NAME

Safe - Compile and execute code in restricted compartments

SYNOPSIS

```
use Safe;

$compartment = new Safe;

$compartment->permit(qw(time sort :browse));

$result = $compartment->reval($unsafe_code);
```

DESCRIPTION

The Safe extension module allows the creation of compartments in which perl code can be evaluated. Each compartment has

a new namespace

The "root" of the namespace (i.e. "main::") is changed to a different package and code evaluated in the compartment cannot refer to variables outside this namespace, even with run-time glob lookups and other tricks.

Code which is compiled outside the compartment can choose to place variables into (or *share* variables with) the compartment's namespace and only that data will be visible to code evaluated in the compartment.

By default, the only variables shared with compartments are the "underscore" variables $_ and @_ (and, technically, the less frequently used %_, the _ filehandle and so on). This is because otherwise perl operators which default to $_ will not work and neither will the assignment of arguments to @_ on subroutine entry.

an operator mask

Each compartment has an associated "operator mask". Recall that perl code is compiled into an internal format before execution. Evaluating perl code (e.g. via "eval" or "do 'file'") causes the code to be compiled into an internal format and then, provided there was no error in the compilation, executed. Code evaluated in a compartment compiles subject to the compartment's operator mask. Attempting to evaluate code in a compartment which contains a masked operator will cause the compilation to fail with an error. The code will not be executed.

The default operator mask for a newly created compartment is the ':default' optag.

It is important that you read the Opcode(3) module documentation for more information, especially for detailed definitions of opnames, optags and opsets.

Since it is only at the compilation stage that the operator mask applies, controlled access to potentially unsafe operations can be achieved by having a handle to a wrapper subroutine (written outside the compartment) placed into the compartment. For example,

```
    $cpt = new Safe;
    sub wrapper {
        # vet arguments and perform potentially unsafe
operations
    }
    $cpt->share('&wrapper');
```

WARNING

The authors make **no warranty**, implied or otherwise, about the suitability of this software for safety or security purposes.

The authors shall not in any case be liable for special, incidental, consequential, indirect or other similar damages arising from the use of this software.

Your mileage will vary. If in any doubt **do not use it**.

RECENT CHANGES

The interface to the Safe module has changed quite dramatically since version 1 (as supplied with Perl5.002). Study these pages carefully if you have code written to use Safe version 1 because you will need to makes changes.

Methods in class Safe

To create a new compartment, use

```
$cpt = new Safe;
```

Optional argument is (NAMESPACE), where NAMESPACE is the root namespace to use for the compartment (defaults to "Safe::Root0", incremented for each new compartment).

Note that version 1.00 of the Safe module supported a second optional parameter, MASK. That functionality has been withdrawn pending deeper consideration. Use the permit and deny methods described below.

The following methods can then be used on the compartment object returned by the above constructor. The object argument is implicit in each case.

permit (OP, ...)

Permit the listed operators to be used when compiling code in the compartment (in *addition* to any operators already permitted).

You can list opcodes by names, or use a tag name; see *"Predefined Opcode Tags" in Opcode*.

permit_only (OP, ...)

Permit *only* the listed operators to be used when compiling code in the compartment (*no* other operators are permitted).

deny (OP, ...)

Deny the listed operators from being used when compiling code in the compartment (other operators may still be permitted).

deny_only (OP, ...)

Deny *only* the listed operators from being used when compiling code in the compartment (*all* other operators will be permitted).

trap (OP, ...)

untrap (OP, ...)

The trap and untrap methods are synonyms for deny and permit respectfully.

share (NAME, ...)

This shares the variable(s) in the argument list with the compartment. This is almost identical to exporting variables using the *Exporter* module.

Each NAME must be the **name** of a non-lexical variable, typically with the leading type identifier included. A bareword is treated as a function name.

Examples of legal names are '$foo' for a scalar, '@foo' for an array, '%foo' for a hash, '&foo' or 'foo' for a subroutine and '*foo' for a glob (i.e. all symbol table entries associated with "foo", including scalar, array, hash, sub and filehandle).

Each NAME is assumed to be in the calling package. See share_from for an alternative method (which share uses).

share_from (PACKAGE, ARRAYREF)

This method is similar to share() but allows you to explicitly name the package that symbols should be shared from. The symbol names (including type characters) are supplied as an array reference.

```
    $safe->share_from('main', [ '$foo', '%bar', 'func' ]);
```

varglob (VARNAME)

This returns a glob reference for the symbol table entry of VARNAME in the package of the compartment. VARNAME must be the **name** of a variable without any leading type marker. For example,

```
    $cpt = new Safe 'Root';
    $Root::foo = "Hello world";
    # Equivalent version which doesn't need to know $cpt's
package name:
    ${$cpt->varglob('foo')} = "Hello world";
```

reval (STRING)

This evaluates STRING as perl code inside the compartment.

The code can only see the compartment's namespace (as returned by the **root** method). The compartment's root package appears to be the `main::` package to the code inside the compartment.

Any attempt by the code in STRING to use an operator which is not permitted by the compartment will cause an error (at run-time of the main program but at compile-time for the code in STRING). The error is of the form "'%s' trapped by operation mask...".

If an operation is trapped in this way, then the code in STRING will not be executed. If such a trapped operation occurs or any other compile-time or return error, then $@ is set to the error message, just as with an eval().

If there is no error, then the method returns the value of the last expression evaluated, or a return statement may be used, just as with subroutines and **eval()**. The context (list or scalar) is determined by the caller as usual.

This behaviour differs from the beta distribution of the Safe extension where earlier versions of perl made it hard to mimic the return behaviour of the eval() command and the context was always scalar.

Some points to note:

If the entereval op is permitted then the code can use eval "..." to 'hide' code which might use denied ops. This is not a major problem since when the code tries to execute the eval it will fail because the opmask is still in effect. However this technique would allow clever, and possibly harmful, code to 'probe' the boundaries of what is possible.

Any string eval which is executed by code executing in a compartment, or by code called from code executing in a compartment, will be eval'd in the namespace of the compartment. This is potentially a serious problem.

Consider a function foo() in package pkg compiled outside a compartment but shared with it. Assume the compartment has a root package called 'Root'. If foo() contains an eval statement like eval '$foo = 1' then, normally, $pkg::foo will be set to 1. If foo() is

called from the compartment (by whatever means) then instead of setting $pkg::foo, the eval will actually set $Root::pkg::foo.

This can easily be demonstrated by using a module, such as the Socket module, which uses eval "..." as part of an AUTOLOAD function. You can 'use' the module outside the compartment and share an (autoloaded) function with the compartment. If an autoload is triggered by code in the compartment, or by any code anywhere that is called by any means from the compartment, then the eval in the Socket module's AUTOLOAD function happens in the namespace of the compartment. Any variables created or used by the eval'd code are now under the control of the code in the compartment.

A similar effect applies to *all* runtime symbol lookups in code called from a compartment but not compiled within it.

rdo (FILENAME)

This evaluates the contents of file FILENAME inside the compartment. See above documentation on the **reval** method for further details.

root (NAMESPACE)

This method returns the name of the package that is the root of the compartment's namespace.

Note that this behaviour differs from version 1.00 of the Safe module where the root module could be used to change the namespace. That functionality has been withdrawn pending deeper consideration.

mask (MASK)

This is a get-or-set method for the compartment's operator mask.

With no MASK argument present, it returns the current operator mask of the compartment.

With the MASK argument present, it sets the operator mask for the compartment (equivalent to calling the deny_only method).

Some Safety Issues

This section is currently just an outline of some of the things code in a compartment might do (intentionally or unintentionally) which can have an effect outside the compartment.

Memory

Consuming all (or nearly all) available memory.

CPU

Causing infinite loops etc.

Snooping

Copying private information out of your system. Even something as simple as your user name is of value to others. Much useful information could be gleaned from your environment variables for example.

Signals

Causing signals (especially SIGFPE and SIGALARM) to affect your process.

Setting up a signal handler will need to be carefully considered and controlled. What mask is in effect when a signal handler gets called? If a user can get an imported function to get an exception and call the user's signal handler, does that user's restricted mask get re-instated before the handler is called? Does an imported handler get called with its original mask or the user's one?

State Changes

Ops such as chdir obviously effect the process as a whole and not just the code in the compartment. Ops such as rand and srand have a similar but more subtle effect.

AUTHOR

Originally designed and implemented by Malcolm Beattie, mbeattie@sable.ox.ac.uk.

Reworked to use the Opcode module and other changes added by Tim Bunce <*Tim.Bunce@ig.co.uk*>.