Schwern

Please don't hammer nails with a gun.

How (not) To Load a Module or Bad Interfaces Make Good People Do Bad Things

By **Michael G Schwern** on October 2, 2011 6:44 PM

tl;dr version: The design of `require` makes it all but impossible to use in a secure and correct fashion. To fix this, Perl needs two new ops: one which will *only* load *files*, and one which will *only* load *modules*. Both would *only* load from `@INC`.

The more I look into the problem, the more I'm convinced that there is no good way to load a module from a variable in Perl. None of the existing techniques or modules fully solve the problem. They all have security holes or limitations. This is kind of embarrassing, it's an easy thing and it should be easy. My investigation into how many ways the simple act of loading a module can go wrong has lead me to believe that the solution is a new op which just loads modules.

For those of you wondering, near as I can tell this is how you correctly and securely load a module from a variable...

```
sub require_module {
    my $module = shift;

    # Is it defined?
    die unless defined $module;

    # Is the caller using utf8?
    require utf8;
    my $with_utf8 = (caller(0))[8] & $utf8::hint_bits;

    # Are Unicode package names ok?
    my $check = $with_utf8 ? qr{\A [[:alpha:]_] [[:word:]]*    (?:
:: [[:word:]]+ )* \z}x
                           : qr{\A [A-Z_a-z]    [0-9A-Z_a-z]*  (?:
:: [0-9A-Z_a-z]+  )* \z}x;

    # Is it a syntactically valid module name?
    die unless $module =~ $check;

    # Transform to a pm file path
    my $file = $module;
    $file .= ".pm";
    $file =~ s{::}{/}g;

    # What were we doing again?
    return require $file;
}
```

Isn't that EASY?! Nothing I've looked at gets this all correct. And please don't cut and paste that into your code, fix one of the module loading modules instead.

This all started when I mentioned on StackOverflow that having your compiler and your exception handler share the same function is a security hole by way of a bad interface. It encourages people to use them interchangeably without really thinking about the consequences. The common example I came up with was `eval "require $module"` which happens a lot.

The real fun came when I tried to fix it.

See, I found a security hole in a major module (details withheld because I'd rather not point it out too conspicuously before it gets updated) which allows arbitrary code execution. Whoops.

About Michael G Schwern

Ya know, the guy who tells you to write tests and not use MakeMaker.

More info »

Search this blog

Submit

Yes, it did filter $module... or it tried to anyway. And yes, it was written by a very competent programmer. Loading a module from a variable is hard. It should be easy. Easy things should be easy.

My first solution was to replace it with `eval { require $module }`. Yeah, it's wrong. I was in a rush. My superhuman strength was needed to lift boxes, some of which were solidly packed with metal bars. It may have been a bank heist, the details are hazy. Funny enough, the module's tests did not catch the mistake (they do now).

If you know why it's wrong, skip ahead to the next `tl;dr`. The rest of you sit down and learn where this whole problem comes from.

`require` is really two functions and that's where the whole problem comes from. Are you seeing a theme? If you pass require a variable, like `require $module`, it will consider that to be a file path and try to load it, appending it to each entry in @INC assuming the path is not absolute (that becomes a concern later). If you pass `require` a bareword, like `require Module::Name`, it will consider that to be a module, validate it (probably the Perl compiler validates it), transform it into a .pm file path and then do what `require $scalar` does.

Got that? Confused? It's hard to keep straight, and that's half the problem. Let's make it clear(er):

```
require Bareword::Name
```

- `Bareword::Name` treated as a module name.
- Checked it is valid module name syntax.
- Transformed into a relative .pm file (ex. `Bareword/Name.pm`)
- Searched through `@INC`.

```
require $file
```

- $file treated as a file path.
- No checks are done.
- Absolute paths and paths starting with "./" are simply loaded.
- Other paths search through `@INC`.

`require Bareword::Name` is safe in that it will load a .pm in your `@INC` *and nothing else*. If the content of @INC can be trusted, you're good. (If it can't, you shouldn't be wasting time reading overly verbose security blog posts. Fix that shit now.)

`require $file` is not safe. It does no validation. It can be used to load *any valid Perl on the filesystem*. Got that? If an attacker injects a file into `/tmp` an unprotected `require $file` might let them execute it with elevated privileges. This is bad. It gets worse.

Trouble is, `require Bareword` lets you load from a bareword and not a variable. This is how Perl knows you're asking for a module and not a file. Rather than have two clearly defined functions we have clever syntax. Yaaay... guh.

The user is forced into two ways around this inflexibility, both unpalatable. Either you write `eval "require $module"` to trick Perl into thinking $module is a bareword, or you take it upon yourself to transform $module into a pm file path and use `require $file`. Both are security holes.

Welcome back `tl;dr` readers! Wipe the drool off the side of your face, we're not done.

Some of you are thinking, "I'm not stupid. I can write `eval "require $module"`! All I have to do is validate $module or make sure it comes from a trusted source!" Maybe.

Once upon a time my friend was showing off this new Glock. He pointed out the lack of a conventional safety, instead there's a number of very clever things to ensure the gun will not fire unless the trigger is deliberately pulled. On the other hand, a loaded Glock will *always* fire if the trigger is deliberately pulled. This is a great feature if you are, say, a trained police officer. If you're a civilian, this is a great way to shoot you or your house guest in the foot. My friend was so confident in the safety of his gun he bragged he could hammer nails with it.

Just because you can hammer nails with a gun doesn't mean you should build a house with it. Just because you think you can make `eval "require $module"` safe doesn't mean it is. Or that everyone else will. The best security is the security you don't have to be careful about. You always have to be careful with `eval STRING`, every single time.

You will get lazy and not validate. Or a clever attacker will figure a way around your validation. Or your validation will reject valid module names. Or code which previously only took safe $module is later changed to take unsafe user input. I've seen all of this. This is not the path to security.

So what about `require $file`? Because it's so easy to jump the `@INC` rails that starts out insecure by design. Here's how...

Let's say an attacker has found a way to get files onto your filesystem, doesn't matter what owner. This is already a security hole, but they don't have any way to execute the file. A cracker will now find a way to escalate their privilege stacking one minor security hole onto another to make a great big one. An unprotected `require $file` is all they need to execute it, because it will load any absolute path. Already `require` is a security risk because you have to jump through more hoops to make it stick to `@INC`.

Filter for absolute paths and you're good? No, also paths starting with `./`. And `../` because those will jump out of `@INC` too. Don't forget to check for `.\` and `.\` on Windows! Writing that down? Good. Save space, there's more.

This is about loading a module, not any old file. Usually people will first transform the module into a file path. That doesn't seem so hard, right?

```
my $file = $module;
$file .= ".pm";
$file =~ s{::}{/}g;
require $file;
```

Done? No. It's not even totally valid, it doesn't handle the `'` namespace separator, but only Klingons and hippies care about that. No, it's worse. It's a security hole and one that is all over our Perl code.

Remember how `require $file` can be made to jump the rails and load any file? You'd think the module transformation would prevent that, but no. Consider the "module" `/tmp/LOL/PWND` which the code above happily loads a `/tmp/LOL/PWND.pm`. Ok, maybe your code has some validation and only lets things that look like module names in. If the validation isn't well written it might allow `::tmp::LOL::PWND`.

The combination of a salad of insecurities makes loading a module almost impossible to properly secure unless you are very knowledgeable and very careful.

That's fine, in situations like this the smart and safe thing is to use a module! Then lots of people who think about this stuff can get it right. Right? Surely a module dedicated to the one act of loading a module has thought this through? Right? Please?

Nope.

perl5i gets it wrong. `$module->require` does not validate `$module` and so is vulnerable to the `::tmp::LOL::PWND` trick.

Module::Load will jump out of `@INC` just like require does.

Class::Load comes close. I haven't been able to make it do anything insecure, but it tells me things which are not valid module names, like "123", are and things which are valid module names, like "Mégå::Mödulé" are not valid. Also it has a rather long dependency chain just to safely load a module.

UNIVERSAL::require ~~works great~~ is vulnerable to shenanigans. By making require a class method call, it uses Perl's own idea of what a valid package name is. Unfortunately, what the Perl parser considers a package name and what the Perl runtime considers a package name is different. The runtime will accept all sorts of junk for the purposes of a class method call, like `subdir/../../../../../tmp/LOL/PWND`. For fuck's sake.

Even if a module is made convenient, correct and secure, people will still avoid adding a CPAN dependency for something as "trivial" as loading a module. The CPAN modules should be fixed to be sure, but they are not the long term solution.

The design of **`require`** makes it almost impossible to secure. Those modules went past a lot of eyeballs, including mine, and they still don't work right. While it is good to patch those modules, what is needed is a better **`require`** built right into Perl. More precisely, two.

The first, let's call it `require_module`, loads modules. It *only* loads modules in `@INC` and does nothing else. By doing this inside Perl it can perfectly validate the module name avoiding both validation mistakes and security holes.

The second, let's call it `require_file`, loads files. It *only* loads files from inside `@INC`. Absolute paths and those which try to updir are rejected. This allows a programmer to be sure what Perl code can be loaded is at least coming from secured locations.

`require_module` is easy, it can be carved out of the existing require code without much trouble (uhh... as far as writing new keywords in Perl goes). `require_file` is made difficult because Perl lacks the built in file path operations. Detecting an absolute path is pretty straightforward, but detecting one that's trying to updir is not. That requires a complete file path parsing library... in C.

Fortunately, only `require_module` is needed to solve our problem. With an in core way to securely and correctly load a module from a variable, the impulse to use any other hack fades away with older versions of Perl. A CPAN module can encapsulate the decision to use the built-in or a pure Perl work around depending on the version of Perl.

I'm writing up the tests and docs now. I'll be doing the work in a branch on Github called feature/split_require if you'd like to contribute. It especially needs someone who can write the actual C code using something better than my rusty chainsaw.

21 comments

Tagged as:

require, security

21 Comments

doy.tozt.net | October 3, 2011 1:03 AM | Reply

For what it's worth, "Mégå::Mödulé" can't really be considered a valid module name, because handling of unicode in package names is entirely dependent on the underlying file system at the moment (because as you said, all perl does is convert :: to / and tack .pm on at the end). Consider:

```
use utf8;

BEGIN {
    package Simple::Module;
    sub import { warn $_[0] }
    $INC{'Simple/Module.pm'} = 1;
}

use Simple::Module;

BEGIN {
    package Mégå::Mödulé;
    sub import { warn $_[0] }
    $INC{'Mégå/Mödulé.pm'} = 1;
}

use Mégå::Mödulé;
__END__
Simple::Module at blah.pl line 5.
Can't locate Mégå/Mödulé.pm in @INC (@INC contains: /home/doy/perl5
BEGIN failed--compilation aborted at blah.pl line 17.
```

This is why Class::Load behaves in this way (it is intentional). Accepting 123 as a valid module name is a bug, reporting it would be helpful.

Pedro Melo | October 3, 2011 6:57 AM | Reply

Why not `require file $file` and `require module $module`?

I prefer to overload something we have than pollute the standard namespace with new functions, given that someone else might already have a function with those names.

Michael G Schwern replied to comment from doy.tozt.net | October 3, 2011 2:30 PM | Reply

Your example works just fine for me here on OS X with 5.8.9, 5.10.1, 5.12.3 and 5.14.1. It also works on OS X writing a real Mégå/Mödulé.pm. I'm not sure why it's failing for you, your code should never touch the filesystem. The comparison with %INC must be failing or your editor interfering maybe? Bizarre.

That some systems don't support Unicode well doesn't mean everyone else has to be held back. More and more will as time goes on, best to get with it.

The logic of "it doesn't work yet at layer X so we're not going to fix it in layer Y" is a particularly vicious variant of the lava flow anti-pattern and a great barrier to progress. Every layer can use it as an excuse to not do any work until the other layers get fixed (see also IPv6). The more layers which don't work, the greater the excuse to not fix any one layer, and the greater chance another layer will be added. The only way out is to start fixing the layer you have control over.

Michael G Schwern replied to comment from Pedro Melo | October 3, 2011 2:50 PM | Reply

> Why not require file $file and require module $module?

While I agree that adding more functions to the namespace is not the best way to do it, there are far better ways, Perl 5 only gives us two hammers and it would be a bit silly to put the functions to load modules in a module.

The problems with overloading `require` far outweigh the problems of adding new keywords. First, and most important, it's far more difficult to document, teach, code, optimize and test a function which does many things than a function which does one. It turns out require is used for *four* things: load a module, load any given file, load a file from @INC, and require a minimum version of Perl. This makes the problem even worse.

Second, if I saw a function in the wild with an interface like that, I would reject it and split it up. The Perl core is held to a *higher* standard than our own code, not a lower one.

Third, we've long since dealt with the new keyword problem. This would be added as a "use 5.16" feature so it would not show up in existing code.

Fourth, it would mean further change to an already very complicated and very important function. I don't want to think about the backwards compatibility consequences. Just changing the prototype on require will be hellish. I'd rather just leave require alone.

Fifth, it makes require even *more* complicated.

Sixth, people aren't writing those functions. A Google Code Search reveals one use of `require_file` and less than a dozen uses of `require_module`. Note there's lots of repeats in there. There's a lot of code GCS doesn't see, but it gives a very good feel for its overall use.

brian d foy | October 3, 2011 4:24 PM | Reply

This is essentially my keynote to Frozen Perl this year. I often ask people the five things they hate about their favorite language, and this is mine for Perl. There are many more consequences of this decision besides the security hole too. We only get the latest version of a module and can only load a namespace once, for instance. It all goes back to require and use.

mpeters.myopenid.com | October 3, 2011 5:25 PM | Reply

AFAIK there are no Perl core keywords (or built-ins) that have underscores and I think that's intentional. So there's already a barrier to entry.

What about `load`?

Michael G Schwern replied to comment from mpeters.myopenid.com | October 3, 2011 5:53 PM | Reply

That "barrier" is entirely cultural, and one we should get to overcoming as it would make things easier to read. But that can be tackled separately. If calling it `requiremodule` and `requirefile` increases the odds of it getting accepted, ok.

`load` has the same issue as `require` in that we need to load several different things, so we need several different names. `loadfile` and `loadmodule` do read a bit better. `load` is a better name, from an objective point of view, as it better describes what the function does... but in a way it is not because it does not convey that we're not just reading the file, we're also compiling it. `require`, it's name giving no clues as to what it does, at least avoids that confusion.

Changing the name from `require` does give the advantage that the third and totally unrelated piece of `require`'s functionality, `require VERSION`, can be given a wholly separate name. Ironically, `require` is not a bad name for that.

The bikeshed is open for business.

Aristotle | October 3, 2011 8:40 PM | Reply

There is a good reason it's called `require` and not something else.

It will only load a file once.

You're saying you require that the file be loaded; if it already has been, your requirement is satisfied.

We already have `load`, it's called `do`.

anthony.derobert.net | October 3, 2011 9:05 PM | Reply

Since when is allowing an attacker to load arbitrary modules out of `@INC` considered safe? I'd hate to encourage such a thing.

Honestly, I've never considered passing untrusted data to require (or do, etc.) a good idea. I'd be wary of doing it even with extreme filtering (e.g., `qr/^My::Namespace::[A-Za-z0-9]+$/`).

Even with only things in @INC, that means I have to be *extremely* careful about installing anything from CPAN.

I mean, I'd be worried about programs not being able to survive the core modules:

```
perl -Mbigint -E 'say 1.5+1.5'
```

Given, of course, just require won't do that, but that's not the case for some modules, and also calling `->import` seems like the next logical step after requiring a module.

Michael G Schwern replied to comment from anthony.derobert.net | October 3, 2011 10:21 PM | Reply

> Since when is allowing an attacker to load arbitrary modules out of @INC considered safe? I'd hate to encourage such a thing.

We don't have to encourage, people already do it. A lot. They need to do it. The question is how do we keep a minor security issue from becoming a major one?

In order to load a module dynamically, for any reason, you're encouraged to do reach for `eval STRING`. This is far more insecure than loading a module out of `@INC`, a set of trusted code in directories under your control. X-Treme filtering might work... or it might not. You have to be *careful*. When it comes to security "don't do that" and "be careful" are just another way to say "future security breach". One security hole I discovered was doing essentially `require eval qq['Our::Namespace::'.$user_input]`. It even filtered $user_input, it just didn't do it right. As a result, cleverly designed input would execute arbitrary code. If they had `require_module` this would not have happened.

If I want to hammer nails, warn me I might hit my thumb, but let me hammer nails. But when I need a hammer, don't hand me a gun.

Michael G Schwern replied to comment from Aristotle | October 3, 2011 10:42 PM | Reply

> There is a good reason it's called require and not something else. It will only load a file once.

I agree that's a good reason it's not called `load`. I don't agree that's a good reason to call it `require`. A good way to judge a name is not how convincing a reason can you come up for it, but how useful is it to a newcomer.

If you showed a newcomer a function called `require`, they would not surmise that it loads and compiles Perl code once and only once. By any conventional vocabulary and logic, the action has little to do with requirements. No other language uses this term. No newcomer will want to load a library and search the docs for "require".

If I showed a newcomer `require Some::Module` I'd expect they'd think it's a declaration of what my code needs (one might say... "requires") but not that it actually goes and gets it. That's what requirements are.

The docs state that "require *demands* that a library be included". That's about as useful as saying `print` *demands* a string appear on your screen. It's a contrived definition written backwards to support the name. Almost like an apologist historian rewriting history to pretend everything is ok.

Aristotle | October 4, 2011 1:02 AM | Reply

I find it hard to believe that it's written backwards. How do you come up with the name `require` at all if not by starting with that definition in the first place? Do you care to try to imagine?

Nor does no other language use it. At least PHP, Ruby, (future) Javascript and Lua have a "`require`" function or keyword.

It isn't unrelated by "any conventional" logic, it is unrelated by *your* logic. Maybe it is even unrelated by *many people's* logic. But your absolutist "it just makes no sense to anyone reasonable" stance is a failure of empathy and quite presumptuous. "If I showed a newcomer" you say; have you? I always find the argument that other hapless people would be confused, when the arguer himself never was, to be odious. Yes, let's think of the children.

Besides: if you want self-documenting, "`use`" is far more nondescript. Likewise is "`do`". How about "`my`"? And "`our`"? How useful a name is to a complete outsider is an important metric, but not the only one. Outsiders soon stop being so, one would hope. The skill in Larry's naming is in capturing intent without getting mired in operational specifics: they are chosen so well that it takes a single explanation for them to stick. To me that is a far more important goal than bald self-descriptiveness.

moritz | October 4, 2011 6:55 AM | Reply

You are totally right, using a module is too much "black box" in Perl 5.

And while require_module is certainly a start in the right direction, it might make more sense to start exposing the more low-level operations to the user space: Validation of a module name, translation from module name to file name, and looking up a file name in @INC.

Aristotle | October 4, 2011 1:20 PM | Reply

I thought about "`do_once`" (or "`doonce`", hmm) for the name of the path-based function. But `do` doesn't care about `@INC`, and since the new function does, it really has to have something to do with `require`.

I also wonder about `require { file => "Foo/Bar.pm" }` (and correspondingly for `package => 'Foo::Bar'`; I'm on the fence about "module" for that one because no other builtin uses that term) which would work just fine with the current prototype. It's true that this would complicate `require` further, but the interface is unambiguous, both internally and with respect to the interface it extends. The only reason I'm unsure is that there'd be no other builtin that has this interface. On the upside, there would be no need to make this version-dependent, since it *can't* break any existing code.

It also bugs me that the UNIVERSAL::require approach is problematic. It's just so perfect. I'd really like to figure out a way to make it safe, even it needs some magic. But I'm not confident that I'm thinking of every possible problem it causes, which would be necessary for thinking about it.

mrstlee | October 4, 2011 2:06 PM | Reply

Whenever I absolutely don't want

```
require $thing
```

to load any old thing I test

```
$thing
```

first to see if it comes from a pre-determined list of trusted directory paths. Sure it is still possible for the mischevious/malevolent to wreak havoc - but at least the person/s responsible are allowed to.

I understand why this issue bothers folk with tidier minds than mine but, well, require has been around for an awful long time now, and the earth is still turning.

Is the problem something that the core language needs to solve, or one that is left for concerned parties to solve via other means? I truly don't know. If it can be solved in a non-invasive manner without inordinate time and effort then fair enough. Doesn't seem to be the case though.

Or maybe the perl community tries an Apple-flavoured compromise? Set up CYCT (CPAN You Can Trust) which only accepts modules that conform to explicitly defined best practices.

Reini Urban | October 4, 2011 3:06 PM | Reply

Added a patch to Module::Load RT https://rt.cpan.org/Public/Bug/Display.html?id=71442

Reini Urban | October 4, 2011 3:15 PM | Reply

I see no general problem in allowing `m|^\.[/\\]|` in require file, I just disabled absolute paths and attempts to step the path upwards in strings given to Module::Load.

I see no immediate action required to act almost hysterically on `require string` on such paths, such as adding new ops. I would recommend to fix the loader modules first, and bring the message out to check for generated strings, esp. with unsafe user input.

`..::` should also be forbidden in bareword module names. This should be forbidden in `Perl_pp_require`, agreed.

dagolden.com | October 4, 2011 5:45 PM | Reply

I appreciate that you're pointing out yet another security loophole that people might not think of.

Yet the only real lesson I see is that Perl code shouldn't incorporate unvalidated, arbitrary user input into its execution logic, and that's not really a new warning. Running under taint would avoid the whole threat described in this post.

That said, I agree that in retrospect the decision to overload module and file loading was a poor one.

-- David

bingos | October 5, 2011 2:44 PM | Reply

Module::Load (version 0.22) has been fixed to not jump the rails now

Michael G Schwern replied to comment from dagolden.com | October 23, 2011 3:19 AM | Reply

> Running under taint would avoid the whole threat described in this post.

Taint mode just tells you if your inputs have been checked. It does **not** tell you that your check is secure. While it will help to use taint mode, and add a filter, it still means you're passing user input to an insecure function (whether "eval qq[require $module]" or "require $path"). There's no reason that function has to be so insecure. Security must be many layered to work.

We all know taint is a royal pain in the ass, made some what less by the -t flag (you're welcome). Because its effects are global it is nigh useless for any decent sized project. It is telling the user "be careful" which doesn't work. Eventually someone will slip.

Even with the safety on, hammering nails with a gun is not safe.

Peter Rabbitson | February 13, 2012 11:15 AM | Reply

Hi, and sorry for resurrecting this old thread.

This blogpost seems to be bookmarked among many fellow programmers, but it fails to even mention Module::Runtime, which (at the time of this writing) already was implementing everything you describe, except for the unicode stuff (which it considers invalid, see RT#74804).

Would you mind updating this post for the sake of fellow googlers?

Cheers!

Leave a comment

Sign in to comment.