NestX86 Tutorial
using bubblesort examples

This is a simple tutorial how to use White Hawk Software's NestX86 tool to protect a simple C program.

Requirements:
-The examples have been tested using Windows 7 and Microsoft Visual Studio 13. Other versions may also work.
-Of course the NestX86 binary is required as well.

Installation:
-Put the directories wherever you want.
-Make sure that NMAKE can access the NestX86 binary from the Makefile(s).

This can be done by editing the makefiles, or by setting up a `WHS_NESTX86_BINDIR` environment variable. This environment variable should point to the directory which contains both, the actual executable, nestx86.exe, as well as a directory with runtime support data, called "whs_protlib".

To run any of the examples

CD into the individual example directory.
Run
"nmake clean"
Now the number of files should be minimal; look at the files.
Run
"nmake"
to build the example.

A protected binary should now have been generated.
In these examples it will be called "prot.exe"
Run
"prot.exe"

Some virus checkers may dislike protected binaries (or even the protection tool).
If yor programs get deleted with or without warning, disable the virus checker and try again.

Bubblesort example 1

Simple obfuscation

This example should get readers familiar with that very simple example program and its protection.

It has a **simple protection script**. Lets have a look at its content

<log file = "@script@/protection.log"/>
This is optional, but I recommend always creating a log file, it will often be used.

<output file = "@script@\prot.obj "/>
Defines the filename of a protected object file. You might use the default, but these examples do not.

```
<inputimage
      name = "Main"
      file = "@script@/bubblesort.obj" >
   <init>_main</init>
   <no_decoywarning/>
   <no_unpacker/>
</inputimage>
```

describes the object file to be protected.

We will call the image (the internal representation of the object file) "Main". Get used to always name the image. Serious protections will involve multiple object files.

The path to the file is using a special form @script@ which simply means the subdirectory which contains the protection script.

An initialization point is named; the protector adds all necessary initializations to this function. See that the function name starts with an underscore? That is the name internally used by the build tools. You will later see how to print the symbols to find function names.

The two nodes <no_decoywarning/> and <no_unpacker/> just disable these features. For the beginning lets disable these to avoid unnecessary complications for the tutorial this way.
-Not having an unpacker simply lets users look at the real obfuscations beyond an initial encryption.
-An otherwise appearing decoy-warning would be annoying; for this tutorial's clarity we do not want to add decoy instructions.

```
<range  name = "R1" func = "_display" image = "Main"/>
<range  name = "R2" func = "_bubblesort" image = "Main"/>
```

This defines 2 ranges used by the protection. The "name" is a name given to this range in this protection script. "func" specifies that the range is the extent of the named function. See the underscore? It is the internal name given by the compiler. Finally they are in the object file (image) named "Main". Just get used to always give the object file; it won't be long and you will be using multiple object files.

```
<aspect
      name  = "obfuscating_aspect"
      type  = "Obfuscation"
      image = "Main"
      strength = "200">
   <range>
      <add ref="R1" />
      <add ref="R2" />
   </range>
</aspect>
```

This is where the actual protection is defined. This defines an "aspect" of the protection, (an Obfuscation [type] cleverly called "obfuscating_aspect"). "strength" gives the strength of the obfuscation. (Bigger strength = more obfuscations [per number of instructions])
Important is the "range", the code-range to be obfuscated. In this case the range is a set of two ranges R1 and R2 previously defined.

Ignore the "SystemAspect" for "final_printing". You will need printing very soon however.

There is no limit to how many aspects a protection may have.

That is the whole protection script. We recommend using cut and paste when you write protection scripts.

In this example, the script guides the whole protection; the protected program looks completely unaware of the protection.

Lastly, don't forget: Debug your program first; debugging it after it is protected will be a nightmare.

Bubblesort example 2

Multiple object files

In this example you learn how to protect a script containing more then one single object file. It is the very same program as in example 1, except that one function is removed from the source file and put into a source file of its own.

```
<workimage
        name = "Swapping"
        file = "@script@/swapping.obj" >
</workimage>
```

This defines the second object file, and we will call it "Swapping".

A different xml node type is used for this. Inclusion doesn't need all protection details; this is simply a pointer to another object file used in the protection.

The XML Node
<include image = "Swapping"/>

(in the inputimage "Main") will cause the actual linking of the "Swapping" object file into the "Main" object file.

Bubblesort example 3

Precise inter-function boundaries

Ok, here we get a little bit more real: Soon you will care about performance impact of a protection. This example uses markers to define ranges of code.

See the source code: It includes a header file which defines the marker functions. There are 4 places where a marker function "whs_marker" will be called. These markers define positions "a", "b", "c" and "d".

So what about an extra function call when we said performance matters? No need to worry: During the protection process the actual function-call instructions will be removed. No implementation for the markers needs to be linked with the application. *Patent pending.*

For debugging reasons, however there is a real implementation available. So if the program needs to be run unprotected, simply link the do-nothing implementation of the markers.

Lets look at the markers: These define positions in the source program.

```
<range  name = "R1" from = "a" to = "b"  image= "Main"/>
<range  name = "R2" from = "c" to = "d"  image= "Main"/>
```

Define two ranges delimited by the markers.

You may notice that the range "R2" is defined by markers which look unstructured. That is ok! The markers do not lead to code generated but to addresses in the generated code. There is no need for protection ranges to match source code ranges of compiled statements.

Bubblesort example 4

Checksum

This example will add a checksum to the bubblesort.

```
<aspect
        name  = "checksum_aspect"
        type  = "Checksum"
        image = "Main"
        stage = "protection">
    <range>
        <add func = "_display" />
        <add func = "_bubblesort" />
    </range>
    <call ref = "invoke" />
</aspect>
```

There is no surprise, we use an aspect of type "Checksum".

The "call" specifies the location where the correctness of the checked range is tested. This "call" node uses a "ref" which is a reference, in this case it refers to a position called "invoke".

```
<position
        name   = "invoke"
        marker = "check"
        image  = "Main">
</position>
```

which is designated in the program with a marker "check".

This may first feel like one too many indirections. But in real world protections the difference between markers and positions will become essential. Markers are defined in a file. Different files can have conflicting marker names. The position however is defined in the protection script. The protection script and the source files are very different name spaces.

Bubblesort example 5

Explicit programmed protection feature

Here we are getting fancy. In high-end protections the programmer can help the protection tool to find usefull facts to exploit for the correctness test.

In this case there is a line (line 14)

```
    whs_assertequal(1, (array[i] < array[j]));
```

in the bubble sort program. This call is like a marker: the protection tool will remove the call and insert an obfuscated test for the assertion instead. *Patent pending.*

(*Yes, there is also an un-obfuscated version available for debugging purposes.*)

This whs_assertequal function has a special form. It is legal c for testing, but during the protection process, the protection tool expects the first parameter to be a literal and the second parameter to be an expression.

The protection tool will add code to perform an aequivalence check, however the literal in the function will NEVER appear in the binary code in an unprotected way.

In this particular example, the expression to be evaluated returns a boolean value. That is not necessary, it could return any integer value.

Note the nice double-functionality: While executing in debugging mode this is a real function doing a traditional assertion. Only during execution of a protected version does this becomes a deeply hidden test.

We recommend to play with this example, looking for other literal values to test and maybe inspecting the generated assembly code.

Hence at least now you might use the "SystemAspect" for printing generated assembly code.

In many cases using a system aspect or a printing aspect is rather aequivalent. The examples use the "SystemAspect", which is implemented in a way that the protector can print during arbitrary protection stages, even before markers are evaluated and removed, or during other hidden stages.

Usefull protection stages for printing are:
markerExtraPass inputPrinting pre_obfuscation liftedPrinting finalPrinting