bunch of gibberish.

Type	Subtype	Name	Value
Statement Source	Executable	FUNCTION	∞⊥¶●●¬╌⁝·♦⏲♦◈⏲∞⊥∞↗
Statement Source	Executable	ASSIGN	╙▣⊗⏲⊥╙↖⏲♦◈⊥¬♦╙↗♠♯¶¶⊗∟⤬-◈↖▲⌐≋∟
Statement Source	Executable	RETURN	≋♠●↗⌟♯⊥↖¶■
Statement Source	Executable	ENDSUB	⌐·⁝·∞∓+¶⤬¶-⏲⌟╙

As we observed earlier, the data set created to store functions has a special structure, a special type assigned, has a dedicated index created, but even with all those "specialties", it is still a data set. And, since data should not be Operating System dependent (after all, they are data!), SAS data sets have this advantage over catalogs: they can be transported between different OSes easily. However, functions are not macros and they do not offer flexibility to which we are accustomed when using the macro code...

In the next section, we try to combine all observations we have gathered so far.

A CHALLENGE

Consider the following constraints:

- we can create encrypted macros containing 4GL code, but they are stored in "not portable" containers - SAS catalogs, and
- we can create (and encrypt) functions stored in portable data sets, but functions do not offer macro language (almost infinite) flexibility.

The situation we would like to have is the possibility to encapsulate the first inside the other, something like:

code: a macro-function hybrid

```
proc FCMP outlib = work.secretFunction.p ENCRYPT ;
  function secretFunction();
    %macro littleSecretMacro(question) / SECURE ;
      data _null_;
        x = rank("*");
        put "Dear user, the answer for &question. is: " x;
      run;
    %mend;
  endsub;
run;
```

and at the same time to have the ability to use capabilities provided by each tool.

The snippet above does not do what we would like to have. Readers proficient with macro programming at once will see an interesting (or rather fun) fact that the snippet will compile without errors but the function created will not contain the macro definition inside its body. A warning in the LOG says:

the log

```
WARNING: FUNCTION 'secretFunction' does not return a value.
```

and catches our attention and clearly highlights the fact that the body of the `secretFunction()` contains no statements at all. The function is completely empty and at the same time the `sasmacr` catalog contains the (encrypted but not transportable) `%littleSecretMacro()`. We are so close but still it's a bummer...

THE RESOLUTION

Sometimes when one works on a problem and the issue is described, or question is asked, in a certain form it seems to be nearly impossible to find the answer. But when the problem description is reformulated using different "expressions", magic happens and the answer pops up like a rabbit from a hat. For example, such situations often happen in mathematics. As funny as it sounds, the same happened when I was working on the problem described in this article. Since I am a Pole, I am thinking in Polish, and the first time I "hit the wall" I was asking myself (in Polish):

"Jak mogę, rozwiązać ten problem?"

Since I could not solve it one way I tried the other. I asked myself the question in English:

"How can I `resolve()` this problem?"

The moment I heard the question "reformulated" in my head, the solution jumped at me like that white rabbit on a knight in Monty Python's movie. And it was not the first time the `RESOLVE()` function "attacked" me this way (see [**Carpenter 2018**]).

The `RESOLVE()` function (see [**Resolve Function**]), though not listed in the "Most Commonly Used Functions" chapter of SAS documentation, is well known and has been considered useful for many years (see [**Whitlock 1998**]). The function takes as an argument a text string or a character expression and "steamrolls" it with the macro processor. As the name of the function says, elements of macro language are resolved. Unlike the case of the `SYMGET()` and `SYMGETN()` functions, not only a single macro variable, but also multiple macro variables in one string, macro calls, and even(!) macro definitions can be resolved.

In the first example, the `RESOLVE()` function calls macro variables `A0` and `B0`, and then the `%C1` macro:

code: learning about the resolve() function

```
%let A0=1 2 3;
%let B0=4 5 6;

%macro C1();
```

```
  data C1;
    do i = 1 to 3;
      put i=;
    end;
  run;
%mend C1;

data _null_;
  rc1 = resolve('&A0. &B0.');
  put rc1=;

  rc2 = resolve('%C1()');
  put rc2=;
run;
```

The results are demonstrated in the LOG:

the log

```
rc1=1 2 3 4 5 6
rc2=data C1;     do i = 1 to 3;       put i=;     end;   run;
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds
```

The `rc1` variable contains values of macro variables `A0` and `B0` separated by a space. The `rc2` variable contains text of the 4GL code generated by the `%C1()` macro (but without running it).

In the second example, the `RESOLVE()` function compiles the `%D2()` macro, whose code is passed as a string text argument to the function:

code: learning about the resolve() function, cont.

```
data _null_;
  rc3 = resolve('
    %macro D2();
      data D2;
        do i = 1 to 3;
          put i=;
        end;
      run;
    %mend D2;
  ');
  put rc3=;
run;

options mprint;
%D2()
```

and the LOG shows the following:

the log

```
[...]
rc3=
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.00 seconds

13
14   options mprint;
15   %D2()
MPRINT(D2):   data D2;
```

```
MPRINT(D2):   do i = 1 to 3;
MPRINT(D2):   put i=;
MPRINT(D2):   end;
MPRINT(D2):   run;

i=1
i=2
i=3
NOTE: The data set WORK.D2 has 1 observations and 1 variables.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds
```

The `rc3` variable value is empty because the text argument (the macro definition), passed in single quotes to the function, was all compiled into the macro.

Now, with the knowledge we have, we can put all the puzzle pieces together. The key elements of the composition are, as usual, highlighted:

code: the macro-function-resolution

```
proc FCMP outlib = work.gsm.secure ENCRYPT ;
  function generateMacro() $;
    rc = RESOLVE('
      %macro ultimateSecretMacro(question) / SECURE ;
        data _null_;
          x = rank("*");
          put "Dear user, the answer for &question. is: " x;
          put "So long, and thanks for all the fish!";
        run;
      %mend;
    ');
    return (rc);
  endsub;
run;
```

In the final part of this section we explain: How does the proposed solution work? First, when we run the code from the snippet, the LOG shows the following:

the log

```
NOTE: Function generateMacro saved to work.gsm.secure.
NOTE: PROCEDURE FCMP used (Total process time):
      real time           0.14 seconds
      cpu time            0.09 seconds
```

and the output from the PRINT procedure gives us the following (similar looking) result but with a significantly longer value in the "ASSIGN" part (see the previous output):

Type	Subtype	Name	Value
Statement Source	Executable	FUNCTION	␣⍐♠⊥¶●╍⁝·♦⏁♦◈⏁∞⊥∞⍐
Statement Source	Executable	ASSIGN	∞⍐␣▣●⍐⌟╫⊗⏁⊥␣↖⏁♦◈⊥¬♦␣⍐♠╫¶¶⊗∟⤫◈↖♦◈⏁∞▲┌⅋∟●⍐⌟╫⊥♠╫¶♦◈⏁∞¶⊗∟⤫♠╫¶¶⊗∟⤫♦◈⏁∞¬♦␣⍐♠╫¶¶⊗∟⤫◈↖♦◈⏁∞▲┌⅋∟●⍐⌟╫⊥♠╫¶♦◈⏁∞¶⊗∟⤫♠╫¶◈⊥¬♦␣⍐♠╫¶⊗∟⤫◈⏁▲┌⅋∟●-⍐⌟╫⊥-♠╫¶♦◈⏁∞¶⊗∟⤫♦⏁↖♦◈-
Statement Source	Executable	RETURN	⅋♠●⍐⌟╫⊥↖¶■
Statement Source	Executable	ENDSUB	┌·⁝·∞∓+¶⤫¶-⏁⌟␣

So far so good! The next step would be to see if the macro works. Unfortunately if we just run:

code: run macro

```
%ultimateSecretMacro('question about life, universe, and all the rest')
```

the result is nothing we would like to see because the LOG shows only:

the log

```
1     %ultimateSecretMacro('question about life, universe, and all the rest')
2     -
3     180
WARNING: Apparent invocation of macro ULTIMATESECRETMACRO not resolved.

ERROR 180-322: Statement is not valid or it is used out of proper order.
```

Is it again a dead end? Well, no, it is not. The warning is one hundred percent valid. We did not execute the function and the macro was not compiled (see the `sasmacr` catalog; there is no `%ultimateSecretMacro()` inside). To fix things we just need to run:

code: execute the function

```
options cmplib=work.gsm;

data _null_;
  rc = generateMacro();
run;
```

The first line, the `CMPLIB=` option, instructs SAS where user-defined functions are stored. The DATA STEP executes the `generateMacro()` function, which runs the `RESOLVE()` function internally and the macro is compiled. As usual, in such situation, the LOG is not very talkative. It just prints a note that the DATA STEP ran successfully, but this time inside the `sasmacr` catalog we can see the newly compiled macro. Now, when we call the macro again, even with all "macro printing" options on, we can see only the text printed by the `PUT` statement and no details about macro source in the LOG:

the log

```
1     options mprint mlogic symbolgen;
2     %ultimateSecretMacro('question about life, universe, and all the rest')

Dear user, the answer for 'question about life, universe, and all the rest' is:
42
So long, and thanks for all the fish!
NOTE: DATA statement used (Total process time):
      real time           0.07 seconds
      cpu time            0.04 seconds
```

Though the solution we obtained is very promising **it is not a "silver bullet"** - we have to remember about some limitations it has. And those limitations of the `SECURE` option are especially important!

LIMITATIONS

In a very early version of the text, this part was planned to be a "small side note", but the discussion we are about to see made it to grow to a full-fledged (and critical) chapter.
The limitations are:

- The first limitation is related to the fact that the argument of the `resolve()` function is limited to 32767 bytes, so if the code we want to hide is "longer" then that we need to split it.
- The second is a consequence of how the `SECURE` option works for macros, according to the documentation:

  > *"[...] due to the way that the macro facility processes and generates text, under certain conditions, it is still possible to recover the final SAS code generated by a secure macro. Secure macros*

should be used only to secure whole program segments, such as entire DATA and PROC steps and never for storing code fragments or passwords. "

Unfortunately the "*under certain conditions*" part is not defined enough to do some testing.

Fortunately there is always a helping hand of the SAS Technical Support Team! And this time, the Team helped greatly, too!

To discuss the limitation, let's start with a macro:

code: a simple macro

```
%macro ABCDE() / SECURE;
  data oneToHide;
    x='you should not see me';
  run;

  %local x;
  %let x=this is another text to hide;
%mend ABCDE;
```

which we want to hide, so we turn on the SECURE option. Two standard tests we already know work great:

code: simple tests

```
options mprint mlogic symbolgen;

/* test 1 */
%put %ABCDE();

/* test 2 */
filename x catalog 'work.sasmacr.ABCDE.macro';
data _null_;
  infile x;
  input;
  put _INFILE_;
run;
```

The first returns an error message as expected. The second returns gibberish to the LOG.

Now we execute one more test with a little help from the following macro :

code: unhide macro

```
%macro unhide(text);
  %put ###&text.###;
%mend unhide;
```

When we run the following:

code: run unhide macro

```
/* test 3 */
%unhide(%ABCDE())
```

in the LOG we see:

the log

```
1    /* test 3 */
2    %unhide(%ABCDE())
MLOGIC(UNHIDE):  Beginning execution.
MLOGIC(UNHIDE):  Parameter TEXT has value
                 data oneToHide; x='you should not see me'; run;
MLOGIC(UNHIDE):  %PUT ###&text.###
SYMBOLGEN:  Macro variable TEXT resolves to
```

```
            data oneToHide; x='you should not see me'; run;
###data oneToHide; x='you should not see me'; run;###
MLOGIC(UNHIDE):  Ending execution.
```

All 4GL code is printed out!

As strange/scary as it sounds, this is the expected behavior and cannot be considered a bug. The explanation provided by the SAS Support Team goes like this:

> *"1) The SECURE option applies only to the execution of the macro with the SECURE options specified in its definition.*
> *2) The generated source created by the execution of a macro compiled with the SECURE option is just like any other macro generated source. Only the generation of the source and the model text in the macro definition are protected."*

When we think about it, these are totally valid and natural arguments. Macro is a "text generator" and when one macro generates some text (e.g., a 4GL code) the other one can use the text as an input. Unfortunately, such behavior makes the SECURE option almost useless... at least in the case of 4GL.

When we take a deeper look at that %unhide() macro, we see that if we modify the original %ABCDE() macro a bit we can "break" unhiding. The modification is by adding "technical" commas to the source code, which break the unhide with the "more parameters than defined" error:

code: break unhide macro

```
%macro ABCDE2() / SECURE;
  proc sql noprint;
    select 1 as a, 2 as b, 3 as c  /* <- this breaks unhide */
    from sashelp.class(obs=0);
  quit;
  data oneToHide;
    x='you should not see me';
  run;
  %local x;
  %let x=this is another text to hide;
%mend ABCDE2;
```

and when we run the %unhide() one more time the LOG shows:

the log

```
1    /* test 3 */
2    %unhide(%ABCDE2())
MLOGIC(UNHIDE):  Beginning execution.
MLOGIC(UNHIDE):  Parameter TEXT has value
                 proc sql noprint; select 1 as a
ERROR: More positional parameters found than defined.
MLOGIC(UNHIDE):  Ending execution.
```

Though it looks promising we can, unfortunately, very easily improve the %unhide() "exploit" to circumvent this case by using the PARMBUFF option and the syspbuff macro variable:

code: improved unhide macro

```
%macro unhide() / parmbuff;
  %put ###&syspbuff###;
%mend unhide;
```

With that "fix" the %unhide() macro becomes "commas resistant" and it looks like the use of the SECURE option is now totally useless... at least in the case of 4GL code.

Rather grim news... But wait! In those examples, the "macro code" from the macro definition was not printed at all. Will it be that the "pure macro code" (i.e., not generating any text) secured macros cannot be "exploited"? The following example demonstrates a "pure" macro:

code: pure macro code

```
%macro ABCDE3() / SECURE;
  %local x i;
  %let x=this is another text to hide;

  %do i = 1 %to 5 %by 2;
    %put We are in [&sysmacroname.], &=i.;
  %end;
%mend ABCDE3;
```

that shows no code when treated by the `%unhide()` macro:

the log

```
1    /* test 3 */
2    %unhide(%ABCDE3())
MLOGIC(UNHIDE):  Beginning execution.
We are in [ABCDE3], I=1
We are in [ABCDE3], I=3
We are in [ABCDE3], I=5
MLOGIC(UNHIDE):  %PUT ###&syspbuff###
SYMBOLGEN:  Macro variable SYSPBUFF resolves to ()
###()###
MLOGIC(UNHIDE):  Ending execution.
```

The text, which was "planned" to be printed by the `%PUT` statement, is printed to the LOG but nothing else. This is good news! If we could hide our 4GL code as "pure macro code" that would do the job. Fortunately, there is a very easy way to do it!

We can wrap 4GL in the `%sysfunc(doSubL(...))` "sandwich". The `DOSUBL()` function (see [**Langston 2013**] and [**McMullen 2020**]) allows (as the name suggests) SAS to "DO SUBmit a Line" of code in a "side SAS session" generated when the function is executed. Let's consider the following example:

code: pure macro code with doSubL() function

```
%macro ABCDE4(setName) / SECURE;
  %local rc x i;

  %let rc = %sysfunc(libname(LIB,</some/path>));
  %put &=rc.;
  %let rc=%sysfunc(doSubL(%str(
    options ps=min nonotes nomprint nosymbolgen nomlogic nosource nosource2;
    data LIB.&setName.;
      a = 42;
      x = 'you should not see me in the log';
      put "TEXT:    we are inside [&sysmacroname.] a=" a;
      put "NOTE:    we are inside [&sysmacroname.] a=" a; /* not printed */
      put "WARNING: we are inside [&sysmacroname.] a=" a;
      drop x;
    run;
  )));
  %let x=this is another text to hide;
  %do i = 1 %to 3;
    %put &=i.;
  %end;
%mend ABCDE4;
```

When we wrap that 4GL code in the `%sysfunc(doSubL(%str(...)))` sandwich, it practically makes that 4GL a "macro code". The LOG shows only:

the log

```
1    /* test3 */
2    %unhide(%ABCDE4(newData))
3  MLOGIC(UNHIDE):  Beginning execution.
4  RC=0
5  [...]
6  TEXT:    we are inside [ABCDE4] a=42
7  WARNING: we are inside [ABCDE4] a=42
8  I=1
9  I=2
10 I=3
11 MLOGIC(UNHIDE):  %PUT ###&syspbuff###
12 SYMBOLGEN:  Macro variable SYSPBUFF resolves to ()
13 ###()###
14 MLOGIC(UNHIDE):  Ending execution
```

(*Note:* The `[...]` hides a several lines long "gap" of blank lines in the LOG text. The gap is caused by the DOSUBL() generated side-session.)

It is *important* to set the PS=MIN option to prevent an "infinite log print out" in case the main SAS session has the ps= option set to MAX. Additional options turned off in the 4GL code did not allow SAS to display notes and source code. What I find very useful is the fact that options change inside the DOSUBL() side session do not propagate to the main session.

We managed to prevent the "exploit" but we need to be aware that limitations make the approach much less general[2] than we would like it to be. To summarize - we have to have:

- either a macro which is 100% "pure macro code"
- or, if 4GL is required, it has to be wrapped up in the DOSUBL() "sandwich".

The sandwich has its limitations too, for example:

- invoking the side-session slows down the execution time,
- since the code is in the `%let rc=...;` construct it does not allow the insertion of nested conditional %IF-THEN-ELSE logic or %DO-LOOPs inside (there are workarounds, see Appendix D - workarounds for DoSubL() sandwich limitations),
- libraries or file references created in the side-session are not visible in the main session.

Assuming we managed to write a "pure macro code" (what *is* doable), a new question arises: Could such a code be printed out like the 4GL code was printed? When we think about it, such a situation should not happen because all macro statements are elements of the language and all 4GL code (or a "not macro text") is data processed by the macro language. Similarly, as when we push a text string "A B C" into the COMPRESS() function and then we insert the result into the LENGTH() function - as the result we get 3. Just three and not some "bigger number" being result of counting composed length of the "*compress("A B C")*" function argument.

We can now improve the example we have been working on:

code: the macro-function-resolution

```
1  proc FCMP outlib = work.gsm.secure ENCRYPT ;
2   function generateMacro() $;
3    rc=RESOLVE(' /* single opening quote */
4     %macro finalUltimateSecretMacro(question) / SECURE ;
5      %local rc;
```

[2] Author regrets to admit it, but this may limit the number of potential users.

```
    %let rc = %sysfunc(doSubL(%str(
      options ps=min nonotes nomprint nosymbolgen nomlogic nosource nosource2;
      data _null_;
       x = rank("*");
       put "Dear user, the answer for &question. is: " x;
       put "So long, and thanks for all the fish!";
      run;
     )));
    %mend;
   '); /* single closing quote */
  return (rc);
 endsub;
run;
```

At the very end of the section we discuss one more, but solvable, limitation. Even if we have a directory of "properly secured" macros (as discussed above) we have to add the wrapper:

code: wrapper

```
proc FCMP outlib = work.gsm.secure ENCRYPT ;
  function generateMacroXXX() $;
    rc = RESOLVE('
            <...>
    ');
    return (rc);
  endsub;
run;
```

to each and every one, and also set the names accordingly, e.g., `generateMacro01()`, `generateMacro02()`, etc.

Having this in mind, we can ask a natural question: Is it possible to automate the process? Fortunately, the answer is: Yes! There exists a dedicated SAS Package - the GSM package - which is a predefined set of macros just for the job, i.e., the automation of generating secure macros.

Before we discuss how the GSM package works, the next section briefly introduces the idea of SAS Packages.

SAS PACKAGES

In the world of programmers, software developers, or data analysts, the concept of a package, as a practical and natural medium to share your code with other users, is well known and common. To give evidence of this statement, let us consider a few very popular examples: Python, TEX, and R. As an endorsement of this fact, it is enough to visit the following web pages:

`https://pypi.org/`
`https://www.ctan.org/`
`https://cran.r-project.org/`

to see the number of available packages. There are, literally, thousands of packages available for those languages! More than a dozen of TEX packages were used while writing this article.

With that said, the fundamental question of a new or a seasoned SAS user should be: "Why there is no such thing like packages in SAS?" To be clear, there are "packages" in the SAS ecosystem (see the footnote in the package definition) but they do not offer such functionality as those mentioned above. For example, the SAS/IML offers (limited) functionality similar to the concept of a package in the R language mentioned above (for comparison see [**Wickham 2015**]) but such functionality is not available in SAS in general.

The **SAS Packages Framework**, introduced in [**Jablonski 2020**], tries to fill that gap.

A **SAS package**[3] is an automatically generated, single, stand alone `zip` file containing organized and ordered code structures, created by the developer and extended with additional automatically generated "driving" files (i.e., description, metadata, load, unload, and help files).

The purpose of a package is to be a simple (and easy to access) code sharing medium, which allows: on the one hand, to separate the code's complex dependencies created by the developer from the user experience with the final product and, on the other hand, to reduce developers' and users' unnecessary frustration related to a remote deployment process.

The SAS Packages Framework is a "pack" of macros, which allows SAS programmers to *use* and to *develop* SAS packages.

To create a package, the developer executes a few simple steps which, in general, are:

- prepare the code (package content files) and a description file,
- fit them into a dedicated structured form,
- download the `SPFinit.sas` (the SAS Packages Framework) file,
- and execute the `%generatePackage()` macro.

To use a package, the user executes even fewer steps which, in general, are:

- download the `SPFinit.sas` (the SAS Packages Framework) file,
- execute the `%installPackage(packageName)` and the `%loadPackage(packageName)` macros.

The framework is open-source MIT licensed project available at GitHub:

`https://github.com/yabwon/SAS_PACKAGES`

In the [**Jablonski 2021**] article, a step-by-step tutorial, which allows to develop SAS packages, is presented. The [**Jablonski 2023**] tutorial covers the subject in even more details.

Note. Though using or building a package is a very straightforward process, the creation of a package content (i.e. macros, functions, formats, etc.) requires some experience. That is why the intended readers for articles mentioned above are *at least* intermediate SAS programmers. A good knowledge of Base SAS and practice in macro programming will be an advantage. Such a background can be found, for example, in [**Carpenter 2012**].

G[ENERATE] S[ECURE] M[ACROS] PACKAGE

The `GSM` Package (Generate Secure Macros) is a dedicated SAS package that facilitates automation of the process that generates "shareable" secure macros.

How does it work? The package contains two macros and (during the execution) generates the third:

- The `%GSM()` macro - the main macro which "interacts" with the user. It converts a list of macros provided by the user into a data set of the `FCMP` procedure functions. The macros definitions (their code) "stored" in functions are encrypted, which allows users to share them without showing their code. It is important to remember that macros provided by the user have to be "secure", i.e. the `SECURE` option has to be added to the macro definition, plus the way of coding as described in the LIMITATIONS chapter should be applied.
- The `%GSMpck_makeFCMPcode()` - an internal macro called by the main one. It executes the process of converting a macro into a user-defined `FCMP` function.
- During the execution a test macro, named `%GSMpck_dummyMacroForTests()`, is generated.

[3] The idea presented in this article should not be confused with other occurrences of "package" concept which could be found in the SAS ecosystem, e.g. Proc DS2 packages, SAS/IML packages, SAS ODS packages, SAS Integration Technologies Publishing Framework packages, or even SAS Enterprise Guide `*.egp` projects files.

How to use GSM Package:

- Copy all files with your secured macros code into a directory. The best approach is to have one file for each macro (remember the length limitation).
- Copy the path to the directory.
- Run the following code:

code: call GSM

```
%GSM(<the path to the directory>, cmplib=<name of the data set>)
```

- Share the generated zip file.

The User who gets your zip file do the following: 1) copies the zip to the destination machine, 2) extracts the zip contents, 3) opens extracted SAS code file, 4) modifies the <...path...> line, 5) and runs the code.

The package and detailed documentation (with parameter description) is available in the GSM repository located on GitHub:

https://github.com/SASPAC/gsm/

The documentation is in the gsm.md file.

EXAMPLE

It is a well know fact that one of the best approaches to learn is by examples (see Ron Cody's books). We will now do a case study and discuss the following example:

Let us assume we have two files f1.sas and f2.sas located in the /home/user/path2files directory. The files contain the following macros with the SECURE option added to the definition:

code: f1.sas

```
%macro abc(x) / SECURE;
%local rc;
%let rc = %sysfunc(dosubl(%str(
  data result;
    do x = 1 to &x.;
      y = 17 * x + 42; /* "secret" formula */
      put x= y=;
    end;
  run;
)));
%mend;
```

and

code: f2.sas

```
%macro xyz(x) / SECURE;
  %local y;
  %do x = 1 %to &x.;
    %let y = %sysevalf(101 * &x. + 555);  /* one more "secret" */
    %put &=x;
  %end;
%mend;
```

When macros are located in the directory (assuming the SAS Packages Framework and the GSM package are installed too, if not see Appendix B - install the SPF and packages) we run the SAS session and we:

- set filename reference to the directory where the SAS Packages Framework and the GSM package are located e.g., /home/user/packages:

code: set filename

```
filename packages "/home/user/packages";
```

- enable the framework from `SPFinit.sas` file:

code: enable the framework

```
%include packages(SPFinit.sas);
```

- load the GSM package:

code: load GSM

```
%loadPackage(GSM)
```

- run the `%GSM()` macro with the first parameter pointing to the directory where `f1.sas` and `f2.sas` are located, and the second naming the data set in which to store the functions:

code: run GSM macro

```
%GSM(/home/user/path2files, cmplib=work.myMacros)
```

When the `%GSM()` macro ends its run a table with a list of files used in the process is displayed in the output:

Obs	base	file
1	/home/user/path2files	f1.sas
2	/home/user/path2files	f2.sas

and also three files: `mymacros.sas7bdat`, `mymacros.sas7bndx`, and `mymacros.zip` are created. The `mymacros.sas7bdat` data set contains encrypted functions definitions and the `mymacros.sas7bndx` is the index file associated with the data set. When we take a look at the data set we see definitions of three functions: `_MACRO_1_()`, `_MACRO_2_()` and `generateMacros()`. First two correspond to our macros, and the last one is a "function wrapper" that allows run all to run all functions generating macro with just one call. Because of the encryption, the source code inside the `mymacros.sas7bdat` data set is not human readable, soon we will discuss what is happening inside.

Before that and before discussing the `zip` file, we take a look at the LOG because it will give us some explanation of the process. The LOG displays the following information:

the log

```
1    %GSM(/home/user/path2files, cmplib=path.myMacros)
NOTE: [GSM] The PATH is /home/user/path2files
NOTE: The OUTPATH set to:
      /home/user/path2files

NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds

NOTE: [GSM] Value of secret is: 2CD02462A2C5F02D111307C48671BE34

*/home/user/path2files*
1   + %GSMpck_makeFCMPcode(/home/user/path2files/f1.sas,1, trim=0,
       outlib=PATH.MYMACROS.secure, source2=, fileNameCode=_8FAFD2c,
       secret=2CD02462A2C5F02D111307C48671BE34, lineEnd=0D0A)
/#*******************************************************#/
         Total code size in bytes: 656

rc_fd=0
/#*******************************************************#/
2   + %GSMpck_makeFCMPcode(/home/user/path2files/f2.sas,2, trim=0,
       outlib=PATH.MYMACROS.secure, source2=, fileNameCode=_8FAFD2c,
       secret=2CD02462A2C5F02D111307C48671BE34, lineEnd=0D0A)
```

```
/#*************************************************************#/
         Total code size in bytes: 492

rc_fd=0
/#*************************************************************#/
   /*---------------------------------------------------------------------------*/
   /* Code to be run on site. Remember to change the Path for the proper one. */
   /*---------------------------------------------------------------------------*/

   libname code '<...path...>' inencoding='utf-8';
   proc sort
    data = code.mymacros
    out = code.mymacros(
    type=etsmodel
    compress=char
    pointobs=yes
    index=(_Key_)
    encoding='utf-8')
    force;
     by _Key_ Sequence;
   run;
   options cmplib = code.mymacros;
   data _null_;
     rc = generateMacros();
   run;
   data _null_; run;

1    + filename _8FAFD2i "/home/user/path2files/mymacros.sas7bndx"
               lrecl=1 recfm=n;
2    + filename _8FAFD2o ZIP '/home/user/path2files/mymacros.zip'
               member='mymacros.sas7bndx' lrecl=1 recfm=n;
3    + data _null_;
4    +  rc1 = fexist("_8FAFD2i");
5    +  rc2 = fcopy("_8FAFD2i", "_8FAFD2o");
6    +  rc3 = fexist("_8FAFD2o");
7    +  putlog (rc:) (=);
8    + run;
rc1=1 rc2=0 rc3=1
9    + filename _8FAFD2i clear;
10   + filename _8FAFD2o clear;
11   + filename _8FAFD2i "/home/user/path2files/mymacros.sas7bdat"
               lrecl=1 recfm=n;
12   + filename _8FAFD2o ZIP '/home/user/path2files/mymacros.zip'
               member='mymacros.sas7bdat' lrecl=1 recfm=n;
13   + data _null_;
14   +  rc1 = fexist("_8FAFD2i");
15   +  rc2 = fcopy("_8FAFD2i", "_8FAFD2o");
16   +  rc3 = fexist("_8FAFD2o");
17   +  putlog (rc:) (=);
18   + run;
rc1=1 rc2=0 rc3=1
19   + filename _8FAFD2i clear;
20   + filename _8FAFD2o clear;

NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds
```

Lines 2 and 4 are just to present parameters values; the PATH is the location of the directory where macro files are stored; the OUTPATH points to a directory in which a result zip file is generated, by default OUTPATH=&PATH.

Line 10 contains info about the SECRET, which is an optional parameter, whose default value is null, and in such case the value of secret is generated from the sha256(datetime(), hex32.) function and is printed in the log. When it is not null, then the value should be an alphanumerical constant (non-alphanumerical characters are removed). This parameter is required if we want to execute the RESOLVE() function; internally it is set inside the generateMacros() as _macro_XX_("&SECRET."). Users who do not know the value will not be able to run the _macro_XX_() function.

Lines 13 to 28 contain calls to the internal %GSMpck_makeFCMPcode() macro, which is called one by one for each file inside the /home/user/path2files directory. Information about file size is printed too.

Now lets jump to lines from 51 to 80, those lines show the process of copying mymacros.sas7bdat and mymacros.sas7bndx files into the mymacros.zip file, which is in fact the "final product". When the %GSM() ends its run the file which is to be shared is the mymacros.zip file.

We have already mentioned twice that the mymacros.sas7bdat file is not just an ordinary data set. It has some special properties, and when it is shared with other users, an additional step is required to make it work. This additional step is the code printed out between lines 29 and 49. A copy of that code is also inserted into the mymacros.zip file as the mymacros.sas file.

In short, to use the created content, we copy the zip file to the destination we want (on a different machine); we extract the content, start a new SAS session, and execute the program from mymacros.sas in that session (remembering to replace <...path...> accordingly.)

We can also see in the LOG that the library INENCODING= option is set to utf-8; this information is extracted from the SAS Session setup so if the session is executed with different encoding, the value will change (though as a general good programming practice I highly recommend you to work in utf-8!). The encodingRestricted parameter of the %GSM() macro is responsible (when set to 1) for forcing a user's session to have the same encoding as the developer's session had.

The "uncovered" source of the generateMacros() function looks like this:

code: uncovered generateMacros

```
1  function generateMacros();
2    if symget('sysencoding') ne "utf-8" then
3      do;
4        put "WARNING: Current system encoding is different than utf-8.";
5        put 'NOTE: It may affect characters that are "encoding specific".';
6        put "NOTE: The best solution is to run SAS session in utf-8 encoding.";
7      end;
8
9    rc = _macro_1_("2CD02462A2C5F02D111307C48671BE34") ;
10   if rc ne '' then
11     do;
12       put 'WARNING: Macro 1 rc =' rc;
13       put 'WARNING: Return code was not null!';
14       put 'WARNING: The provided code may not be a macro.';
15     end;
16   rc = _macro_2_("2CD02462A2C5F02D111307C48671BE34") ;
17   if rc ne '' then
18     do;
19       put 'WARNING: Macro 2 rc =' rc;
20       put 'WARNING: Return code was not null!';
```

```
      put 'WARNING: The provided code may not be a macro.';
    end;
  return(0);
endsub;
```

and the "uncovered" source of the `_macro_XX_()` function looks like this:

code: uncovered _macro_XX_

```
function _macro_XX_(secret $) $;
  if secret = '2CD02462A2C5F02D111307C48671BE34' then
    rc = RESOLVE('
          < ... >
         ');
  return (rc);
endsub;
```

Note: It is worth mentioning that:

a) a naive test for presence of the `SECURE` word in the macro definition file is executed and it returns an error message when the word is missing,

b) but, in fact, not all macros stored in the `/home/user/path2files` directory have to be "secured", to avoid the error message mentioned above it is enough to add "`/* secure */`" text,

c) if the `resolve()` returns non-empty text, a warning is issued.

CONCLUSION

In the article, we learned how to resolve the problem of creating SAS programs that, on the one hand, are executable on different operating systems and, on the other, have their source code hidden from (too curious) executor's eyes. Also, the `GSM` package, which is dedicated to automating such processes, was presented and discussed.

The End

REFERENCES

[Whitlock 1998] Ian Whitlock, "The RESOLVE Function - What Is It Good For?", NESUG Proceedings, 1998, https://www.lexjansen.com/nesug/nesug98/code/p088.pdf

[Sun & Carpenter 2011] Eric Sun, Art Carpenter, "Protecting Macros and Macro Variables: It Is All About Control", MWSUG Proceedings, 2011, https://www.mwsug.org/proceedings/2011/appdev/MWSUG-2011-AD13.pdf

[Carpenter 2012] Art Carpenter, "Carpenter's Guide to Innovative SAS Techniques", SAS Press, 2012

[Langston 2013] Rick Langston, "Submitting SAS Code On The Side", SAS Global Forum 2013 Proceedings, 032-2013, https://support.sas.com/resources/papers/proceedings13/032-2013.pdf

[Wickham 2015] Hadley Wickham, "R Packages: Organize, Test, Document, and Share Your Code", O'Reilly Media, 2015, http://r-pkgs.had.co.nz/description.html

[Carpenter 2018] Art Carpenter, "Using Memory Resident Hash Tables to Manage Your Sparse Lookups", WUSS Proceedings, 2018, https://www.lexjansen.com/wuss/2018/41_Final_Paper_PDF.pdf

[Jablonski 2020] Bartosz Jabłoński, "SAS Packages: The Way to Share (a How To)", SAS Global Forum 2020 Proceedings, 4725-2020 https://www.sas.com/content/dam/SAS/support/en/sas-global-forum-proceedings/2020/4725-2020.pdf extended version available at: https://github.com/yabwon/SAS_PACKAGES/blob/main/SPF/Documentation

[McMullen 2020] Quentin McMullen, "A Close Look at How DOSUBL Handles Macro Variable Scope", SAS Global Forum 2020 Proceedings, 4958-2020, https://support.sas.com/resources/papers/proceedings13/032-2013.pdf

[Jablonski 2021] Bartosz Jabłoński, "My First SAS Package - a How To", SAS Global Forum 2021 Proceedings, 1079-2021 https://communities.sas.com/kntur85557/attachments/kntur85557/proceedings-2021/59/1/Paper_1079-2021.pdf also available at: https://github.com/yabwon/SAS_PACKAGES/tree/main/SPF/Documentation/Paper_1079-2021

[Jablonski 2023] Bartosz Jabłoński, "Share your code with SAS Packages a Hands-on-Workshop", WUSS 2023 Proceedings, 208-2023, https://www.lexjansen.com/wuss/2023/WUSS-2023-Paper-208.pdf

[Jablonski(2) 2023] Bartosz Jabłoński, "A SAS Code Hidden in Plain Sight", WUSS 2023 Proceedings, 189-2023, https://www.lexjansen.com/wuss/2023/WUSS-2023-Paper-189.pdf

[Usage Note 23210] *How to hide macro code so that it does not appear in the log when the program is executed*, https://support.sas.com/kb/23/210.html

[Macro Statement] The MACRO statement documentation: https://documentation.sas.com/doc/en/pgmsascdc/9.4_3.5/mcrolref/p1nypovnwon4uyn159rst8pgzqrl.htm (link as of August 2025)

[FCMP Procedure] The FCMP procedure documentation: https://documentation.sas.com/doc/en/pgmsascdc/9.4_3.5/proc/p0urpv7yyzylqsn1g2fycva2bs3n.htm (link as of August 2025)

[Resolve Function] The Resolve() function documentation: https://documentation.sas.com/doc/en/pgmsascdc/9.4_3.5/mcrolref/p1ccfaw12wm8o3n1evurg9ov7dp6.htm (link as of August 2025)

ACKNOWLEDGMENTS

The author would like to acknowledge **Anamaria Dinu** and **Lex Jansen** for their inspiration, which spark the idea behind the GSM package.
The author would like to acknowledge the SAS Technical Support Team, especially Kathryn McLawhorn, for valuable discussion.
The author would like to acknowledge Filip Kulon and Troy Martin Hughes whose contribution made this paper look and feel as it should!

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at one of the following e-mail addresses:

yabwon✉gmail.com or bartosz.jablonski✉pw.edu.pl

or via the following LinkedIn profile: www.linkedin.com/in/yabwon or at the communities.sas.com by mentioning @yabwon.

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.
Other brand and product names are trademarks of their respective companies.

Appendix A - code coloring guide

The best experience for reading this article is in color and the following convention is used:

- The code snippets use the following coloring convention:

code: is surrounded by a black frame

```
In general we use black ink for the code but:
- code of interest is in orange ink so that it can be highlighted,
- to distinguish code snippets for easier reading dark blue is used,
- and comments pertaining to code are in a bluish ink for easier reading.
```

- The LOG uses the following coloring convention:

the log - is surrounded by a blueish frame

```
The source code and general log text are blueish.
Log NOTEs are green.
Log WARNINGs are violet.
Log ERRORs are red.
Log text generated by the user is purple.
```

Appendix B - install the SPF and packages

To install the SAS Packages Framework and a SAS Package we execute the following steps:

- First we create a directory to install SPF and Packages, for example: `/home/user/packages` or `C:/packages`.
- Next, depending if the SAS session has access to the internet:
  - if it does - we run the following code:

code: install from the internet

```
filename packages "/home/user/packages";

filename SPFinit url
 "https://raw.githubusercontent.com/yabwon/SAS_PACKAGES/main/SPF/SPFinit.sas";
%include SPFinit;

%installPackage(SPFinit)
%installPackage(packageNameYouWant)
```

  - If the SAS session does not have access to the internet we go to the framework repository:

    https://github.com/yabwon/SAS_PACKAGES

    next (if not already) we click the stargazer button [★] ;-) and then we navigate to the `SPF` directory and we copy the `SPFinit.sas` file into the directory from step one (direct link: `https://raw.githubusercontent.com/yabwon/SAS_PACKAGES/main/SPF/SPFinit.sas`). And for packages - we just copy the package zip file into the directory from step one.
- From now on, in all subsequent SAS session, it is enough to just run:

code: enable framework and load packages

```
filename packages "/home/user/packages";
%include packages(SPFinit.sas);
%loadPackage(packageNameYouWant)
```

to enable the framework and load packages. To update the framework or a package to the latest version we simply run:

code: update from the internet

```
%installPackage(SPFinit packageYouWant1 packageYouWant2 packageYouWant3)
```

See [**Jablonski 2020**], [**Jablonski 2021**], and [**Jablonski 2023**] for details.

Appendix C - safety considerations

The SPF installation process, in a "nutshell", reduces to copying the `SPFinit.sas` file into the packages directory. It is the same for a packages too.

You may ask: *is it safe to install*?

Yes, it's safe! When you install the SAS Packages Framework, and later when you install packages, the files are simply copied into the packages directory that you configured above. There are no changes made to your SAS configuration files, or autoexec, or registry, or anything else that could somehow "break SAS." As you saw, you can perform a manual installation simply by copying the files yourself. Furthermore the SAS Packages Framework is:

- written in 100% SAS code, it does not require any additional external software to work,
- full open source (and MIT licensed), so every macro can be inspected.

When we work with a package, before we even start thinking about loading content of one into the SAS session, both the help information and the source code preview are available.

To *read help information* (printed in the log) you simply run:

code: get help info

```
%helpPackage(<packageName>, <*|componentName|license>)
```

To **preview source code of package components** (also printed in the log) you simply run:

code: get code preview

```
%previewPackage(<packageName>, <*|componentName>)
```

The asterisk means "print everything", the `componentName` is the name of a macro, or a function, or a format, etc. you want see.

Appendix D - workarounds for DoSubL() sandwich limitations

Note: This section intention is not to explain all the details. Its purpose is rather to point possible directions and to show some examples (more code can be found in the SAS code files attached to the article).

It is possible to embed a single level `%if-%then-%else` conditional logic inside the `DoSubL()` sandwich with help of the `%nrStr()` macro function:

code: single level conditional logic inside doSubL sandwich

```
%let rc=%sysfunc(doSubL(%nrStr(
  options ps=min nonotes nomprint nosymbolgen nomlogic nosource nosource2;
  %if <CONDITION> %then
    %do;
      <4GL CODE FOR TRUE>
    %end;
  %else
    %do;
      <4GL CODE FOR FALSE>
    %end;
)));
```

The approach is possible because the code "inside" the sandwich is recognized as a "side session open code". Though the single level logic covers a vast area of possible programming scenarios and will for sure be useful, unfortunately nested logic and loops are not supported in open code. They both requires a macro.

One of possible resolutions is to "reverse" the code order by embedding the "sandwich" in the conditional logic or a loop. As simple as it seems it has a few drawbacks. The first one, `DoSubL()` is executed multiple times, which means: creating, opening, closing and removing side session multiple times what increases execution time (according to this best practice saying: "do a `do-loop` inside `dosubl()` but do not other way round"). The second, sometimes the logic of a program (loop in particular) cannot be executed in separate side sessions, for example when a macroloop is executed inside DATA STEP.

Two following macros: `%iff()` and `%doLoop()`, plus some experience with macroquoting, can be useful:

code: utility macros

```
%macro iff(cond, true, false) / secure;
%put NOTE:[&sysmacroname.] START ************************;
%if %sysevalf(&cond.) %then
%do;
%put # TRUE #;
&true.
%end;
%else
%do;
%put # FALSE #;
&false.
%end;
%put NOTE:[&sysmacroname.] END **************************;
%mend iff;

%macro doLoop(start, end, execute, by=1, index=i) / secure;
%put NOTE:[&sysmacroname.] START ************************;
%put NOTE- &=start. &=end. &=by. &=index.;
%do &index. = &start. %to &end. %by &by.;
%unquote(&execute.)
%end;
%put NOTE:[&sysmacroname.] END **************************;
%mend doLoop;
```

The above utility macros taken as they are and treated with the `%unhide()` macro won't give us a lot of 4GL "secrecy". But when used inside the "sandwich" they 1) became safe and 2) provides us with conditional logic and loops.

Example 1.

code: Do-Loop

```
%macro secretA(color,x) / secure;
%local rc;
%let rc = %sysfunc(doSubL(%nrstr(
  options ps=min nomlogic nosymbolgen nomprint nosource nosource2;

  %doLoop(1,&x.
    ,%nrstr(
      data &color._&t.;
        x = &t.;
        text = "&color.";
      run;
    )
    ,by=2
    ,index=t
  )
)));
%mend secretA;
```

Example 2.

code: Nested conditional logic

```
%macro secretB(x,y) / secure;
%local rc;
%let rc = %sysfunc(doSubL(
  %str(options ps=min nomlogic nosymbolgen nomprint nosource nosource2;)

  %iff(&x. > 0
    ,%iff(&y. > 0
        ,%str(
          data red3;
            x = &x.; y = &y.;
            text = "X and Y are greater than 0";
          run;)
        ,%str(
          data green3;
            x = &x.; y = &y.;
            text = "X is greater than 0";
          run;)
      )
    ,%str(
      data blue3;
        x = &x.;
        text = "X is NOT greater than 0";
      run;)
  )
));
%mend secretB;
```

Example 3.

code: Do-Loop inside DATA STEP

```
%macro secretC(x) / secure;
%local rc;
%let rc = %sysfunc(doSubL(%nrstr(
  options ps=min nomlogic nosymbolgen nomprint nosource nosource2;

  data %doLoop(0,&x.-1,%nrstr( part_&i.));
    set sashelp.cars;

    select;
      %doLoop(0
             ,&x.-1
             ,%nrstr( when(mod(_N_,&x.) = &j.) output part_&j.; )
             ,index=j)
      otherwise do;
          put "ERROR: wrong value!!";
          put _all_;
        end;
    end;
  run;
)));
%mend secretC;
```

INDEX