

Mastering Perl

by brian d foy
The Perl Review
version 1.71
July 15, 2010

Contents

Introduction

About this course

The path to mastery

Modulinos

Programs versus modules

Bring back main()

Bring back main(), continued

Tell Perl where to start

Make it a module

run() as a class method

Who's calling?

Who's calling? (output)

caller() in a module

Compile as a module, run as a program

Testing our program

Adding to the program

Adding to the program, continued

Finer-grained testing

Packaging

Wrapper programs

Installing programs

Altering the shebang lines

Other methods

Distribute through CPAN

Conclusion

Further reading

23

24

25

26

2 Jury rigging modules

Sometimes modules don't work

28

3

Maintaining your local version

29

Send a patch to the author

30

Some authors disappear

31

Some authors hate you

32

Jury rigging methods

33

Change a copy

34

Globally replace a subroutine

35

Locally replace a subroutine

36

10

Save the original definition

37

11

Move a subroutine definition

38

12

Wrapping subroutines

39

13

Handling context

40

14

Hook::LexWrap

41

15

Watch before and after

42

16

These are only temporary fixes

43

17

Methods are a bit different

44

18

Make a subclass

45

19

Override a method

46

20

Extend a method

47

21

Further reading

48

22

Data Security

Caveats	50
Bad data can ruin your day	51
Use three-argument open	52
Use it with strings too	53
Use it with strings too, continued	54
You can also read from strings	55
Use list form of system and exec	56
Use list form of system and exec, continued	57
IPC::System::Simple	58
Don't trust external data	59
Taint checking	60
Taint checking, continued	61
Taint environments	62
Taint environments, continued	63
Tainted arguments	64
Tainting is viral	65
Tainting is viral, continued	66
Side effects of tainting	67
Untainting data	68
The American method	69
The Prussian method	70
Scoped regex tainting	71
Choosing good data with tainted data	72
Tainted I/O	73
Tainted I/O, continued	74
Taint warnings instead of errors	75

The -U switch	76
Tainting DBI	77
Use DBI placeholders	78
Use DBI placeholders, continued	79
Use different database handles	80
How users can cheat	81
Further Reading	82
Further Reading, continued	83

Profiling

Profiling is better than benchmarking	85
The basics of profiling	86
A recursive subroutine	87
Calling a Profiler	88
Recursion profile	89
Iteration, not recursion	90
Iteration profile	91
Really big numbers	92
Memoize	93
Memoize, continued	94
What happened?	95
Modern profiling with NYTProf	96
Record DBI queries	97
Database optimization	98
Profiling DBI Statements	99
Profiling DBI methods	100
Profiling test suites	101
Devel::Cover HTML report	102

Devel::Cover detail	103
Further reading	104

Conclusion

Main points	106
More information	107

Questions

Mastering Perl

by brian d foy
The Perl Review
version 1.71
July 15, 2010

Introduction

About this course

- Selected topics based on *Mastering Perl*
- Mostly *not* about syntax or wizardly tricks
- Not for masters, but people who want to control Perl code
- Not necessarily the way to do it, just the way I've done it
- Create “professional”, robust programs other people can use
- We'll cover
 - * modulos
 - * jury rigging
 - * profiling
 - * security

The path to mastery

- The guild system had a progression of skills
- Apprentices were the beginners and worked with supervision
- Journeymen were competent in their trade
- Masters taught journeymen
- Journeymen studied under different masters
 - * different masters teach different tricks and methods
 - * journeyman develop their own style
- A masterpiece showed that a journeyman mastered his trade

Modulinos

Programs versus modules

- For most people, programs or scripts are our main effort in everyday work.
- However, all of the good development tools are for modules, including tools for:
 - * Testing
 - * Packaging
 - * Distribution
 - * Installation
- We can combine the two so programs get the benefits of modules.
- A *modulino* is a little module that acts like both a module and a program. It just needs to serve the application instead of the general case.

Bring back main()

- In some languages, I have to let the computer know where to start my program:

```
/* hello_world.c */  
  
#include <stdio.h>  
  
int main ( void ) {  
    printf( "Hello C World!\n" );  
  
    return 0;  
}
```

- A Perl program implies a `main()` loop for us as the `main::` package. Normally I write:

```
print "Hello Perl World!\n";
```

Bring back main(), continued

- I can rewrite that to bring back `main()`:

```
#!/usr/bin/perl
```

```
sub main {  
    print "Hello Perl World!\n";  
  
    # Perl still adds the exit 0 for us  
}
```

- However, the Perl program doesn't know where to start!

Tell Perl where to start

- Since `main()` isn't special, I have to tell Perl what to run:

```
#!/usr/bin/perl
```

```
main();
```

```
sub main {  
    print "Hello Perl World!\n";  
}
```

- Calling it `run()` sounds more like what I want:

```
#!/usr/bin/perl
```

```
run();
```

```
sub run {  
    print "Hello Perl World!\n";  
}
```

- I'm at the same place I started, but now I can take the next step to make it a modolino.

Make it a module

- A module is really a package with some subroutines. Sometimes it's a classical library, and other times it's an object-oriented class.
- Modules compile code but don't run code until we tell it too.
- With my `run()` subroutine, I almost have the same setup as a regular module.
- I add an explicit package and treat `run()` as a class method. I save it in *MyApplication.pm*.

```
#!/usr/bin/perl
```

```
package MyApplication;
```

```
__PACKAGE__->run();
```

```
sub run {  
    print "Hello Perl World!\n";  
}
```

Make it a module, continued

- I'm still running code just by loading this module (assuming `.` is in `@INC`):

```
$ perl -MMyApplication -e 'dummy program'  
Hello Perl World!
```

- And I can still run it as a script:

```
$ perl MyApplication.pm  
Hello Perl World!
```

Who's calling?

- `caller()` gives us information about the call stack.
- It's usually part of a subroutine:

```
#!/usr/bin/perl
```

```
my @caller_info = caller();  
print "top: @caller_info\n";  
middle();
```

```
sub middle {  
    my @caller_info = caller();  
    print "middle: @caller_info\n";  
    bottom()  
}
```

```
sub bottom {  
    my @caller_info = caller();  
    print "bottom: @caller_info\n";  
}
```

Who's calling?, continued

- It returns the package, filename, and line number of the code that invoked the subroutine:

```
top:                                     # empty list for the top level
middle: main /Users/brian/Desktop/caller.pl 5
bottom: main /Users/brian/Desktop/caller.pl 10
```

caller() in a module

- In scalar context, `caller()` returns true if it is not at the top level (so, something called the current code).

- As a module, the caller is the code that loaded the modulino:

```
#!/usr/bin/perl
```

```
package MyCalledApplication;
```

```
print "Caller was true!\n" if caller();
```

- From the command line, `caller()` returns true if I load the modulino with `-M`:

```
$ perl -MMyCalledApplication -e 'dummy program'  
Caller is true!
```

- As a program, `caller()` returns false because it is at the top level.

```
$ perl MyCalledApplication.pm  
$
```

no output because caller is false

caller() in a module, continued

- Now I know how to tell if I am using a file as a modulino or a program: just check `caller()`:
 - * true: modulino
 - * false: program

Compile as a module, run as a program

- When I load *MyApplication.pm* as a module, I don't want it to run yet.
- If it acts like a library then I can load it and use its subroutines, especially for unit testing.
- I have to delay my call to my `run()`, and I can use `caller` to do that.
- We don't want to run as a program is `caller()` returns true:

```
#!/usr/bin/perl
```

```
package MyApplication;
```

```
__PACKAGE__->run() unless caller();
```

```
sub run {  
    print "Hello Perl World!\n";  
}
```

Testing our program

- Most programs are hard to test because I can't get at the pieces of them without running all of the other stuff.
- If I write my programs as modules and separate portions into subroutines, I can test it just like any other module.

```
use Test::More tests => 3;  
use Test::Output;
```

```
my $class = 'MyApplication';
```

```
use_ok( $class );
```

can I load the module?

```
can_ok( $class, 'run' );
```

does it have the subroutine I need?

```
stdout_is(  
    sub{ $class->run() },  
    "Hello Perl World!\n"  
);
```

Adding to the program

- Now that I can test parts of it, I should separate it into as many parts as reasonably possible.
 - * There is some overhead with method calls, so don't go crazy
 - * The more I can break it into pieces, the easier it is for other people to subclass.
- Perhaps I don't like the "Hello Perl World!" message. To change it, I have to override all of the `run()` method. That's no fun.

Adding to the program

- Instead, I rewrite *MyApplication.pm* so the action and the data are separate:

```
#!/usr/bin/perl
```

```
package MyApplication;
```

```
__PACKAGE__->run() unless caller();
```

```
sub run {  
    print $_[0]->message, "\n";  
}
```

```
sub message {  
    "Just Another " . $_[0]->topic . " Hacker,"  
}
```

```
sub topic { "Perl" }
```

Finer-grained testing

- Now with several components, I can test parts of it separately:

```
use Test::More tests => 7;
use Test::Output;

my $class = 'MyApplication';

use_ok( $class );

can_ok( $class, 'topic' );
is( $class->topic, 'Perl',
    'The default topic is Perl' );

can_ok( $class, 'message' );
is( $class->message,
    'Just Another Perl Hacker,' );

can_ok( $class, 'run' );
stdout_is( sub{ $class->run() },
    "Just Another Perl Hacker,\n" );
```

Packaging

- Since my program now behaves like a module, I can package it as a module.
- There's nothing particularly special about creating the module, so use your favorite tool to do it.
- `Module::Starter`

```
$ module-starter --module=MyApplication  
  --author=Joe \  
  --email=joe@example.com
```
- `Distribution::Cooker`

```
$ dist_cooker MyApplication
```
- It's easier to do this before I write *MyApplication.pm* so all the documentation and other bits are there.
- If I don't start this way, I just copy the *MyApplication.pm* file into the right place.

Wrapper programs

- Even though the module file acts like a program, it's usually not in the user's path.
- I have a couple ways to make my program available. The best is probably a wrapper script that passes the arguments to the module.

- Here's the modern `perldoc` program:

```
require 5;  
BEGIN { $^W = 1 if $ENV{'PERLDOCDEBUG'} }  
use Pod::Perldoc;  
exit( Pod::Perldoc->run() );
```

- The `dist_cooker` program from `Distribution::Cooker` does the same sort of thing:

```
use Distribution::Cooker;  
  
Distribution::Cooker->run( @ARGV );
```

Installing programs

- For MakeMaker, you list the programs you want to install in the EXE_FILES parameter to WriteMakefile():

```
use ExtUtils::MakeMaker;

WriteMakefile(
    ...
    EXE_FILES => [ qw(script/my_program) ]
);
```

- For Module::Build, use the script_file parameter to new:

```
use Module::Build;
my $build = Module::Build->new(

    script_files    => ['script/dist_cooker'],
    ...
);

$build->create_build_script;
```

Installing programs, continued

- Both of these alter your script slightly to make it work for the person installing the script

- * Alter the shebang line for the perl that invoked the build script

- * Adds some shell magic to find perl in odd cases:

```
#!/usr/local/perl5/perls/perl-5.10.1/bin/perl
    eval 'exec /usr/local/perl5/perls/perl-5.10.1/
bin/perl -S $0 ${1+"$@"}'
    if $running_under_some_shell;
```

Other methods

- I don't have to create a separate program if I can link to the module file.
 - * Not all systems support linking
- In the pre-build, I can copy the module file to a file with the program's name.
 - * The module docs and the program docs would be the same
 - * I could make separate doc pages (*program.pod*, *my_program.1*, *my_program.html*)

Distribute through CPAN

- CPAN has a “Script Archive”, but virtually nobody uses it.
- The `App::` namespace collects distributions that represent applications
- As a distribution, there is nothing special about my program. Install it like a module:

```
$ cpan App::MyApplication
```
- For free, I automatically get:
 - * RT bug tracking
 - * CPAN Testers reports
 - * AnnoCPAN
 - * *and much more*
- If this isn't open source, you can still create your own CPAN and use the same open source tools for all of that.

Conclusion

- All the good tools are built around modules and distributions.
- Modules are easy to test, so write programs based on modules.
- Distribute programs as normal Perl distributions.

Further reading

- “How a Script Becomes a Module” originally appeared on Perlmonks:

`http://www.perlmonks.org/index.pl?node_id=396759`

- I also wrote about this idea for *The Perl Journal* in “Scripts as Modules”. Although it’s the same idea, I chose a completely different topic: turning the RSS feed from *The Perl Journal* into HTML:

`http://www.ddj.com/dept/lightlang/184416165`

- Denis Kosykh wrote “Test-Driven Development” for *The Perl Review* 1.0 (Summer 2004) and covers some of the same ideas as modulino development:

`http://www.theperlreview.com/Issues/subscribers.html`

Jury rigging modules

Sometimes modules don't work

- Modules might not work for various reasons
 - * design bugs
 - * conflicts with other modules
 - * interfaces change
 - * underlying libraries change
 - * an older version works, but the newer one doesn't
- You want to fix them, but there are some problems
 - * you don't want change the original source
 - * you don't want to maintain a fork
 - * you want your changes to make it in the main line

Maintaining your local version

- You might maintain a local version
- But if you change the original source, you might overwrite it
- CPAN tools always install the latest CPAN versions, but only if it thinks your version is older.
- You could set the version to be virtually infinite:

```
our $VERSION = 0xFFFFFFFF;
```
- But now you can't update your local version, and it might be incompatible with updates for other modules.

Send a patch to the author

- The least amount of work is to get the module maintainer to incorporate your fix.
- Git is handy because you don't need a server
- Download the source and make a git archive:

```
% cd Some-Module-1.23
% git init
% git add .
% git commit -a -m "Some::Module 1.23"
```
- Make your changes, and commit again:

```
% git commit -a -m "Explain your changes"
```
- Make some diffs:

```
% git diff XXX
```
- Most distros use *<http://rt.cpan.org>*
- Some distros are in Github.

Some authors disappear

- The distribution maintainer might be long gone
- PAUSE has a process to let people take over abandoned modules
- *http://www.cpan.org/misc/cpan-faq.html#How_adopt_module*
- Sometimes you can even convince someone else to take it over

Some authors hate you

- Well, maybe not hate, but they don't want your patches.
- That's different than them working slower than you'd prefer.
- If you've been patient and nothing else works, a fork might be appropriate.
- Make your changes, upload to PAUSE with new package names.
- Now you get to be the maintainer who disappears.
- That's the most amount of work, and work is bad.

Jury rigging methods

- There are a variety of ways to do things, each appropriate for different sorts of fixes.
 - * change a copy of the source
 - * replace subroutines
 - * wrap subroutines
 - * subclass and extend
 - * subclass and override

Change a copy

- Instead of changing the original source, change a copy
- Reverting isn't as foolproof as it should be.
- Copy the original source to a new file.
- Make your changes, without ever losing the original.
- Adjust PERL5LIB to load your version:
`export PERL5LIB=/dir/with/copy:$PERL5LIB`
- Perl always loads the first one it finds, not the latest version.
- To find the one you loaded, check `%INC` at the end

```
END {  
    use Data::Dumper;  
    print Dumper( \%INC );  
}
```

Globally replace a subroutine

- I can override the broken subroutine in my program:

```
BEGIN {  
    use Broken::Module;           get old definitions first!  
    package Broken::Module;  
    no warnings 'redefine';  
  
    *broken_sub = sub {  
        # fixed code;  
    };  
}
```

- When the module is fixed, I can remove this code.
- With a little extra work, I can limit the fix to specific versions:

```
unless (eval { Broken::Module->VERSION('1.23') })  
{  
    *broken_sub = sub {...};  
}
```
- The `version` module provides facilities for version math, too.

Locally replace a subroutine

- I can override the broken subroutine temporarily:

```
use Broken::Module;
```

get old definitions first!

```
{  
no warnings 'redefine';  
package Broken::Module;  
  
local *broken_sub = sub {  
    # fixed code;  
};  
  
broken_sub( @args );  
}
```

Save the original definition

- Maybe you want to save the original subroutine:

```
use Broken::Module;
```

get old definitions first!

```
my $old_broken_sub = \&broken_sub;
```

```
{
```

```
package Broken::Module;
```

```
no warnings 'redefine';
```

```
*broken_sub = sub {
```

```
    # fixed code;
```

```
};
```

```
}
```

```
broken_sub( @args );
```

```
$old_broken_sub->( @other_args );
```

Move a subroutine definition

- You can also rename the bad subroutine:

```
use Broken::Module;
```

get old definitions first!

```
{  
package Broken::Module;  
*old_broken_sub = \&broken_sub;
```

```
no warnings 'redefine';  
*broken_sub = sub {  
    # fixed code;  
};  
}
```

```
broken_sub( @args );
```

```
old_broken_sub( @other_args );
```

Wrapping subroutines

- Sometimes you can just wrap the subroutine.
- You can wrap a subroutine so you can adjust input and output:

```
sub wrapped_foo
{
  my @args = @_;
  ...;                               # prepare @args for next step;
  my $result = foo( @args );
  ...;                               # clean up $result
  return $result;
}
```

Handling context

- You might have to do more than you really imagined:

```
sub wrapped_foo
{
  my @args = @_;
  ...;                                     # prepare @args for next step;
  if( wantarray ) {                       # list context
    my @result = foo( @args );
    return @result;
  }
  elsif( defined wantarray ) {           # scalar context
    my $result = foo( @args );
    ...;                                  # clean up $result
    return $result;
  }
  else {                                  # void context
    foo( @args );
  }
}
```

Hook::LexWrap

- Hook::LexWrap can handle all of the details:

```
use Hook::LexWrap;

wrap 'sub_to_watch',
  pre =>
    sub { print "The arguments are [@_]\n" },
  post =>
    sub { print "Result was [$_-1]\n" }
;

sub_to_watch( @args );
```

Watch before and after

- Use `Hook::LexWrap` to see before and after a subroutine, globally:

```
use Hook::LexWrap;
```

```
sub divide {  
    my( $n, $m ) = @_;  
    my $quotient = $n / $m;  
}
```

```
wrap 'divide',  
    pre =>  
        sub { print "The arguments are [ @_ ]\n" },  
    post =>  
        sub { print "Result was [ $_[ -1 ] ]\n" };
```

```
my $result = divide( 4, 4 );
```

- This is very handy for debugging.

These are only temporary fixes

- None of these are long term solutions.
- What if someone wants to patch your patch? Which redefinition gets there first?
- Or when you want to back out your changes? What is the final definition?

Methods are a bit different

- Don't try any of this with methods, which are different beasts.
- Their definition might not be where you think it is due to inheritance.

Make a subclass

- If you can, create a subclass.
- You can override or extend just the broken parts.
- Start with an empty subclass (the null subclass test):

```
package Local::Foo
use parent qw(Foo);
1;
```

*Local shouldn't ever conflict
or base.pm*

- Adjust your program to use your subclass:

```
# use Foo
use Local::Foo;
```

```
#my $object = Foo->new();
my $object = Local::Foo->new( ... );
```

- Your program should still work.
- If not, there are even more bugs in the module.

Override a method

- Overriding replaces the definition of a method

```
package Local::Foo  
use parent qw(Foo);
```

```
sub some_method  
{  
  my( $class, @args ) = @_  
  ...;  
}
```

do what you need to do

```
1;
```

Extend a method

- Extending adds to the definition of a method
- You could provide an adapter:

```
package Local::Foo
use parent qw(Foo);

sub some_method
{
    my( $class, @args ) = @_ ;
    ... munge arguments here
    my $self = $class->SUPER::some_method(
        @args );
    ... do my new stuff here.
}

1;
```

Further reading

- The *perlboot* documentation has an extended subclassing example. It's also in *Intermediate Perl*.
- I talk about `Hook::LexWrap` in “Wrapping Subroutines to Trace Code Execution,” *The Perl Journal*, July 2005: <http://www.ddj.com/dept/lightlang/184416218>.
- The documentation of `diff` and `patch` discusses their use. The `patch` manpage is particularly instructive because it contains a section near the end that talks about the pragmatic considerations of using the tools and dealing with other programmers.

Data Security

Caveats

- This isn't a security course, so we're not talking about application-level stuff.
- The Perl language has some features that can cause some pain if you don't use them wisely.
- We'll cover some basic good practices
- Most of the section features taint-checking
- This isn't comprehensive

Bad data can ruin your day

- Most programs have to deal with external data and resources.
- Given any chance to give input, people will do it wrong.
- Not checking file names is more common than we would expect:

```
open FILE, $input{in_file};  
while( <FILE> ) { print }
```

- Imagine some of the input that could mess up this poor code:

```
/etc/passwd  
rm -rf |
```

- The problem is a pre-Perl 5.6 thing when we only had the filename to do everything:

```
open FILE, 'output.dat';  
open FILE, '> output.dat';  
open FILE, '>> output.dat';  
open FILE, 'program |';  
open FILE, '| program';
```

- Not only that, none of these check errors!

Use three-argument open

- With Perl 5.6 and later we can fix problems by separating the modes from the name.

```
open FILE, ">", $file or die "Could not open  
$file: $!";
```

- Even if we are reading files, use the three-arguments just to be sure

```
open FILE, "<", $file or die "Could not open  
$file: $!";
```

Use it with strings too

- Okay, this really has nothing to do with security, but since we're talking about `open`, now's a good time for this.

- Most people build up strings with concatenation:

```
while( <$fh> ) {  
    my $record = ...do some processing...;  
    $string .= $record;  
}
```

- Do it with a filehandle instead by using a scalar reference

```
my $file = \ '  
open my($output), '>', $file or die ...;
```

```
while( <$fh> ) {  
    my $record = ...do some processing...;  
    print $output, $record;  
}
```

Use it with strings too, continued

- No more special `as_string` method code!

```
sub as_string {
    my $self = shift;
    my $string = \ '';
    open my($output), '>', $string or die ...;
    $self->to_fh( $output );
}
```

You can also read from strings

- Multi-line regexes can be a pain.

```
my @matches = m/^.....$/m;
```

what's \$/

- You might think splitting is better:

```
my @lines = split /$/, $string;
while( @lines ) { ... }
```

- If you want to deal with strings line--by-line, read from them as a filehandle:

```
open my( $fh ), '<', \ $string;
```

```
while( <$fh> ) {
    ... process line from string ...
}
```

- No more splitting on lines!

Use list form of system and exec

- The system and exec built-ins have the problem too:

```
system( "/bin/echo $message" );
```

WRONG!

- What's in message? Maybe there are shell metacharacters!

```
'Hello World!'; mail joe@example.com < /etc/  
passwd
```

- In the single argument form, Perl passes everything to the shell just as it is. The shell then interprets it as it likes.

- In the multiple argument form, Perl quotes the meta-characters for me:

```
@args = ( "/bin/echo", $message );  
system @args;
```

list form, which is fine.

- That's still a problem is everything shows up in `$args[0]`, making it the single argument call again:

```
my @args = ( '/bin/echo; rm -rf /' );  
system @args;
```

still only one argument!

Use list form of system and exec, continued

- I get around this with a bit of indirect object notation that always uses the list mode:

```
system { $args[0] } @args;
```

- Whatever is in `$args[0]` is the command name. There shouldn't be a command named `'/bin/echo; rm -rf /'`
- This is still a bit platform-dependent.

IPC::System::Simple

- `system` and `exec` interact with the shell.
- Mostly, we don't care as long as we get the answer.
- Paul Fenwick spent a lot of time figuring out the edge cases on various platforms and put it all into `IPC::System::Simple`, available on CPAN.

- The `systemx` and `capturex` versions never touch the shell:

```
use IPC::System::Simple qw(systemx capturex);
```

```
systemx( $command, @args );           like system(), but no shell
```

```
my $output = capturex( $command, @args );           like  
backticks, but no shell
```

```
my @output = capturex( $command, @args );
```

- `IPC::System::Simple` also handles all of the operating system specific problems.

Don't trust external data

- Avoiding the shell keeps the shell from doing some damage, but we should catch problems sooner.
- Examine the data before you use it.
- There are many sources of external data:
 - * user input
 - * environment variables
 - * command-line arguments
 - * data files
 - * config files

Taint checking

- Perl has a special mode that can mark data as tainted and trace it through the entire program.
- Anything that touches the tainted data also becomes tainted.
- Perl stops you from sending tainted data outside the program.
- Taint-checking affects the entire program, and you have to turn it on before you start doing anything.

- Use the `-T` switch from the command line:

```
% perl -T program.pl
```

- Or on the shebang line:

```
#!/perl -T
```

- For modperl, turn on taint checking in the apache configuration

```
PerlTaintCheck On  
PerlSwitches -T
```

```
mod_perl 1  
mod_perl 2
```

Taint checking, continued

- Taint-checking is automatically on if the real and effective user or group is different
- There's a big caveat here: taint-checking is a development tool, not a guarantee that nothing bad will happen.
- It's easy for programmers to defeat taint-checking, so you still have to examine code.

Taint environments

- `%ENV` is tainted because it is external data.

```
#!/usr/bin/perl -T
system qq|echo "Hello Perl!"|;
```

- The error message tells us that `PATH` is suspicious:

```
Insecure $ENV{PATH} while running with -T
switch at ...
```

- What happens if someone made thier own echo?

```
$ cat >> echo
rm -rf /
^D
$ export PATH=.:$PATH
$ perl program.pl
```

- Now we're running the wrong echo!
- Perl knows this and only allows certain paths in `$ENV{PATH}`

Taint environments, continued

- The best thing to do is to scrub the values and assign your own:

```
delete @ENV{qw(IFS CDPATH ENV BASH_ENV)};  
$ENV{PATH} = '/usr/bin/local:/usr/bin';
```

- Better yet, use full paths everywhere:

```
#!/usr/bin/perl -T  
delete $ENV{PATH};  
system "/bin/cat /Users/brian/.bashrc"
```

Tainted arguments

- The command-line arguments are tainted too.
- We can checked taintedness with `Scalar::Util`:

```
#!/usr/bin/perl -T
# tainted-args.pl
```

```
use Scalar::Util qw(tainted);
```

```
# this one won't work
print "ARGV is tainted\n" if tainted( @ARGV );
```

```
# this one will work
print "Argument [$ARGV[0]] is tainted\n" if
    tainted( $ARGV[0] );
```

- When we run this command, Perl stops us:

```
$ perl tainted-args.pl foo
Argument [foo] is tainted
```

Tainting is viral

- Any tainted data affects data we build from them:

```
#!/usr/bin/perl -T
use strict;
use warnings;

use File::Spec;
use Scalar::Util qw(tainted);

my $path = File::Spec->catfile( $ENV{HOME},
    "data.txt" );
$path is tainted

print "Result [$path] is tainted\n" if tainted(
    $path );

open my($fh), $path or die "Could not open
    $path";

print while( <$fh> );
```

Tainting is viral, continued

- The problem is `$ENV{HOME}`. What if it has a pipe in it?

```
$ HOME="| cat ../../../../etc/passwd;" ./sub*
```

- Perl catches that:

```
Insecure dependency in piped open while running  
with -T switch at ...
```

- We could also solve this with three-argument open:

```
open my($fh), '<', $path or die "Could not open  
$path";
```

Side effects of tainting

- Perl ignores some external data when we turn on taint-checking, like PERLLIB and PERL5LIB.
- You can still change @INC:

```
$ perl -Mlib=/Users/brian/lib/perl5 program.pl
```

```
$ perl -I/Users/brian/lib/perl5 program.pl
```

```
$ perl -I$PERL5LIB program.pl
```

Untainting data

- The only *APPROVED* way to untaint data is with a regex that captures the data:

```
my( $file ) = $ARGV[0] =~ m/^( [A-Z0-9_.-]+ )$/  
    ig; $file is not tainted
```

- The lazy programmer can easily cheat:

```
my( $file ) = $ARGV[0] =~ m/(.*)/i;
```

- If we're in a non-ASCII environment, matching just A to Z isn't any good. The locale pragma knows how to deal with `\w`.

```
{  
use locale;
```

```
my( $file ) = $ARGV[0] =~ m/^( [ \w . - ]+ )$/;  
}
```

- There are two philosophies on untainting data: the Prussian and the American way.

The American method

- The American method disallows characters that it thinks are bad
`my($file) = $ARGV[0] =~ m/ ([^$% ; |]+) /i ;`
- We have to be really careful that we list all the possible bad characters.
- This isn't a good solution

The Prussian method

- The Prussian method checks that the data only has allowed characters:

```
my( $file ) = $ARGV[0] =~ m/ ([a-z0-9_.-]+) /i;
```

- Maybe I miss some allowed characters, but missing valid input is better than missing malicious input.
- Taking it even farther, we can specifically turn off the untainting features:

```
{  
use re 'taint'; actually turns off untainting
```

```
# $file still tainted  
my( $file ) = $ARGV[0] =~ m/ ^ ( [ \w . - ] + ) $ / ;  
}
```

Scoped regex tainting

- We can turn off untainting for all regexes and only turn on untainting when we need it:

```
use re 'taint';
```

```
{  
no re 'taint';
```

```
# $file not tainted  
my( $file ) = $ARGV[0] =~ m/^( [\w.-]+ )$/;  
}
```

Choosing good data with tainted data

- We can choose the good data with tainted data, and the taint does not affect

```
my $value = $tainted_scalar ? "Fred" :  
    "Barney";
```

- The ternary operator is really just shorthand for the full `if()` structure:

```
my $value = do {  
    if( $tainted_scalar ) { "Fred" }  
    else { "Barney" }  
};
```

Tainted I/O

- Data that I read from files is tainted too:

```
use Scalar::Util qw(tainted);
```

```
open my($fh), $0 or  
    die "Could not open myself! $!";
```

```
my $line = <$fh>;
```

```
print "Line is tainted!\n" if tainted( $line );
```

Tainted I/O, continued

- Untaint data per-filehandle by using the `IO::Handle` module:

```
use IO::Handle;
use Scalar::Util qw(tainted);

open my($fh), '<', $0 or
    die "Could not open myself! $!";

$fh->untaint;

my $line = <$fh>;

print "Line is not tainted!\n" unless tainted(
    $line );
```

Taint warnings instead of errors

- If you are adding taint-checking to an existing script, you might not be able to get it to run quickly.

```
#!/usr/bin/perl -T  
# print_args.pl
```

```
system qq|echo "Args are @ARGV"|;
```

- Instead of real taint-checking, we can get taint-warnings with `-t` to find the problems but not stop the script:

```
$ perl -t print_args.pl foo bar  
Insecure $ENV{PATH} while running with -t  
switch at ....  
Insecure dependency in system while running  
with -t switch at ...  
Args are foo bar
```

The -U switch

- We can also disable taint-checking with `-U`, but we don't get warnings:

```
$ perl -TU print_args.pl foo bar  
Args are foo bar
```

- We can get warnings back with `-w`:

```
$ perl -TUw print_args.pl foo bar  
Insecure $ENV{PATH} while running with -T  
switch at ....  
Insecure dependency in system while running  
with -T switch at ...  
Args are foo bar
```

Tainting DBI

- Tainting works because Perl recognizes when we are explicitly using an external resource.

- It can't tell when modules, such as DBI, might harm us.

- DBI can turn on its own taint mode:

```
my $dbh = DBI->connect( $dsn, $user, $password,  
    { TaintIn => 1, ... }  
    );
```

- We can also tell DBI to taint the results:

```
my $dbh = DBI->connect( $dsn, $user, $password,  
    { TaintOut => 1, ... }  
    );
```

- Or we can do both at the same time:

```
my $dbh = DBI->connect( $dsn, $user, $password,  
    { TaintIn => 1, TaintOut => 1, ... }  
    );
```

Use DBI placeholders

- Database operations can have the same problem:

```
use CGI;  
use DBI;
```

```
my $cgi    = CGI->new;  
my $dbh    = DBI->connect( ... ); # fill in the  
    details yourself  
my $name   = $cgi->param( 'username' );
```

```
my $query = "SELECT * FROM Users WHERE  
    name='$name'";
```

WRONG!

- What is in that username parameter? Maybe it's an SQL injection:

```
buster'; DELETE FROM Users; SELECT * FROM Users  
WHERE name='
```

Use DBI placeholders, continued

- Avoid the problem with a prepared statement that uses placeholders:

```
my $sth = $dbh->prepare("SELECT * FROM Users  
WHERE name=?");  
my $rc  = $dbh->execute( $name );
```

- Placeholders handle proper quoting and escaping, and can also do some very basic validation:

```
$sth->bind_param(1, $value,  
{ TYPE => SQL_INTEGER }  
);
```

Use different database handles

- Create separate database users with only the permissions that they need:

- * Read only

- * Update only

- Create different database handles for each:

```
my $dbh_reader = DBI->connect( $dsn, $reader,  
    $reader_password,  
    { TaintIn => 1, TaintOut => 1, ... }  
);
```

```
my $dbh_updater = DBI->connect( $dsn, $updater,  
    $updater_password,  
    { TaintIn => 1, TaintOut => 1, ... }  
);
```

How users can cheat

- Even if you never cheat, someone around you probably will and you need to recognize their tricks.

- They can just match everything:

```
my( $file )= $input =~ m/(.*)/;
```

- They can use hash keys, which aren't real SVs (scalar value structures in perl internals)

```
my @data = keys %{ map { $_, 1 } @input };
```

Further Reading

- Start with the *perlsec* documentation, which gives an overview of secure programming techniques for Perl.
- The *perltaint* documentation gives the full details on taint checking. The entries in *perlfunc* for `system` and `exec` talk about their security features.
- The *perlfunc* documentation explains everything the `open` built-in can do, and there is even more in *perlopentut*.
- Although targeted toward web applications, the Open Web Application Security Project (OWASP, <http://www.owasp.org>) has plenty of good advice for all types of applications.
- Even if you don't want to read warnings from the Computer Emergency Response Team (CERT, <http://www.cert.org>) or SecurityFocus (<http://www.securityfocus.com/>), reading some of their advisories about perl interpreters or programs is often instructive.

Further Reading, continued

- The documentation for DBI has more information about placeholders and bind parameters, as well as `TaintIn` and `TaintOut`. *Programming the Perl DBI* by Tim Bunce and Alligator Descartes is another good source, although it does not cover the newer taint features of DBI.

Profiling

Profiling is better than benchmarking

- Benchmarking is often pre-mature
- Profiling shows you the performance of your program
 - * speed
 - * memory
 - * whatever
- See what's taking up your resources
- Focus your efforts in the right places

The basics of profiling

- Profiling counts something
- All the code runs through a central point, a recorder
- While recording, the program is slower
- At the end I get a report
- Use the report to make a decision

A recursive subroutine

- A recursive subroutine runs itself many, many times.
- Everyone seems to like to use the factorial implementation, so I'll use that:

```
sub factorial
{
  return unless int( $_[0] ) == $_[0];
  return 1 if $_[0] == 1;
  return $_[0] * factorial( $_[0] - 1 );
}

print factorial($ARGV[0]), "\n";
```

Calling a Profiler

- Invoke a custom debugger with `-d`
`perl -d:MyDebugger program.pl`
- `MyDebugger` needs to be in the `Devel::*` namespace
- Uses special DB hooks for each statement
- Find several on CPAN
 - * `Devel::DProf`
 - * `Devel::NYTProf`
 - * `Devel::SmallProf`
 - * `Devel::LineProfiler`

Recursion profile

- Runs several statements for each call

```
% perl -d:SmallProf factorial.pl 170
```

- Creates a file named *smallprof.out*

```
===== SmallProf version 1.15 =====  
                Profile of factorial.pl  
  
Page 1  
=====
```

count	wall	tm	cpu	time	line
0	0.000000	0.000000			1:#!/usr/bin/perl
0	0.000000	0.000000			2:
170	0.000000	0.000000			3:sub factorial {
170	0.001451	0.000000			4: return unless int(\$_
					[0]) == \$_[0];
170	0.004367	0.000000			5: return 1 if \$_[0] == 1;
169	0.004371	0.000000			6: return \$_[0] *
					factorial(\$_[0]-1);
0	0.000000	0.000000			7: }

Iteration, not recursion

- Perl 5 doesn't optimize for tail recursion, so it can't optimize recursion.
- I shouldn't run more statements than I need.
- Better algorithms beat anything else for efficiency.
- With iteration, I don't need to create more levels in the call stack.

```
sub factorial {  
    return unless int( $_[0] ) == $_[0];  
  
    my $product = 1;  
  
    foreach ( 1 .. $_[0] ) { $product *= $_ }  
  
    $product;  
}  
  
print factorial( $ARGV[0] ), "\n";
```

Iteration profile

- Now I don't call needless statements

```
===== SmallProf version 2.02=====
          Profile of factorial-iterate.pl
```

Page 1

```
=====
```

count	wall tm	cpu time	line
0	0.000000	0.000000	1:#!/usr/bin/perl
0	0.000000	0.000000	2:
0	0.000000	0.000000	3:sub factorial {
1	0.000001	0.000000	4: return unless
			int(\$_[0]) == \$_[0];
1	0.000000	0.000000	5: my \$f = 1;
170	0.000111	0.000000	6: foreach (2 ..
			\$_[0]) { \$f *= \$_ };
1	0.000009	0.000000	7: \$f;
0	0.000000	0.000000	8: }

Really big numbers

- Now I want have a program that takes a long time.
- My perl tops out at 170!, then returns `inf`.
- The `bignum` package comes with Perl 5.8, and I can use really big numbers

```
use bignum; get really large numbers
```

```
sub factorial {  
    return unless int( $_[0] ) == $_[0];  
  
    my $product = 1;  
  
    foreach ( 1 .. $_[0] ) { $product *= $_ }  
  
    $product;  
}  
  
print factorial( $ARGV[0] ), "\n";
```

Memoize

- This still isn't good because it's one shot.
- By *memoizing*, I remember previous computations for future speed-ups:

```
my @Memo      = (1);

sub factorial {
    my $number = shift;

    return unless int( $number ) == $number;
    return $Memo[$number] if $Memo[$number];

    foreach ( @Memo .. $number ) {
        $Memo[$_] = $Memo[$_ - 1] * $_;
    }

    $Memo[ $number ];
}
```

Memoize, continued

```
while(1) {  
    print 'Enter a number> '  
    chomp( my $number = <STDIN> );  
    exit unless defined $number;  
    print factorial( $number ), "\n";  
}
```

What happened?

- One shot is not so bad
- I redo a lot of work if I call `factorial` many times.
- Memoizing is faster each time, but takes more memory.

Modern profiling with NYTProf

- `Devel::NYTProf` is a `Devel::DProf` replacement written by Adam Kaplan at the *New York Times*, and now maintained by Tim Bunce.
- `Devel::NYTProf` is both a statement profiler and a subroutine profiler, so I get more information out of it.
- I invoke it in the same way:

```
% perl -d:NYTProf journals
```
- I can get different sets of reports:

```
% nytprofhtml  
% nytprofcvs
```
- A demonstration is the best way to show off NYTProf.

Record DBI queries

- Create a routine through which all queries flow:

```
package My::Database;
```

```
my %Queries;
```

```
sub simple_query
```

```
{
```

```
  my( $self, @args ) = @_;
```

```
  my $sql_statement = shift @args;
```

```
  $Queries{$sql_statement}++;
```

Profiling hook

```
  my $sth = $self->dbh->prepare($sql_statement);
```

```
  unless( ref $sth ) { warn $@; return }
```

```
  my $rc = $sth->execute( @args );
```

```
  wantarray ? ( $sth, $rc ) : $rc;
```

```
}
```

Database optimization

- Often, the database bits are the slowest part of my program
- Most of the work is not in my program because it's in the database server
- My program waits for the database response
- I usually talk to the database more than I need to
 - * Repeated `SELECT`s for the same, unchanging data
- My queries are too slow
 - * Optimize the slowest, most frequent ones

Profiling DBI Statements

- Uses the `DBI_PROFILE` environment variable
- Using `!Statement` orders by the query text

```
$ env DBI_PROFILE='!Statement' perl dbi-profile.pl
```

```
DBI::Profile: 109.671362s 99.70% (1986 calls)  
dbi-profile.pl @ 2006-10-10 02:18:40
```

```
'CREATE TABLE names ( id INTEGER, name CHAR(64)  
)' => 0.004258s
```

```
'DROP TABLE names' => 0.008017s
```

```
'INSERT INTO names VALUES ( ?, ? )' =>  
3.229462s / 1002 = 0.0032223s avg (first  
0.001767s, min 0.000037s, max 0.108636s)
```

```
'SELECT name FROM names WHERE id = 1' =>  
1.204614s / 18 = 0.066923s avg (first  
0.012831s, min 0.010301s, max 0.274951s)
```

```
'SELECT name FROM names WHERE id = 10' =>  
1.118565s / 9 = 0.124285s avg (first
```

Profiling DBI methods

- Set DBI_PROFILE to !MethodName

```
$ env DBI_PROFILE='!MethodName' perl dbi-profile2.pl
```

```
DBI::Profile: 2.168271s 72.28% (1015 calls)
dbi-profile2.pl @ 2006-10-10 02:37:16
```

```
'DESTROY' =>
```

```
0.000141s / 2 = 0.000070s avg (first
0.000040s, min 0.000040s, max 0.000101s)
```

```
'FETCH' => 0.000001s
```

```
'STORE' =>
```

```
0.000067s / 5 = 0.000013s avg (first
0.000022s, min 0.000006s, max 0.000022s)
```

```
'do' =>
```

```
0.010498s / 2 = 0.005249s avg (first
0.006602s, min 0.003896s, max 0.006602s)
```

```
'execute' =>
```

```
2.155318s / 1000 = 0.002155s avg (first
0.002481s, min 0.001777s, max 0.007023s)
```

```
'prepare' => 0.001570s
```

Profiling test suites

- I can profile my test suite to see how much code it tests
- I want to test all code, but then there is reality
- Where should I spend my testing time to get maximum benefit?
- The `Devel::Cover` module does this for me

```
% cover -delete clear previous report
```

```
% HARNESS_PERL_SWITCHES=-MDevel::Cover make  
test
```

```
% ./Build testcover for Module::Build
```

```
% cover generates report from data  
Reading database from Dev/HTTP/Size/cover_db
```

- Sends text report to standard output
- Also creates an HTML report

Devel::Cover HTML report

Database: /Users/brian/Dev/HTTP/Size/cover_db

file	stmt	branch	cond	sub	pod	time	total
blib/lib/HTTP/Size.pm	95.5	70.0	69.2	100.0	100.0	100.0	88.4
Total	95.5	70.0	69.2	100.0	100.0	100.0	88.4

Done Disabled

Devel::Cover detail

File Coverage: blib/lib/HTTP/Size.pm

file:///localhost

Listen Write Read Perl Gmail TPR Submissions brian Publishing

Disable CSS Forms Images Information Miscellaneous Outline Resize

File Coverage

File: blib/lib/HTTP/Size.pm
Coverage: 88.4%

line	stmt	branch	cond	sub	pod	time	
1							# \$Id: Size.pm,v 1.11 2006/06/0
2							package HTTP::Size;
3	3 3 3			3		62 13 70	use strict;
4							

Done Disabled

Further reading

- The *perldebguts* documentation explains custom debuggers
- “Creating a Perl Debugger” (<http://www.ddj.com/184404522>) and “Profiling in Perl” (<http://www.ddj.com/184404580>) by brian d foy
- “The Perl Profiler”, Chapter 20 of *Programming Perl, Third Edition*
- “Profiling Perl” (<http://www.perl.com/lpt/a/850>) by Simon Cozens
- “Debugging and Profiling mod_perl Applications” (http://www.perl.com/pub/a/2006/02/09/debug_mod_perl.html) by Frank Wiles
- “Speeding up Your Perl Programs” (<http://www.stonehenge.com/merlyn/UnixReview/col49.html>) and “Profiling in Template Toolkit via Overriding” (<http://www.stonehenge.com/merlyn/LinuxMag/col75.html>) by Randal Schwartz

Conclusion

Main points

- Profile your application before you try to improve it
- Be very careful and sceptical with benchmarks
- Make your program flexible through configuration
- Use Log4perl to watch program progress, report errors, or debug
- Use lightweight persistence when you don't need a full database server

More information

- The Perl Review: *www.theperlreview.com*
- Feel free to email me: *brian.d.foy@gmail.com*
- See all of my talks, *<http://www.pair.com/~comdog/>*
- Also on SlideShare, *http://www.slideshare.net/brian_d_foy*
- Often on Perlcast, *<http://www.perlcast.com>*

Questions