

Inexpensive protection of your source code

Snob

Simple Name Obfuscator

Reference Manual

Version 1.1



MacroExpressions
<http://www.macroexpressions.com>

Table of Contents

0. SNOB: EXECUTIVE SUMMARY	1
0.1 WHAT IS IT?	1
0.1.1 <i>Code in a script language</i>	2
0.1.2 <i>Compiled code: Library deliverables</i>	2
0.2 WHAT DOES SNOB CONSIST OF?	2
0.3 WHAT MAKES SNOB DIFFERENT?	2
0.4 PLAIN-TEXT EULA	2
1. THE SNOB SOLUTION	3
1.1 LANGUAGE-DEPENDENT CONFIGURATION FILES	3
1.2 EXTENSION-INDEPENDENT CONFIGURATION FILES	7
1.2.1 <i>reserved.snob</i>	7
1.2.2 <i>APIfiles.snob</i>	8
2. HOW SNOB TOKENIZES TEXT	8
3. HOW SNOB OBFUSCATES A NAME	9
4. HOW TO INVOKE SNOB AND WHAT IT PRODUCES	9
5. HINTS AND TIPS	10
5.1 STANDARD AND THIRD-PARTY NAMES	10
5.2 COPYING TEXT FILES	10
5.3 DEALING WITH MANGLED NAMES	11
6. SNOB ERROR REPORTING	11
7. KNOWN ISSUES	12

Acknowledgement

Regular expression support is provided by the PCRE library package, which is open source software, written by Philip Hazel, and copyright the University of Cambridge, England. See

<http://ftp.csx.cam.ac.uk/pub/software/programming/pcre/>

0. Snob: Executive Summary

0.1 What is it?

Snob is a simple (or stupid, if you like it better) name obfuscator tool. It replaces meaningful names (identifiers) in your source code with meaningless and similarly looking ones. This makes the code very hard to read for a human being (but not a computer). As a usual practice, name obfuscators are used when the source code containing proprietary knowledge needs to be distributed.

When a reader encounters your obfuscated code, at the very least he understands that you wanted to protect it and that it is not really for human eyes. If he still wants to figure out how the code works, the task is much harder if the names are meaningless.

Here are a few cases where you may favor name obfuscation:

0.1.1 Code in a script language

If your deliverable is, say, in JavaScript, everyone can see client-side code by looking at the source of the Web page. Even with the server-side scripts, you do not necessarily want your customer to understand the inner workings of the code.

Sometimes, a piece of your project is just handy to write in Perl or a shell script language. When the delivery time comes, you may regret that you need to expose your code. (Granted, those things are cryptic enough in themselves but there are quite a few fluent speakers of those languages.)

Same goes for Python, Basic, and pretty much any interpreted language.

0.1.2 Compiled code: Library deliverables

In a way, the only case where none of your names are exposed is a standalone executable (OK, maybe with calls to standard or third-party shared libraries/DLLs).

If your deliverable contains a library, the latter is going to have some of the original code's names in its relocation table. And, of course, your shared library (or DLL, as the case may be) exposes names of exported interfaces.

If you deliver a static (linkable) library, it has names internal to the library in relocation table(s).

0.2 What does *Snob* consist of?

Snob is a standalone executable which doesn't require any installation or uninstallation. It does rely on project configuration files as described in further sections.

0.3 What makes *Snob* different?

Yes, there are name obfuscators out there. So, why another one?

The answer is, the known name obfuscators are too expensive and too inflexible. For example, they do not handle projects written in multiple programming languages, or they do not obfuscate the elements of aggregate data types (like C's `struct`).

Snob puts you in control: you tell Snob what the definition of a name is, and Snob does the rest. Much of Snob configuration is specific to the programming language(s) of your project; Snob is distributed with a somewhat simplistic sample configuration file for C. Any donated language support configurations will be posted on MacroExpressions website for free sharing among Snob users.

0.4 Plain-text EULA

Here is the Snob End-User License Agreement in plain English:

You (whether you are a physical person or any kind of organization) get a non-exclusive license to use Snob at your own risk for the sole purpose of obfuscating any files whatsoever. However, you may not

distribute the obfuscated files in source, compiled or any other form as part of your product, unless you purchase a distribution license.

The distribution license is dirt cheap and allows you to distribute the files you obfuscated in any form, but it limits the files you can obfuscate to those which you have legal rights to do so. Legal arrangements on that do vary, and you are responsible to find out whether you are violating any.

You further agree to hold MacroExpressions harmless regardless of any malady whatsoever the use of Snob caused you, including but not limited to a spoiled day, computer crash or lost business.

You cannot use Snob in jurisdictions where provisions of this license agreement are in contradiction with any law.

Enjoy!

1. The Snob Solution

Snob presumes that your project has its root directory, and that every file and subdirectory (recursively) belongs to the project – and that nothing outside this directory hierarchy belongs to the project.

Part of your project is Snob configuration files. They all have the extension `.snob`.

1.1 Language-dependent configuration files

NOTE. This section assumes that you are familiar with regular expressions, and, in particular, with its Perl variety. An independent reference of regular expressions used in Snob is available in the distribution and online. It is an adaptation of the PCRE regular expressions reference.

Snob relies on association of a filename extension with a corresponding configuration file name (presuming that the filename extension corresponds to a language, such as `.java` for a Java source file).

If somewhere in the project directory hierarchy there is a file with extension `.ext`, Snob looks for a configuration file named `dotext.snob` (such as `dotc.snob` for `.c` files or `dotfoo.snob` for `.foo` files). So named configuration file is searched from the directory where the `.ext` file was found all the way up to the project's root directory. If `dotext.snob` was not found, Snob looks, as a last resort, in the directory where the Snob executable, `snob.exe`, was started from. The first found configuration takes effect. If the configuration file `dotext.snob` was not found, Snob skips the processing of the `.ext` file it started with.

To put it in other words, extension-dependent configuration files have a fixed name `dotext.snob` for the filename extension `.ext` and affect `.ext` files in the project directory and its subdirectories until overridden with another `dotext.snob` file in some subdirectory, which acts on directories down the hierarchy until overridden, etc.

`dotext.snob` files are plain text files containing configuration options and optional comments. A comment is any line that is not recognized as a configuration option. For future compatibility, it is advisable to start a comment line with a blank space.

The following configuration options are recognized:

name=<regular expression>

This option defines what a name is outside literal strings and comments. Here is an example from a `dotc.snob` file:

```
name=\b[_A-Za-z][_0-9A-Za-z]*
```

which says that a name is a sequence of letters, underscores and digits starting with a non-digit and on a word boundary. Note that, strictly speaking, this definition is correct for C but incorrect for Snob. The reason is that Snob is unaware of C line continuation syntax (a backslash followed by a newline). Those who break lines in the middle of a token (before they are excommunicated from programming) should take this into account.

Since line continuation complicates the regular expression and slows down its processing, we will ignore it in all C tokens *except* single-line comments (from `//` to the end of line, which are recognized by most compilers) where this omission may have a devastating effect. All other broken tokens will result in compilation errors of the obfuscated code.

For comparison, here is a definition of a Perl name (which would go to a `dotpl.snob` file):

```
name=(?<=[${}%@])[_A-Za-z][_0-9A-Za-z]*
```

i.e., a sequence of letters, underscores and digits, starting with a letter and following one of `$`, `%` or `@`. As you can see, in Perl's array `@foo`, only `foo` is considered a name. The reason is that `$foo[$bar]` refers to the same entity (`foo`) so the name must be identical in both instances.

There may be more than one definition of a name; extra definitions are considered as alternatives.

keyword=<regular expression>

This option specifies language tokens (typically, keywords) which Snob would otherwise confuse with a name. Here is a sample line from `dotc.snob`:

```
keyword=auto|break|c(ase|har|on(st|tinue))|d(efault|o|ouble)
```

Again, there can be more than one `keyword=` option; different options considered as alternatives.

For comparison, Perl's `dotpl.snob` does not need `keyword=` option because Snob cannot confuse Perl name (as defined above) with a Perl keyword. For the same reason, C's token `->` doesn't have to be listed as a keyword.

Note however that `keyword=` option may specify text covering more than one token of the language. For purely aesthetic reasons, in such cases a synonym option `ignore=` is recommended.

ignore=<regular expression>

This option is similar to the `keyword=` option in that it defines a piece of text that Snob must preserve. In fact, its action is identical to that of the `keyword=` option and it can be used to define pieces of text covering more than one token.

Here is an example from a `dotc.snob` which ignores angle-bracket `include` statements:

```
Ignore <>-includes
ignore=(?m)^[ \t]*\#[ \t]*include[ \t]*<.*?>
```

reserved=<regular expression>

This option is similar to the `keyword=` or `ignore=` option in that it defines what Snob will ignore. The difference is that the option acts on the whole project rather than on subdirectory hierarchy starting with where the option is found.

Here is how it works:

Assume for example that there is a `dotfoo.snob` file in a subdirectory `foo_dir` of the project directory, and that there is a line `reserved=blah` in that file. Assume further that there is a file with the extension `.foo`, say, `bar.foo`, for which our `dotfoo.snob` is the configuration file (that is, `bar.foo` is in `foo_dir` directory or it is down `foo_dir` directory tree and the first `dotfoo.snob` found going up from `bar.foo`'s directory is in `foo_dir`.) In this case, this `dotfoo.snob` configuration file takes effect and, in particular, its line `reserved=blah` takes effect. The meaning of it is that Snob will not take `blah` as a name in any file with any extension in any directory of the project directory hierarchy.

However, if there is no `.foo` file for which our `dotfoo.snob` is a configuration file, then no option in this `dotfoo.snob` has an effect and, in particular, the option `reserved=blah` is not seen.

If this looks rather complicated it, well, is. The `reserved=` option is somewhat advanced and its intent is to indicate what words are not names when a language is brought into the project.

Example from a `dotc.snob`:

```
reserved=to(upper|lower)|ato(f|i|l)
```

Consider, for instance, an Assembler language project, where `atoi` doesn't mean anything special and, when encountered, can be considered a name. Now you might introduce C language modules into the project, and there `atoi` is a name of a standard library function. The deal is, however, that an Assembler module can now call this very `atoi` function. Thus, `atoi` must be a reserved word for both C and Assembler, but only if there are C modules in the project.

Now, consider introduction of C++ modules in this mixed Assembler and C project. In this scenario the word `new` is C++ language keyword and a valid name in both Assembler and C. Should then `new` be covered by `keyword=` option or by `reserved=` option? Let's consider our options (no pun intended) here.

If we have a line `reserved=new` in an active `dotcpp.snob` configuration file, then the net effect of it is that the name `new` will not be obfuscated in Assembler and C modules. No harm done, except that the name is revealed. If, instead, we have a line `keyword=new` in `dotcpp.snob`, then `new` will

be treated as a name in Assembler and C and as a non-name in C++. Whether we can do that depends, of course, on whether Assembler or C modules can use the same C++'s `new`. In this case, the answer is negative, so `new` should be in a `keyword=` option of a `dotcpp.snob` file. This issue deserves a closer examination because of header (.h) files shared between C and C++ modules. We'll cover this issue again when describing `include=` option.

comment=<regular expression>

This option specifies the areas of text which Snob will throw out entirely from the source file. Here is an example from a `dotcpp.snob` file:

```
Single-line comment with continuation lines
comment=(?ms) /\.*?(?<!\)\$
Multi-line comment
comment=(?s) (/\/.*?\/)
```

As you can see, multiple `comment=` options can be used and are considered alternatives. To demystify the regular expressions there:

Single-line comment definition says that a comment is any text from double slash to the first occurrence of a newline not preceded by a backslash character.

Multi-line comment definition says a comment is any text enclosed between `“/*”` and the closest to it `“*/”`.

string=<regular expression>

This option defines text that Snob has to copy verbatim to the output (obfuscated) file. Its typical use is to preserve the literal strings. Here is an example from a `dotc.snob`:

```
String goes from one unescaped quote to the next
string=(?<!\)\\".*?(?<!\)\\"
A character is a string for our purposes (with single quotes)
string=(?<!\)\'.*?(?<!\)\'
```

string=<regular expression> name=<regular expression>

Some languages, such as, e.g., shell script languages or Perl, recognize names within a literal string; typically, a name must be preceded by an unescaped dollar sign (\$). This option allows to define a string along with instructions to Snob on how to spot names in it.

Here is an example from a hypothetical `dotsh.snob` where a double-quoted string may contain uppercase names preceded by a dollar sign:

```
String goes from one unescaped quote to the next
Name within string must be preceded by an unescaped $.
string=(?<!\)\\".*?(?<!\)\\" name=(?<=(?<!\)\\$) [_A-Z] [_A-Z0-9]
```

Note that there must be a single space character between the end of definition of a string and the `n` of the `name=` option.

include=<pathname >

This option instructs Snob to read configuration options from the specified file(s). For instance, in a C project, configuration file `doth.snob` for the `.h` (header) files might contain a single line:

```
include=dotc.snob
```

This will make `.h` files inherit configuration options of `.c` files.

Here is how Snob searches for the file to include.

If the pathname is a relative pathname (i.e., doesn't contain drive letter or share host name and doesn't start with a forward or back slash), then it is considered relative to the directory where the configuration file containing the `include=` option is located. E.g., if a `dotfoo.snob` is located in the Foo directory somewhere in the project directory tree and contains the line

```
include=..\..\MyConfigFile
```

then Snob looks for `MyConfigFile` in the parent directory of the parent directory of the Foo directory.

Snob doesn't check that such a directory is under the project directory hierarchy.

If the pathname is not a relative pathname, Snob considers it a complete file specification. For instance, the line

```
include=E:MyConfigFile
```

instructs Snob to read configuration from the file `MyConfigFile` located in the current working directory of the drive `E:` whatever that directory might be. That is so even if the project directory is on the drive `E:` as well.

In either case, if Snob fails to open the specified file, it issues an error.

use=<pathname >

This option is almost identical to the `include=` option. The only difference is that the pathname must be a relative pathname and it is considered relative to the directory where `snob.exe` was located. For instance, if `snob.exe` is located in `C:\Snob` then the line

```
use=conf\MyConfigFile
```

instructs Snob to get configuration options from the file `C:\Snob\conf\MyConfigFile`.

1.2 Extension-independent configuration files

Snob supports additional configuration files with fixed names `reserved.snob` and `APIfiles.snob`. They are described below.

1.2.1 reserved.snob

If a file named `reserved.snob` exists anywhere in the project directory hierarchy, each non-empty line of it is considered to be a name for Snob to preserve in any of the project's files. It is very similar to `reserved=` option in the extension-specific configuration files but it acts independently of whether any specific file extension is actually used in the project. It is also much faster for Snob to process.

There can be several `reserved.snob` files in various project subdirectories; they act cumulatively.

1.2.2 `APIfiles.snob`

If a file named `APIfiles.snob` exists anywhere in the project directory hierarchy, each non-empty line of it is considered to be a pathname of a file that Snob must treat in a special way.

As the name implies, `APIfiles.snob` is intended to contain the names of files exposing API (Application Programmer's Interface) to your customer. An API file therefore must not be obfuscated at all and all its comments must be preserved. In other words, an API file must be copied to the obfuscation location verbatim. For this to work correctly, Snob must figure out what names are there in this API file and remember those names as not subject to obfuscation.

Pathname specification can contain wildcards (* and ?) in the filename part of it, which will cause all matching files to be considered API files. However, Snob will *not* report an error if it finds no matches to the wildcard specification.

Snob slightly stretches this notion of API files in the following way:

If the pathname is a bare filename, without any path information, then it is considered to be an API file in the directory where the `APIfiles.snob` is located. It will be copied verbatim and the names found in it will be treated as not subject to obfuscation.

If there is any directory (or drive) information in the pathname, then the names found in it will still be learned as not subject to obfuscation, but the file itself is not remembered in any special way. If the pathname points outside the project directory hierarchy, that's exactly the desired effect. If, however, the pathname points inside the project, Snob will process the file normally. That is, if the file's extension is unknown to Snob, it (the file) will be skipped. Otherwise, Snob will attempt to obfuscate it, but since the file will have all its names not subject to obfuscation, the net effect will be that the file will be stripped of its comments.

2. How Snob tokenizes text

For Snob, the text in a source file consists of tokens as specified in the configuration file:

- keyword
- ignore
- string (perhaps, with names)
- comment
- reserved (name)
- name
- and the rest of the text.

Snob starts from the beginning of the source file as the current position and finds the match with the least offset from the current position. If there is more than one match, the type of match higher in the list above is selected. E.g., if a piece of text matches definitions of a keyword and a name, it is considered a keyword. Similarly, if a text matches an `ignore` and a `comment`, it is considered an `ignore`.

The latter has a curious application for UNIX shell scripts, where the “#!” comment specifies the shell interpreter for the script. The problem is that since Snob removes comments, it leaves the script without an indication of what interpreter should run it. A solution is to declare a “#!” comment in `ignore=`

option. Then Snob will match a “#!” comment as an `ignore` and as a `comment`. Since `ignore` is higher on the priority list above, a “#!” comment will be ignored by Snob and thus left intact. (A simpler and faster solution is of course to define comments, for Snob purposes, as not starting with a ‘!’.)

When the winning match is selected, the segment of the source file between the current position and the beginning of the match is copied to the output file.

Then, the winning match is processed as follows:

If it is a comment, it is skipped and not copied to the output file (unless the file is remembered as API).

If it is a name and it if it can be obfuscated, the obfuscated name is copied to the output file

If it is a string, it is copied to the output file while replacing obfuscatable names in it (if any)

In all other cases, the matched text is copied to the output file.

Finally, the current position is moved to the end of the winning match and the process repeats until the end of the file is reached.

3. How Snob obfuscates a name

When Snob encounters a name in any project file with known configuration, it assigns to this name a unique number – essentially, a counter of different names. When Snob replaces the name, the replacement is a fixed prefix, followed by a fixed-width hexadecimal representation of the number assigned to the name, followed by a fixed suffix.

By default, the prefix is “C”, the width is 8 and the suffix is empty, and in the beta version this cannot be changed.

Example: if the name `myname` was assigned the number 17, it will be replaced by `C0000011` (11 is a hex 17).

4. How to invoke Snob and what it produces

Snob is invoked from the command line with two arguments: the project directory name and the target directory name, with an optional third argument `--preserve-lines`. The target directory must not exist. If the third argument is supplied, multi-line comments will be not just removed but replaced with the appropriate number of empty lines, so that obfuscation would preserve line numbering.

Examples:

```
snob myprojects\project1 C:\obuscate\project1obf
snob . ..\..\obf --preserve-lines
```

Snob creates the target directory and reproduces there the directory subtree of the project directory. Then it copies the files it knows about from the project directory subtree to the corresponding location in the target directory. Files with unknown extension (like `.snob` configuration files or object files) are not copied. While copying, Snob replaces names with their obfuscated versions and removes comments as instructed by `.snob` configuration files throughout the project directory subtree.

Finally, snob writes the file `projmap.snob` to the target directory. `projmap.snob` is a text file each line of which comprises a pair of obfuscated name and its original name, the two separated by “:”, like

```
C000000C9 : SNOBEVENT_OBFUSCATE_FINISH
```

In addition, Snob prints a fair amount of information to the standard output: directories it visits, files it processes, skips or copies etc. This output can be redirected to a file for later analysis.

5. Hints and tips

5.1 Standard and third-party names

Snob is too ignorant to understand issues like namespaces. If it sees a name which is not reserved, not a keyword and not seen in API files, Snob will replace this name with another name it invents. This simplicity is intended. The problem is in using names from a standard or third-party library. For instance, if you call the standard C function `time()`, you need to let Snob know that the name `time` is to be preserved. A side effect is that if you have a struct with an element called `time`, then Snob will preserve it as well. If you don't want this to happen, you need to follow a careful naming convention such that a name appears in one namespace only.

So, you decided to use the function `time()`, and you want to tell Snob that the name is not to be obfuscated. There are three ways to do so:

- declare it in a `keyword=` or `reserved=` or `ignore=` statement in the language-dependent configuration file `dotc.snob`;
- list `time` in a `reserved.snob`;
- declare the header `time.h` an API file

The first option is slower than the second one; it should be used only if you want to obfuscate the name `time` in languages other than C (for this example). The third option guards not only `time` but also all names seen in `time.h`, and, like the second option, for all files regardless of extension. It can be taken to an extreme by including all standard headers by writing a line to `APIfiles.snob` such as these:

```
C:\Program Files\Microsoft Visual Studio .NET 2003\Vc7\include\*  
C:\Program Files\Microsoft Visual Studio .NET 2003\Vc7\include\sys\*
```

This is a way to protect all standard names at once; Snob may take some time to comprehend all headers, but it is not necessarily a problem since you don't run Snob very often. The true problem is that you seldom would use all standard headers in your code, so you may inadvertently use names that are in use in standard headers, and those names will not be obfuscated, resulting in degraded quality of obfuscation.

Thus, which solution you want to accept depends on how many standard names you use in the project. If just a few, like in an embedded environment, `reserved.snob` may be just the right choice: you list the names known to be used and that's it. If, on the other hand, you do, say, Windows programming, you hardly know in advance which structures, classes and functions you will end up using. In this case, listing Windows headers as API files may be a better idea.

Some languages like C and C++ let you know where the standard (or third-party) names are: they are in the common headers. That's why you can list those headers in the `APIfiles.snob` file and not to worry about obfuscating a standard name.

If your programming language does not provide such a facility, you are limited to the first two options of the previous section.

5.2 Copying text files

If you need to have text files in your project directory subtree copied to the target directory subtree (such as makefiles, IDE project definition text files), you can simply list them in `APIfiles.snob`.

5.3 Dealing with mangled names

A language compiler may mangle names seen in the source code. This is not a problem by itself, but to access a language construct from a different language requires some labor and knowledge of how exactly the original name is mangled, the latter often being compiler-specific.

As a simple example, consider a C compiler that adds an underscore (`_`) in front of each name and a mixed C and Assembler project. If a construct seen by C and Assembler has a name in C, say, `theName`, then its Assembler name is `_theName`. The name definition for the Assembler files should then be something like “a word preceded by an underscore or a word not starting with an underscore.” This would ensure that the names naming the same construct in C and Assembler are still compatible. For instance, if `theName` obfuscates to `C01234567`, then C files will use `C01234567`, and Assembler files will use `_C01234567`. Not that it is readable, but something about the name is still revealed: namely, that it’s a name visible from C. You should be aware of this.

If a C compiler mangles names mostly just for kicks, a C++ compiler mangles names for a purpose. The purpose is of course *to reveal* type information (such as the number and the types of a function’s arguments) to the linker – and to a curious hacker. The C++ source code name obfuscation does not mitigate the exposure of type information in the C++ names. For a good reverse engineer, this information tells as much as an accurately chosen name of a construct. Short of tinkering with object files, there is nothing that can be done about it.

Probably, very few people would want to access C++ constructs from outside C++ realm. So constructing C++-aware name definitions for, say, C may not have any practical importance. In general though, mangling names in one language affects name definitions in another language if the languages access the same constructs. The C/Assembler example earlier demonstrates a way of dealing with this.

6. Snob error reporting

In case Snob encounters an error, it doesn’t try to recover since the result would be meaningless anyway.

Snob may report the error in a hierarchical way, starting from top and drilling down to a greater detail, e.g.,

```
Regular expression error in : .c
Regular expression error in : ../snobtest\dotc.snob
Regular expression error in : missing ) line 16 column 23
```

The hierarchical error reporting is especially useful when dealing with included files.

Snob recognizes the following error categories:

- Internal error
- Out of memory
- Error reading file
- Cannot create target directory
- Bad configuration file
- Error writing file
- Regular expression error
- Bad search filespec

- Include/use nestedness limit exceeded

Upon exit, Snob returns 0 on success or non-zero if an error was encountered.

Snob run is successful if the last two lines sent to the standard output are

```
End processing the project
Finished
```

7. Known issues

If a regular expression you specified is very complicated to match, Snob may quietly die of stack overflow. It is fairly easy to fix this behavior for the price of slower execution but it's not worth it because the problem regular expression is certainly not what you intended it to be.

The obfuscated project in the target directory may no longer build. Short of Snob failure (which you are asked to report to snob@macroexpressions.com), there are three potential sources of the problem, two of which are your fault and the third may be yours or of the toolchain you are using. Here they are:

- You provided a `.snob` extension-dependent configuration file for a binary type. Snob almost certainly will cripple a binary file to an unusable state. If this was an object file, you will get a complaint from the linker. If it is a voodoo of an Integrated Development Environment (IDE), you won't be able to even open your project. **Please, do not provide `dotobj` `.snob` and the like configuration files!**
- The configuration you provided is incorrect. Please, make sure that all your regular expressions do what you want them to do. Debug them on a tiny toy project where their correctness can be established by inspection. An additional issue arises when using standard or third-party header files (i.e., those which do not become a part of your project). Please, see the *Hints ant Tips* section for a relevant discussion.
- The build tool uses absolute path information (as opposed to relative to project directory). This is unfortunate: since the obfuscated project ends up in a different directory, it won't build indeed. To mitigate this problem, you have a few options: If you use your own makefiles to build the project, fix them to use project-relative path information. If you use an IDE, check if you can configure it to use relative paths. If you cannot, you may (in Windows environment) invent a drive, say, `R:` using the `append` command; configure the IDE to use the `R:` drive for the project and then switch the directory mapped to `R:` from the original directory to the obfuscated directory and back as needed. No, it's not elegant but if you create small batch files to do this, it's tolerable.