Bernstein Discussion Of S/W Obfuscation (= Anti-Reverse Engineering) To Protect Unexploded Ordinance

Author: *Avraham DOT Bernstein PLUS obfuscation AT gmail DOT com*
WWW: *https://www.avrahambernstein.com*
Last Update: *2025-08-18*

O/S Assumption

For purposes of this article we assume that the S/W is built using an embedded Linux system that supports Dynamic Share Object (DSO) modules. The alternative would be to use a static build of the O/S such as Musl which is outside the scope of this article (but we can handle static builds too).

Obfuscation Limitations

Obfuscation protection cannot "overly increase" the size of the application's RAM and FLASH footprint, nor can it "overly increase" the run-time speed of the application. For a rule of thumb, the footprint cannot be over 20% larger, nor can the speed reduction be over 20%. The software must have the ability to incorporate pragmas that will control, or possibly completely eliminate, obfuscations at selected points in the S/W.

Build System

We assume that the S/W is built using the "C" programming language that conforms to MISRA C restrictions.

We will supply rules, described in detail below, for automatically refactoring the "C" source code by utililizing a licensed commercial version of SrcML and Python Beautiful Soup, where:

1. *SrcML* converts the "C" source code to an XML-AST format.
2. Python and *Beautiful Soup* are used to refactor the XML-AST source code.
3. *SrcML* is used again to convert the refactored XML-AST back to "C" source code.

Primary Refactoring Techniques

1. We will refactor each *DSO* so that they will not formally export any user functions or data when the *DSO's* exports are loaded into the calling function's address space via the `dlopen` function! Normally the sole

purpose of designing a library is to access these exports! But knowledge of these exports are critical for the adveserial hacker who is attempting to reverse engineer the library. Therefore we have developed a hidden channel mechanism to access a "shrouded" export table when the caller invokes the `dlopen` function by using an `attribute(constructor)` function defined in the `.init` section of the DSO.

2. We will refactor the *DSO* so that all calls to `libc` functions are "shrouded". Run-time tracing of `libc` functions is an important reverse engineering technique that we must block.

3. The shrouding mechanism, aka "key wrapping", will double the size of the addresses we wish to protect. The mechanism will be light weight (`inline`), and implemented via registers - in order that there is no single function which the adversary can attack.

4. Shrouding can also be used to protect constants and strings.

5. Parameters to protected functions will also be shrouded. This will make it difficult for our adversary to do run-time debugging of the call stack because every function invocation will be different even when the same parameters are used

6. There will be a separate local variable and global variable shrouding mechanism - each relying upon its own separate light weight (`inline`) library specific *PRNG*.

7. Loops that are *order independent* can be randomized.

8. We will supply a *pragma* mechanism in order to allow the user to fine tune the usage of the above techniques.