Safer Scripting Through Precompilation

Ben Laurie
(ben@algroup.co.uk)

January 3, 2007

1 Introduction

One of the challenges in modern systems is the conflict between the desire to run software from a wide variety of untrusted sources and the need to prevent malicious activity by those scripts.

The current standard practice is to attempt to achieve this through permissions, but this has been shown repeatedly to fail in a variety of ways. If permissions are made too granular, they become impossible to configure and so tend to become useless. If they are less granular, loopholes appear through which malicious scripts can wriggle. In either case, providing useful defaults whilst still providing security has proved to be a daunting (or, perhaps, judging on the evidence, impossible) task.

Capabilities (in the object-capabilities sense[1]) allow authorisation through designation, a paradigm that permits fine-grained security without the need for tedious configuration. For historical reasons they have largely been ignored for the last few decades, meaning that there are few platforms that support them, none in wide use[1].

It has long seemed to me that the best way to introduce capabilities into existing environments is through modified versions of widely used programming languages, for two reasons – firstly, programmers are already familiar with the language, so the learning curve is shallow, and secondly because it is then possible to leverage the large body of existing code, through the process known as "taming".

I'd made several attempts at this, with little success – until I realised that the thing to do was to compile a modified version of the language into the unmodified base language, and introduce capabilities that way.

This paper outlines an experimental implementation of this idea for Perl, called CaPerl. It assumes familiarity with Perl.

2 Overview

There are many different ways to express capabilities, but perhaps the simplest and most easily understood is to implement them as standard objects in an object-oriented language. Of course, they are not capabilities unless some of the standard behaviour of O-O languages is eliminated. It should not be possible to "look inside" an object. There must be a distinction between public and private methods. Global variables should be avoided.

It is also necessary to create the capabilities in the first place, either externally to the system or through the use of "trusted" code.

Perl sounds like a fantastically unsuitable language to use as a basis for this, since there are so many ways to bend the rules in Perl. Interestingly, though, the use of a precompiler can quite easily enforce these rules precisely because Perl *is* so flexible.

[1] Although the AS/400 is allegedly a capability system, this is not exposed to applications, sadly.

The approach is to use Perl's introspection in the compiled code to ensure that each package can only access its own data, to add restrictions to what can be written by only accepting a subset of Perl, and to add slight extensions to the language to enable public and private methods to be created, and to differentiate between trusted and untrusted code.

The result is surprisingly useable, even with only a subset of the compiler implemented.

3 Objects

Perl isn't *really* object oriented, instead it uses the concept of blessing to link objects to code (via their package). This makes restricting access to objects quite easy. Every time CaPerl code attempts to dereference a blessed object, using the `->` operator, the code emitted first checks the dereferenced object using the built-in `ref` function to see whether it "belongs" to the current package. If not, then the code `croak`s.

4 Public and Private Methods

In fact, because there is also the need to distinguish between trusted and untrusted code, CaPerl introduces the concept of a "trusted" method, as well as public and private ones. These are flagged with a keyword in the `sub` declaration, for example:

```
trusted sub some_sub { ... }
```

These are checked with the built-in `caller` function. A private method can only be called from the package it is defined in, a trusted method can only be called from trusted code and a public method can, of course, be called from anywhere.

Note that because the called routine checks the caller, CaPerl makes no restrictions on calling methods on objects in untrusted code. In fact, such calls are the only thing of interest untrusted code can do with an object not its own.

5 Trusted and Untrusted Code

Trusted code is needed both to create the environment for untrusted code and to "tame" existing plain Perl code. The two are distinguished by a compile-time flag. Trusted code is allowed to do things untrusted code cannot: it can load modules written in Perl rather than CaPerl, it can call trusted methods in CaPerl code and it can use certain dangerous built-in Perl functions.

When compiled CaPerl code wants to check whether its caller is trusted or untrusted it does so simply by checking for a global variable in the package, which is introduced by the compiler.

6 Global Variables

Untrusted code should not be allowed access to globals. This is accomplished by the trivial expedient of prohibiting the use of `::` in variable names!

7 Built-in Functions

Perl has a large number of dangerous built-in functions, so untrusted code is only permitted to run a subset. This is achieved by prohibiting function calls (as opposed to calling methods on objects), and then permitting a few selected functions, for example `shift`.

"Dangerous" from a capabilities perspective does include functions that would normally be considered quite harmless, such as `print` or `fileno`, so these can only be called by trusted code.

8 Taming

Existing Perl code is not, of course, written with capability discipline in mind, so it cannot be exposed directly to untrusted CaPerl code. The process of "capability-ising" such code is known in some circles as taming. What this generally consists of is writing a very thin wrapper around the desired module in CaPerl. This is perhaps best illustrated by an example, a partial wrapper for `Term::ReadLine`.

```
package Wrap::Term::ReadLine;

use untrusted Term::ReadLine;

trusted sub new {
    my $class=shift;

    my $self={};
    bless $self,$class;

    $self->{readline}=new Term::ReadLine();

    return $self;
}

public sub readline {
    my $self=shift;
    my $prompt=shift;

    croak if ref($prompt) ne '';
    return $self->{readline}->readline($prompt);
}
```

```
1;
```

Taming can also be used to produce safer versions of built-in functions, such as `opendir` or `read`.

9 Departures From Perl

Of course, the extra keywords used are changes from Perl, but there were also syntax restrictions introduced, partly to reduce the complexity of the compiler[2] and partly to reduce the risk that cleverly written code would somehow circumvent CaPerl's security.

One of the major changes was to remove the ability to call functions without parentheses. This turns out to be surprisingly hard to support, particularly since, for example, `f g(a),b` is ambiguous: it could mean `f(g(a),b)` or `f(g((a),b))` . It may be that a future version will reintroduce this, since it was surprising how often experienced Perl authors use this facility.

Currently, string interpolation is also omitted from the language. This doesn't restrict what can be written, but does make some things rather more verbose than in standard Perl. It isn't particlarly hard to get this back, by parsing interpolated strings into a series of concatenations of the contents (in fact, this is what standard Perl does anyway).

Because Perl packages can introduce global functions, and access to built-ins must be restricted, the ability to use functions which are not object methods has been completely removed for untrusted code. This is not really a great handicap – at worst, the occasional function will have an unused extra parameter.

It is not currently clear how to support inheritance, so CaPerl currently doesn't.

10 An Example of CaPerl's Use

There are clearly many environments in which a restricted language such as CaPerl could be used. The "real world" I have been using in testing is CGI scripts. The idea is that a webserver could be run which permits arbitrary users to upload CaPerl scripts, which are then run in a restricted environment.

The version I have been running passes these scripts just two capabilities. One is a capability which can be used to write HTML back to the HTTP client[3] and the other gives each script access to a subdirectory of its own – it can list the directory (that is, get the functionality normally provided by `opendir` and `readdir`), and open (for read or write) or create any file in that directory. It cannot, however, move outside its own directory – the code implementing the capability prevents that.

[2] Perl's lexer, parser and compiler are rather horribly intertwined in order to permit some of Perl's shorthands and grammatical ambiguities.

[3] Although I currently use a version of this which gives the script a very free hand to write what it wants on the returned page, it would be quite easy to make a version that only permitted a subset of HTML to be used, for example.

Why is this useful? Even this trivial example has at least one quite profound use – imagine two mutually untrusting parties want to enter into a contract – they want to exchange two binary objects. But since they do not trust each other, each cannot be sure that the other will release his object once his own has been released. The usual answer is to use a trusted intermediary, but now we have an automated solution. The two write a simple script that takes the two objects and stores them, but won't release them until they are both present. They upload this to a trusted machine which runs it in the environment described above. They can then execute their transaction without any further help, and the owner of the trusted machine can be sure that their script can do nothing bad, even though he has no control over its content at all.

11 Future Directions

Support Perl more completely.

Figure out how to manage inheritance.

Apply the approach to other scripting languages, such as Python.

References

[1] Mark Miller, Ka-Ping Yee, and Jonathan Shapiro. Capability myths demolished. `http://zesty.ca/capmyths/`, 2003.