Chapter 26

# Locking Ruby in the Safe

Walter Webcoder has a great idea for a portal site: the Web Arithmetic Page. Surrounded by all sorts of cool mathematical links and banner ads that will make him rich is a simple web form containing a text field and a button. Users type an arithmetic expression into the field, click the button, and the answer is displayed. All the world's calculators become obsolete overnight; Walter cashes in and retires to devote his life to his collection of car license plate numbers.

Implementing the calculator is easy, thinks Walter. He accesses the contents of the form field using Ruby's CGI library and uses the eval method to evaluate the string as an expression:

Download **samples/taint_1.rb**

```
require 'cgi'
cgi = CGI.new("html4")
# Fetch the value of the form field "expression"
expr = cgi["expression"].to_s
begin
  result = eval(expr)
rescue Exception => detail
  # handle bad expressions
end
# display result back to user...
```

Roughly seven seconds after Walter puts the application online, a twelve-year-old from Waxahachie with glandular problems and no real life types system("rm *") into the form, and like his computer's files, Walter's dreams come tumbling down.

Walter learned an important lesson: *All external data is dangerous. Don't let it close to interfaces that can modify your system.* In this case, the content of the form field was the external data, and the call to eval was the security breach.

Fortunately, Ruby provides support for reducing this risk. All information from the outside world can be marked as *tainted*. When running in a safe mode, potentially dangerous methods will raise a SecurityError if passed a tainted object.

# Safe Levels

The variable $SAFE determines Ruby's level of paranoia. Table 26.1 on page 440 gives more details of the checks performed at each safe level.

| $SAFE | Constraints |
|---|---|
| 0 | No checking of the use of externally supplied (tainted) data is performed. This is Ruby's default mode. |
| ≥ 1 | Ruby disallows the use of tainted data by potentially dangerous operations. |
| ≥ 2 | Ruby prohibits the loading of program files from globally writable locations. |
| ≥ 3 | All newly created objects are considered tainted and untrusted. |
| ≥ 4 | Ruby effectively partitions the running program in two. Nontrusted objects may not be modified. |

The default value of $SAFE is zero under most circumstances. However, if a Ruby script is run *setuid* or *setgid*[1] or if it run under mod_ruby, its safe level is automatically set to 1. The safe level may also be set by using the -T command-line option and by assigning to $SAFE within the program. It is not possible to lower the value of $SAFE by assignment.

The current value of $SAFE is inherited when new threads are created. However, within each thread, the value of $SAFE may be changed without affecting the value in other threads. This facility may be used to implement secure "sandboxes," areas where external code may run safely without risk to the rest of your application or system. Do this by wrapping code that you load from a file in its own, anonymous module. This will protect your program's namespace from any unintended alteration.

Download **samples/taint_2.rb**

```
File.open(filename,"w") do |f|
  f.print ...   # write untrusted program into file.
end
Thread.start do
  $SAFE = 4
  load(filename, true)
end
```

With a $SAFE level of 4, you can load *only* wrapped files. See the description of Kernel.load on page 571 for details.

This concept is used by Clemens Wyss on Ruby CHannel (http://www.ruby.ch). On this site, you can run the code from the first edition of this book. You can also type Ruby code into a window and execute it. And yet he doesn't lose sleep at night, because his site runs your code in a sandbox.

1. A Unix script may be flagged to be run under a different user or group ID than the person running it. This allows the script to have privileges that the user does not have; the script can access resources that the user would otherwise be prohibited from using. These scripts are called *setuid* or *setgid*.

You can find a listing of the Ruby source code for this sandbox on the Web at http://www.approximity.com/cgi-bin/rubybuch_wiki/wpage.rb?nd=214.

The safe level in effect when a Proc object is created is stored with that object. The safe level may be set during the execution of a proc object without affecting the safe level of the code that invoked tha proc. A proc may not be passed to a method if it is tainted and the current safe level is greater than that in effect when the block was created.

# Tainted Objects

Any Ruby object derived from some external source (for example, a string read from a file or an environment variable) is automatically marked as being tainted. If your program uses a tainted object to derive a new object, then that new object will also be tainted, as shown in the following code. Any object with external data somewhere in its past will be tainted. This tainting process is performed regardless of the current safe level. You can see whether an object is tainted using Object#tainted?.

```
# internal data                          # external data
# =============                          # =============

x1 = "a string"                          y1 = ENV["HOME"]
x1.tainted?       # =>   false           y1.tainted?       # =>   true

x2 = x1[2, 4]                            y2 = y1[2, 4]
x2.tainted?       # =>   false           y2.tainted?       # =>   true

x1 =~ /([a-z])/   # =>   0               y1 =~ /([a-z])/   # =>   2
$1.tainted?       # =>   false           $1.tainted?       # =>   true
```

You can force any object to become tainted by invoking its taint method. If the safe level is less than 3, you can remove the taint from an object by invoking untaint.[2] This is not something to do lightly.

# Trusted Objects

1.9

Ruby 1.9 adds *trust*, a new dimension to the concept of safety. All objects are marked as being *trusted* or *untrusted*. In addition, running code can be trusted or not. And, when you're running untrusted code, objects that you create are untrusted, and the only objects that you can modify are those that are marked untrusted. What this in effect means is that you can create a sandbox to execute untrusted code, and code in that sandbox cannot affect objects outside that sandbox.

2. You can also use some devious tricks to do this without using untaint. We'll leave it up to your darker side to find them.

Let's get more specific. Objects created while Ruby's safe level is less than 3 are trusted. However, objects created while the safe level is 3 or 4 will be untrusted. Code running at safe levels 3 and 4 is also considered to be untrusted. Because untrusted code can modify only untrusted objects, code at safe levels 3 and 4 will not be able to modify objects created at a lower safe level.

Download **samples/taint_5.rb**

```
dog = "dog is trusted"
cat = lambda { $SAFE = 3; "cat is untrusted" }.call
puts "dog.untrusted? =  #{dog.untrusted?}"
puts "cat.untrusted? =  #{cat.untrusted?}"
# running at safe level 1, these operations will succeed
puts dog.upcase!
puts cat.upcase!

# running at safe level 4, we can modify the cat
lambda { $SAFE = 4; cat.downcase! }.call
puts "cat is now '#{cat}'"
# but we can't modify the dog
lambda { $SAFE = 4; dog.downcase! }.call
puts "so we never get here"
```

*produces:*

```
dog.untrusted? =  false
cat.untrusted? =  true
DOG IS TRUSTED
CAT IS UNTRUSTED
cat is now 'cat is untrusted'
prog.rb:17:in `downcase!': Insecure: can't modify string (SecurityError)
from /tmp/prog.rb:17:in `block in <main>'
from /tmp/prog.rb:17:in `call'
from /tmp/prog.rb:17:in `<main>'
```

You can set and unset the trusted status of an object using Object#untrust and Object#trust (but you have to be at below safe level 4 to call untrust and below safe level 3 to call trust). The method Object#untrusted? returns true if an object is untrusted.

Table 26.1. Definition of the Safe Levels

**$SAFE >= 1**

- The environment variables RUBYLIB and RUBYOPT are not processed, and the current directory is not added to the path.
- The command-line options -e, -i, -l, -r, -s, -S, and -x are not allowed.
- Can't start processes from $PATH if any directory in it is world-writable.
- Can't manipulate or chroot to a directory whose name is a tainted string.
- Can't glob tainted strings.
- Can't eval tainted strings.

1.9

- Can't load or require a file whose name is a tainted string (unless the load is wrapped).
- Can't manipulate or query the status of a file or pipe whose name is a tainted string.
- Can't execute a system command or exec a program from a tainted string.
- Can't pass trap a tainted string.

**$SAFE >= 2**

- Can't change, make, or remove directories, or use chroot.
- Can't load a file from a world-writable directory.
- Can't load a file from a tainted filename starting with ~.
- Can't use File#chmod, File#chown, File#lstat, File.stat, File#truncate, File.umask, File#flock, IO#ioctl, IO#stat, Kernel#fork, Kernel#syscall, Kernel#trap. Process.setpgid, Process.setsid, Process.setpriority, or Process.egid=.
- Can't handle signals using trap.

**$SAFE >= 3**

- All objects are tainted when they are created.
- Can't untaint objects.
- Can't add trust to an object.
- Objects are created untrusted.

**$SAFE >= 4**

- Can't modify a nontainted array, hash, or string.
- Can't modify a global variable.
- Can't access instance variables of nontainted objects.
- Can't change an environment variable.
- Can't close or reopen nontainted files.
- Can't freeze nontainted objects.
- Can't change visibility of methods (private/public/protected).
- Can't make an alias in a nontainted class or module.
- Can't get metainformation (such as method or variable lists).
- Can't define, redefine, remove, or undef a method in a nontainted class or module.
- Can't modify Object.
- Can't remove instance variables or constants from nontainted objects.
- Can't manipulate threads, terminate a thread other than the current thread, or set abort_on_exception.
- Can't have thread local variables.
- Can't raise an exception in a thread with a lower $SAFE value.
- Can't move threads between ThreadGroups.
- Can't invoke exit, exit!, or abort.
- Can load only wrapped files and can't include modules in untainted classes and modules.
- Can't convert symbol identifiers to object references.
- Can't write to files or pipes.
- Can't use autoload.
- Can't taint objects.
- Can't untrust an object.