Decryption Filters

All decryption filters work on the principle of "security through obscurity." Regardless of how well you write a decryption filter and how strong your encryption algorithm, anyone determined enough can retrieve the original source code. The reason is quite simple - once the decryption filter has decrypted the source back to its original form, fragments of it will be stored in the computer's memory as Perl parses it. The source might only be in memory for a short period of time, but anyone possessing a debugger, skill, and lots of patience can eventually reconstruct your program.

That said, there are a number of steps that can be taken to make life difficult for the potential cracker. The most important: Write your decryption filter in C and statically link the decryption module into the Perl binary. For further tips to make life difficult for the potential cracker, see the file *decrypt.pm* in the source filters module.

64.6 CREATING A SOURCE FILTER AS A SEPARATE EXECUTABLE

An alternative to writing the filter in C is to create a separate executable in the language of your choice. The separate executable reads from standard input, does whatever processing is necessary, and writes the filtered data to standard output. `Filter:cpp` is an example of a source filter implemented as a separate executable - the executable is the C preprocessor bundled with your C compiler.

The source filter distribution includes two modules that simplify this task: `Filter::exec` and `Filter::sh`. Both allow you to run any external executable. Both use a coprocess to control the flow of data into and out of the external executable. (For details on coprocesses, see Stephens, W.R. "Advanced Programming in the UNIX Environment." Addison-Wesley, ISBN 0-210-56317-7, pages 441-445.) The difference between them is that `Filter::exec` spawns the external command directly, while `Filter::sh` spawns a shell to execute the external command. (Unix uses the Bourne shell; NT uses the cmd shell.) Spawning a shell allows you to make use of the shell metacharacters and redirection facilities.

Here is an example script that uses `Filter::sh`:

```
use Filter::sh 'tr XYZ PQR' ;
$a = 1 ;
print "XYZ a = $a\n" ;
```

The output you'll get when the script is executed:

```
PQR a = 1
```

Writing a source filter as a separate executable works fine, but a small performance penalty is incurred. For example, if you execute the small example above, a separate subprocess will be created to run the Unix `tr` command. Each use of the filter requires its own subprocess. If creating subprocesses is expensive on your system, you might want to consider one of the other options for creating source filters.

64.7 WRITING A SOURCE FILTER IN PERL

The easiest and most portable option available for creating your own source filter is to write it completely in Perl. To distinguish this from the previous two techniques, I'll call it a Perl source filter.

To help understand how to write a Perl source filter we need an example to study. Here is a complete source filter that performs rot13 decoding. (Rot13 is a very simple encryption scheme used in Usenet postings to hide the contents of offensive posts. It moves every letter forward thirteen places, so that A becomes N, B becomes O, and Z becomes M.)

```
package Rot13 ;

use Filter::Util::Call ;
```