COBF

The C/C++ Sourcecode Obfuscator

A Program
By

Bernhard Baier
email: bernhard.baier@gmx.net
WWW: http://home.arcor.de/bernhard.baier

V 1.06
2006-01-07

1. Introduction

COBF (aka C-Obfuscator) is a program which manipulates C or C++ sourcefiles in a way that they aren't readable by human beings; but they remain compilable.

The benefit of COBF lies in the distribution of programs (freeware, shareware, commercial software etc.):
The distribution as a executable binary is often too inflexible (especially for Unix or Windows NT, because it is normally not feasible to support all possible platforms); on the other side one might not give away the sourcecode (e. g. it looks poor [badly commented, Spaghetti code], or you just want to prevent plagiarism).

Here COBF can play an important role: It removes all comments and sourcecode formattings and renames all variable and function identifiers. In the future a special add-on filter will break up all *for*, *while* and *if* statements into *goto* statements.

Here is an example:

```
#define MAX_INDEX 10
int my_output()
{
      int count;
      for (count = 0; count < MAX_INDEX; ++count)
            print_result(count);
}
```

becomes something like that:

```
a l47(){a l234;g(l118=0;l118<10;++l118)l93(l118);}
```

You need only little imagination to realize that programs prepared in such a way are practical unreadable. Conclusions to the original contents are only possible with great effort (similiar to dissassembling on machine language level).

2. Installation

There is no special installation procedure for COBF. Simply unpack the archive *cobf.zip* to a newly created directory. See *files.txt* for a complete content description list of the *COBF* archive!

2.1. Using an existing executable of *COBF*

In the *src\win32\Release\* directory you can find an executable of *COBF* for Windows95/NT/XP.

For a first try run *demo_test.bat* in the *demo/*-directory if you are using Windows 95 / NT / XP. (Please have before a look into the *pp_ger_msvc.bat* file in the *etc* directory whether it fit to you local compiler/preprocessor installatation

2.2. Compilation of *COBF*

If you want to make changes on the COBF sources or if there is no binary for your machine you must compile the *COBF* sources

For Unix/Linux you might start with the makefile located in the src/unix directory (Please keep in mind that the makefile expects that all C++ source files have the extension ".cc" instead of ".cpp". Also it has to be noticed that the GNU C/C++ compiler has sometimes problems with source files having linefeed + carriage return as line delimiter (as it is ususal for DOS), not only a line feed. This leeds especially to problem with macros spanning over more than one line with the '\' delimiter. Convert such files with the *dos2unix* command.)

If you want to make changes on the file *scan.l* you need additionally a LEX-compatible scanner generator (e. g. GNU-*flex*). Take notice: The used LEX must generate ANSI-C, not old-style K&C function definitions! However if this is the case you must manually change the generated file *lex_yy.c* or the corresponding LEX library files. The reason for this is that *lex_yy.c* will be compiled as C++ sourcecode and C++ doesn't allow old-style function definitions.

2.3. Creating the Script for Calling the Preprocessor

COBF needs for full functionality a C / C++ preprocessor which should be shipped as a command line tool with your C / C++ compiler. *COBF* does not call the preprocessor directly but it uses a special shell script (in the standard COBF package the script names start with *pp_** and are located in the *etc/* -directory – normally they must be adapted to your operating system and compiler environment.)
This preprocessor shell script expects 2 arguments:

```
pp_<xyz>(.bat) Inputfile Outputfile
```

COBF expects that the *pp_** script writes the preprocessed *Inputfile* to *Outputfile*. There is no need to tell the invoked preprocessor something about include pathes; the include file handling is completly done by *COBF*.

The *pp_borlandc.bat* script was tested with the Borland C/C++ preprocessor. The *pp_ger_msvc.bat* script was tested with the preprocessor coming with Microsoft Visual Studio. The *pp_cc* script was tested with the GNU C preprocessor.

3. Invocation and Configuration of *COBF*

3.1. Introduction

Please read the following instructions carefully. Otherwise your shrouded sources won't be compilable!

IMPORTANT! Look after every *COBF* session to the protocol file *cobf.log*!! Take notice of eventually occuring warnings!

Now lets look at a simple example. We want to shroud the following sourcefile *test.c* (you can find this example in the *demo*-directory):

```
/* test.c - simple test program for COBF */

#include <stdio.h>
#ifdef unix
#define MAX_COUNT 10
#else
#define MAX_COUNT 20
#endif

int main()
{
      int i;
      for (i = 0; i < MAX_COUNT; ++i)
            printf("Hello %d!\n", i);
      return 0;
}
```

For example the *demo_test.bat* (should work with all Windows Osses like 95 or XP) has the folling content:

```
..\src\win32\release\cobf @demo_token.inv -o output -b -p
..\etc\pp_ger_msvc.bat test.c
```

After invoking the batch file you should find the following output in the *output* directory (under the *demo* directory)

```
/* COBF by BB -- 'test.c' obfuscated at Sun Jan 21 18:44:05 1996
*/
#include<stdio.h>
#include"cobf.h"
#ifdef unix
#define b 10
#else
#define b 20
#endif
c e(){c a;d(a=0;a<b;++a)f("\x48\x65\x6c\x6c\x6f\x20\x25\x64\x21\n",a)
;g 0;}
```

This shrouded sourcefile *test.c* in the *output* directory is compilable and yields the same object file as the original *test.c* sourcefile!

You may wonder that seemingly such essential C keywords as *if* or *for* disappeared. The disbandment is simple: Look at the headerfile *cobf.h* in the output directory:

```
/* COBF by BB -- obfuscated at Sun Jan 21 18:44:05 1996
*/
#define c int
#define e main
#define d for
#define f printf
#define g return
```

While local identifiers (here in the example the index variable *i*) are completly replaced by a new identifier, some other identifiers like the C tokens (*if*, *for*) or identifiers with extern linkage (*main*, *printf*) are only exchanged on preprocessor level.

Now let's take a look on the generated logfile *outdir/cobf.log* (the comments were later inserted):

```
Logfile for shrouding at Sun Jan 21 18:44:05 1996

output dirctory: output
shell script for performing preprocessing: ..\etc\cobf_pp.bat

/* Part 1 */
List of shrouded sourcefiles:
(H) included by another sourcefile (P) preprocessed header
(S) separately shrouded header     (X) external header
test.c              includes: stdio.h[X]

/* Part 2 */
No headerfile was included by cobf!

/* Part 3 */
The following external includefiles were found:
(all external visible identifiers in this files MUST be defined as tokens
for cobf!!)
stdio.h

/* Part 4 */
No headerfile was separately shrouded!
```

3.2. Invocation

General syntax:

```
cobf options [@invocationfile] ... filename [...]
```

Arguments in an invocationfile are treated as if they were directly passed by command line.

Main options:

-hi *filename*	specifies *filename* as an internal includefile, i. e. a headerfile which will be not separately shrouded and instead included by *COBF*
-hs *filename*	specifies *filename* as a separate includefile, i. e. a headerfile which will be separately shrouded
-i *path*	adds *path* to the search path list for source and header files
-m *filename*	adds identifiers in *filename* to the system macro list (will not be shrouded or preprocessed)
-mi *filename*	Specifies the identifier mapping file *filename*. Each line must consist of the original identifier and then the new identifier separated by one or more whitespaces. Indetifier clashes to shrouded identifiers are detected.
-t *filename*	adds identifiers in *filename* to the keyword list (only shrouded by preprocessor)
-o *outputdir*	destination for temporary and shrouded files
-p *batchfile*	script for invoking the preprocessor (1st argument input file, 2nd input file)

Output options:

-b	Preserve original filenames in output directory (per default the shrouded files are renamed to a?.ext where '?' are consecutive numbers beginning with 0)
-c *filename*	concatenate all sourcefiles to one file *filename*
-g	do not shroud strings (per default strings are obfuscated by using the ASCII-equivalent in hex for each character)
-dd *filename*	dump identifier dictionary to *filename*
-dm *filename*	dump identifier mapping list to *filename*
-r *right margin*	specifies the left most column for the shrouded sourcefiles
-u	Treat keyword lists (specified with the -*t*-option) in the same manner as system makro lists (specified with the -*m*-option). The result is no shrouding on preprocessor level.
-x *prefix*	For all output files add string *prefix* to each identifier
-xn	do not use characters 'a' - 'z' after the prefix for the most used identifiers

Debug options:

-a	Do not delete temporary files generated by each *COBF* pass
-d	include debug comments into shrouded files
-di	Add original identifier to each shrouded identifier
-f	filter mode (no shrouding)
-n	no preprocessing
-s	stop after pass 1 (optional -s0 to -s5)
-v	verbose (optional -v1 to -v9 to increase verbose level)

3.3. The includefile handling of *COBF*

COBF distinguishes between three kinds of includefiles:

External Includefiles

We call an includefile an external includefile, if it isn't explicitly passed to *COBF* as a comand line parameter.

So external includefiles are not an intrinsic component of your sourcefile configuration. Rather it is expected that they are per default existing on the target platform. Examples are *stdio.h* (ANSI) or *unistd.h*.(Unix).

All global visible identifiers in external includefiles ***must*** be declared via the *-t-* or *-m-* command line option. The file *cansilib.tok* contains a subset of reserved identifiers conforming to the ANSI-C-standard. It is not considered that ANSI-C generally reserves all identifiers for example beginning with *is* (like *isalpha* etc.) because it is actually only possible to declare explicitly known identifiers. But in practice this shouldn't be a problem because all "fixed" identifiers of the ANSI-C-standard library declared in the well-known headers like *stdio.h* or *stdlib.h* are (hopefully) registered.

There are actually no corresponding token lists for C++ or generally for operating system dependend includefiles (e. g. *dos.h* for DOS, *windows.h* for Windows 3.1/95/NT or *unistd.h* for Unix). If you have written such lists please let me know.

External includefiles are marked with *[X]* in the protocol file *cobf.log*.

Internal Includefiles

We call an includefile an internal includefile if it is passed via the *-hi*-Option to *COBF*. If *COBF* finds in an early stage of analyzing a sourcefile an #include statement with an internal headerfile, it simply substitutes (without help of the preprocessor!) the #include statement with the contents of the includefile. Recursive including is prevented. The resulting sourcefile will be preprocessed if not the *-n*-option is given (no preprocessing)

Internal includefiles are marked with *[P]* in the protocol file *cobf.log*.

Separate Headerfiles

We call an includefile a separate includefile if it is passed via the *-hs*-Option to *COBF*. Separate includefiles will not be preprocessed. Consequently #define or #if-Statements in the remaining sourcefiles which contain macros defined in separate includefiles will not be preprocessed.

Separate includefiles are marked with *[S]* in the protocol file *cobf.log*.

4. Strategies for Obfuscation of C/C++ Sourcecode

How Do I Handle <...> Includes?

Headerfiles, which export functions, macros or variables available on the target platform should be treated as external includefiles (see above).

Such headerfiles are normally included with angle brackets, like that:

```
#include <stdio.h>
```

So

- do not pass such headerfiles via the command line to *COBF*
- declare all identifiers exported by such headerfiles via the *-t* (or *-m*-)option

This guarantees that that your sourcefiles remain compilable on the target platform

- without unresolved externals reported by the linker
- independend of the fact whether a function is implemented as a "real" function or as a macro.

How Do I Achieve the Best Possible Compression and Encryption Of My Sourcefiles?

Use the *-a*-Option; all sourcefiles will be concatenated to one file and the shrouded as a single piece of sourcecode.

Caution: It might be neccessary to make changes that your sourcefiles are compileable as a single piece of sourcecode. Possible pitfalls are for example repeated includefiles, redefinition of macros or multiple declarations of static variables.

How Do I Preserve the Original File Structure?

To preserve the original file structure do the following:

- declare your own headerfiles as separate includefiles
- to prevent the renaming of your sourcefiles use the *-b*-option

Take care! *COBF* doesn't invoke the preprocessor for separate includefiles; preprocessor statements in other sourcefiles with dependencies to macros defined in separate includefiles won't be shrouded, too. So conclusions to the original sourcecode may be easier. So it's recommended to use separate includefiles only where it is necessary (see next chapter for an example).

How Do I Preserve Portability and Configurability of My Sourcefiles?

COBF distinguishes between system macros of 1st and 2nd order.

The so called 1st order system macros are declared with the -*m*-option (examples are *__LINE__*, *unix*)
A macro will automatically added to the list of 2nd order system macros by COBF when

- the macro is #defined in a separate includefile
- the mcro is #defined in a #if-block with the #if-statement containing a system macro.

Then the following rules apply:

- Preprocessor statements containing system macros (either 1st oder 2nd order) are not preprocessed by COBF.
- *COBF* does not shroud 1st order system macros
- *COBF* substitutes 2nd order system macros on the preprocessor level (as an consequence *COBF* generates for each 2nd order system marco a #define-statement in the headerfile *cobf.h*)

An example:

```
#ifdef unix
#define PATH_SEPARATOR "/"
#else
#define PATH_SEPARATOR "\\"
#endif
```

After shrouding it might look like this:

```
#ifdef unix
#define l4711 "/"
#else
#define l4711 "\\"
#endif
```

In this example *unix* is a 1st order system macro. Therefore *COBF* declares automatically *PATH_SEPARATOR* as 2nd order system macro. None of these two macros must be proprocessed by COBF itself to preserve portability on different target platforms; but in this example it is suitable to exchange the macro *PATH_SEPARATOR* with a less-readable identifier.

The concept of system macros is also usefull to preserve source code configurability.

An example:

```
/* File myconfig.h */
#define MAX_WINDOWS 20
#define MAX_COLORS     256
...
#if MAX_COLORS > 256
#include "rgb.h"
#endif
...
```

It should be posssible for the receiver of the shrouded sources to configure the sources in a pretended way.
There are three posibilities:

- define all macros in myconfig.h explicitly as system macros
- declare myconfig.h as a separate includefile; the macros there defined will be renamend, but they won't be preprocessed
- declare blocks of system macros with the special system macro __COBF__:

An example:

```
#if !defined __COBF__   /* __COBF__ should be never defined so the
                           condition is always true! */

#define MAX_WINDOWS     20
#define MAX_COLORS     256

#endif
```

The trick is that *COBF* internally keeps *__COBF__* as a 1st order system macro. Additionally all macros which are defined in a #if block containing *__COBF__* were also declared as 1st order system macros (in contrary to the above rule, normaly they would declared as 2nd order system macros).
The benefit in this example is that the identifiers *MAX_WINDOWS* and *MAX_COLORS* keep their original names after the shrouding too.

5. Technical Details

There are some details about internal procedures the interested reader may want to know.

5.1. C or C++

For *COBF* it makes no difference whether the sources are coded in C or C++ because *COBF* analyzes the sources only on the token level. There is no syntax check. A consequence is, for example, that C++-style comments (// ..) are allowed for C programs too.

5.2. Whitespace Compression

A C-Obfuscator shall remove all comments and as much as possible whitespaces from the program sources.

The removal of comments is easy. One have to care a little more to remove all unneeded whitespaces.

It's obviously erroneous to change

```
int a;
```

to

```
inta;
```

It is not so obviously, that this C++ fragment

```
List<List<String> >
```

shall not be changed to

```
List<List<String>>
```

because ">>" is a new token (the shift operator).

A C analogue:

```
a = b + +3;
```

Here it's not allowed to remove the space between the plus signs.

Here is another strange combination:

```
extern double a, *c;
b = a / * c;
```

COBF treats the source file as a sequence of tokens separated by whitespaces (normally blanks and newlines) or comments (C-style and C++-style comments)

COBF inserts a blank between two tokens if otherwise

- two letters or numbers
- two equal charactes
- a asterisk and a non-letter (or vice versa)

would immediately succeed.

I hope I considered all possible "strange" character and token combinations which can occur in C or C++. On the contrary it should not be difficult to find examples where the above rules are inserting unneeded blanks.

5.3. Identifier renaming

The basic idea of a C obfusctor is quite simple:[1].

Let's take a look at the follwing program section:

```
int main()
{
      int i:
      i = 7;
      {
            int i;
            i = 8;
            printf("%d\n",i);
      }
      printf("%d\n", i);
}
```

For *COBF* it is completly irrelevant that *i* is the identifier of two different objects. It is sufficient to replace *systematicly* all identical identifiers with another ones (which should not occure elsewhere in the original sources).

[1] The C Obfusactor OPQCP (OpaqueCopy) from Russ Fish (fish%kzin@cs.utah.edu) inspired me to do the same.

6. Troubleshooting

If a program you obfuscated with COBF isn't compileable you should first look at the file *cobf.log*. There is a brief summary what *COBF* has done when shrouding your sources.

Here is a short list of possible traps and pitfalls and hints:

- use the –di option to get better understandable compiler error messages (don't forget to remove this option for the final obfuscating run!)
- Not all "public" symbols in the external headerfiles (marked with [X] in cobf.log) are known to COBF. See the section "How Do I Handle <...> Includes?" for details.
- Check your general obfuscating stretegy: What is is an external, internal or separate header file?
- There are problems with macros containing the "#" or "##" operator or identifiers builded with the toking pasting operator. Look a at the affiliated sections in the cobf.log file!
- A system token (e. g. a C key word like *const* or *void*) gets redefined. Increase the verbose level via the *-v* option to find this out!
- You are using not fully supported #include statement constructs e. g. the include filename is a #defined value
- A #pragma directive may in some strange situations not be properly treated by COBF
- You are using inline assembling
- You encountered a bug of *COBF* (to isolate the problem use the debug options, e. g. "–a" to have a look at the temprary files COBF generates)

In every case, by increasing the verbose level and using the *-d* option (debug info is inserted in the shrouded source file by comments) you should be able to encircle the problem.

Acknowledgements

Barry Corlett (barry@bramley.spacenet.de) proof-read this documentation.
My brother Thomas (thomas.baier@stmuc.com) provided the first useful application of COBF: The source distribution of 3DTO3D, an execellent 3D format conversion tool.
Alexei Kostin reported the bug concerning the wrong handling of 8 bit character input.
Claudius Schnörr proposed source code changes for being compliant with ANSI C++ and reported a bug (clash between shrouded vs. system token identifiers)

Change History

Version	**Date**	**Comment**
1.0		1st public version
1.01	1998-03-22	minor fixes
1.02	2000-10-31	added -x option (prefix)
1.03	2002-05-01	- Documentation and installation cleanup - default PC target is now MS Visual Studio - source code change: when opening a stream now explicit ios flags will be passed (old method seems to open the stream with wrong default settings ..)
1.04	2003-01-01	bugfix: now 8 bit input is correctly handled (due to switch to GNU-Flex 2.5)
1.05	2006-01-02	Adaptation of the source code for ANSI-C++ (tested with Visual Studio .NET 2003 and gcc 3.4.4) New command line options '-xn', '-di' and '-dd' Bugfix: identifier clash between system token list and shrouded identifiers now detected
1.06	2006-01-07	New command line options '-mi' and '-dm' Changed semantics for building the shrouded identifier: default prefix is now 'l' which can be overwritten by the '-x' option