Perl Gotchas

Dieser Vortrag beschäftigt sich mit typischen Problemen beim Umgang mit Perl und wendet sich sowohl an Anfänger wie auch an erfahrene Entwickler.

Autor

Steffen Ullrich <Steffen_Ullrich@genua.de>

- Autor von `Net::SIP`, `Devel::TrackObjects`
- Maintainer von `IO::Socket::SSL`
- CPAN: SULLR
- seit 2001 bei GeNUA mbH, 100% Perl Job

Abstract

- Dieser Vortrag beschäftigt sich mit typischen Problemen beim Umgang mit Perl.
- Alle hier geschilderten Probleme sind vom Autor wiederholt in der Realität beobachtet worden oder er ist gar selber darüber gestolpert.
- Insbesondere geht es Probleme
    - bei denen sich die Sprache anders als aus Erfahrung (C, awk, sh, ...) gewohnt verhält
    - Probleme spezifisch für den Einsatz im Sicherheitsumfeld
    - weitere Fettnäpfchen
- Weiterhin gibt es einige Hinweise für besseren Code, die sich bei uns in der Firma bewährt haben.

Loop Variable in `for` ist Alias

- Loop Variable in `for` Loop ist Alias und sollte deshalb nicht verändert werden (außer man weiß was man tut)
- Verhalten als Alias ist unerwartet:

```
1 # Aufgabe: aus dieser Liste Hash A => 10, ... bilden
2 my @in = ( 'A=10','B=11','C=12' );
3 my %hash;
4 for my $string (@in) {
5   $hash{$string} = $1 if $string =~s/=(.*)//;
6 }
7 # hash OK, aber @in: 'A','B','C'
```

- Verhalten als Alias wird genutzt:

```
1 # Aufgabe: Punkt mit Komma ersetzen
2 my %data = ( a => 10.41, b => 11.3, c => 17.49 );
3 for my $num (values %data) {
4   $num = sprintf("%.02f", $num); # zwei Nachkommastellen
5   $num =~s{\.}{,};               # Komma statt Punkt
6 }
7 # data: ( a => '10,41', b => '11,30', c => '17,49' )
```

Socket::inet_aton löst Namen auf

- libc:

  The routines inet_aton() ... interpret character strings representing numbers expressed in the Internet standard '.' notation.

- perl

```
1 use Socket;
2 inet_aton( '10.0.4.3' ); # OK
3 # blockierender DNS Lookup !!
4 inet_aton( 'www.example.com' );
```

Block trotz nonblocking socket

- Lesen von $! bei Fehler benutzt libc strerror.
- Zumindest bei OpenBSD mit NLS führt das zu (blockierendem) Zugriff auf Festplatte:

```
1 my $sock = IO::Socket::INET->new...;
2 $sock->blocking(0);
3 if ( ! defined sysread( $sock,... ) ) {
4   if ( $! == EAGAIN )...
```

- Bei Applikationen mit viel asynchroner IO ist damit die Geschwindigkeit durch die Festplatte beschränkt :)
- Lösung: perl ohne NLS compilieren

Missbrauch von Funktionsprototypen I

- Doku: siehe perldoc perlsub, Überschrift "Prototypes"
- Kamen bei uns irgendwann mal im Mode, um den Code besser zu dokumentieren
- Aber: Prototypen wirken nicht auf Methoden!!!

```
1 package Test;
2 sub new { return bless {},shift };
3 sub test($\@) {
4     my ($self,$list) = @_;
5     warn "$self: @$list\n";
6 }
7 package main;
8 my $t = Test->new;
9 my @a = ( 1,2,3,4 );
```

- als Funktion aufgerufen

```
10 # &test($t,\@a) -> .. 1 2 3 4
11 Test::test( $t,@a );
```

- als Methode aufgerufen

```
12 # &test($t,@a)
13 # -> Can't use string ("1") as an ARRAY ref
14 $t->test( @a );
```

Missbrauch von Funktionsprototypen II

- Prototypen können gefährlich sein!

```
1 sub babel($;$) {
2   my $string = shift;
3   my $lang = shift || 'DE';
4   warn "babel( '$string','$lang' )\n";
5 }
```

- Oops!

```
6 sub babel_wrapper_1 { return babel(@_) }
7 babel_wrapper_1( 'test' );       # -> '1','DE'
8 babel_wrapper_1( 'test','EN' );  # -> '2','DE'
```

- perlsub:

  The prototype affects only interpretation of new-style calls to the function, where new-style is defined as not using the "&" character.

```
 9 sub babel_wrapper_2 { return &babel(@_) }
10 babel_wrapper_2( 'test' );       # -> 'test','DE'
11 babel_wrapper_2( 'test','EN' );  # -> 'test','EN'
```

Aber wozu dann überhaupt Prototypen?

- Jedenfalls nicht zur Dokumentation der Parameter
- perldoc perlsub:

  ...the intent of this feature is primarily to let you define subroutines that work like built-in functions..

- z.B. neue Funktion ngrep analog zu grep:

```
 1 sub ngrep(&$@) {
 2     my ($sub,$n) = (shift,shift);
 3     my @result;
 4     for (@_) {
 5         push @result,$_ if $sub->();
 6         last if @result == $n;
 7     }
 8     return @result;
 9 }
10 my @list = ( 1..20 );
11 my @l3 = ngrep { $_ > 10 } 3,@list;
12 warn "@l3\n"; # 11 12 13
```

our vs my vs vars

- my

  Deklariert lokale Variable, die innerhalb des Scopes existiert.

- vars

  Macht eine globale Packagevariable ohne Meckern von 'use strict' im Package sichtbar.

- our

  Macht eine globale Packagevariable ohne Meckern von 'use strict' im Scope sichtbar.

  Fast das gleiche wie `vars`, aber ...

Scope ist Datei oder Block, nicht Package

```
 1 use strict;

 2 package A;
 3 my $my_var1 = 'A.my1';
 4 our $our_var1 = 'A.our1'; # geht auch mit vars

 5 {
 6   package B;
 7   my $my_var2 = 'B.my2';
 8   our $our_var2 = 'B.our2';
 9 }

10 package C;
11 print $my_var1;          # -> A.my1
12 print $our_var1;         # -> A.our1
13 ## print $my_var2;       # Syntaxfehler
14 ## print $our_var2;      # Syntaxfehler
15 print $B::my_var2;       # -> undef
16 print $B::our_var2;      # -> B.our2
```

Achtung bei @ISA und our!

- Dadurch das ein Package kein Scope ist kann es zu folgendem Fettnäpfchen kommen:

```
1 use strict;
2 package A;
3 package B;
4 our @ISA = 'A';
5 package C;
6 our @ISA = 'A';
7 package D;
8 @ISA = 'B';
```

- OK

```
 9 package main;
10 print "A::ISA=@A::ISA\n"; # leer
11 print "B::ISA=@B::ISA\n"; # B::ISA=A
```

- Oops, 'our' in Zeile 8 vergessen:

```
12 print "C::ISA=@C::ISA\n"; # C::ISA=B !!!
13 print "D::ISA=@D::ISA\n"; # leer !!!!
```

Achtung beim Mixen `use vars` und `our`

- typischer Code bei Migration von `use vars` nach `our`:

```
1 package A;
2 package B;
3 our @ISA = 'A';

4 package C;
5 use vars '@ISA';
6 @ISA = 'B';
```

- Oops: `use vars` macht globale Variable im **Package** sichtbar, `our` dagegen im **Scope**

```
 7 package main;
 8 print "A::ISA=@A::ISA\n"; # leer
 9 print "B::ISA=@B::ISA\n"; # B::ISA=B !!!
10 print "C::ISA=@C::ISA\n"; # leer !!!
```

.. und local?

local weist einer globalen Variable einen neuen Speicherplatz zu, den alten bekommt sie am Ende des Scopes wieder

- my, our: innerhalb des Scopes
- local: bis zum Ende des Scopes
- In sort werden $a und $b local-isiert
- In for, grep, map wird $_ local-isiert

.. und local? Beispiel

```
 1 sub show { print $_,"\n" }
 2 $_ = 10; show();              # -> 10

 3 { local $_ = 12; show() }     # -> 12
 4 show();                       # -> 10

 5 show() for ( 1,2,3,4 );       # for: macht $_ local: 1 2 3 4
 6 show();                       # -> 10

 7 { $_ = 100; show() }          # -> 100, kein local!
 8 show();                       # -> 100

 9 sub plus_eins { $_++ }
10 plus_eins();
11 show();                       # -> 101
```

use strict/warnings wirken auf Scope

- oft gesehen, aber unsinnig, da OldHack nicht im Scope:

```
1 no strict;
2 use OldHack;
3 use strict;
```

- strict und warnings wirken nur auf aktuellen Scope, d.h. Datei oder Block (und nicht Package!) und können auch verschachtelt sein:

```
1 use strict;
2 ...
3 no strict;
4 ...old code...
5 use strict;
```

```
1 # Besser:
2 use strict;
3 {
4   no strict;
5   ...old code...
6 }
```

Was macht eigentlich 'use'?

- perldoc -f use
- ruft require auf um das Module zu laden (es sei denn es ist schon geladen, was aus %INC ersichtlich ist)
- schaut, ob das Module eine Klassenmethode `import` hat und ruft dann diese auf.

  Zum Beispiel implementiert das Module Exporter eine solche Methode um Funktionen in den aufrufenden Namespace zu exportieren und stellt diese Methode anderen Modulen zur Verfügung:

  ```
  package Test;
  use base 'Exporter;
  # Test->import ruft Exporter::import auf
  ```

- daraus folgt: der Methodenname `import` ist für einen speziellen Zweck reserviert und sollte nur für diesen verwendet werden, das gilt auch für `unimport` welches bei '`no Module`' aufgerufen wird
- gibt noch weitere spezielle Methoden und Funktionen, aber diese sind alle großgeschrieben (BEGIN, CHECK, INIT, END, CLONE, CLONE_SKIP, DESTROY, siehe perlmod)

Fettnäpfchen Kontext

- einige Funktionen haben unterschiedlichen Verhalten, je nach Kontext:

```
1 my $size = @x;                              # Anzahl Element
2 my ($first) = @x;                           # erstes Element
3 my $time = localtime();                     # Zeit als String
4 my ($sec,$min,$hour,...) = localtime(); # Zeit als Liste
```

- man muss berücksichtigen für welchen Kontext eine Funktion gedacht ist

```
1 sub div {
2   my ($dividend,$divisor) = @_;
3   return if $divisor == 0;
4   return $dividend/$divisor;
5 }
6 my %d10 = ( -1 => div(10,-1), 0 => div(10,0), 1 => div(10,1) );
7 # -> ( -1 => -10, 0 => 1, 10 => undef )
```

- man muss sich im Klaren sein, in welchem Kontext eine Funktion benutzt wird

```
1 sub my_rand { # liefert N random Zahlen
2   my $n = shift || 1;
3   return map { rand(2**16) } (1..$n);
4 }

5 $x = my_rand(1); # !!! liefert immer '1'
6 ($y) = my_rand(1); # OK
```

im Zweifelsfalle Klammern setzen

- und man sollte genügend Zweifel haben

```
open F,$file || die $!;
```

- ist nicht das gleiche wie

```
open( F,$file ) || die $!;
```

- sondern das gleiche wie

```
open F, ( $file || die $! );
```

Spezielle Variablen

- siehe perldoc perlvar
- neben offensichtlichen speziellen wie $; $, $/ $^X... gibt es auch $a und $b:

```
use strict;
my $A = 10;
my $B = $a + 1; # kein Syntaxfehler!!
```

- aber wir benutzen doch sowieso alle sprechende Variablen, die länger als ein Zeichen sind, oder?
- matcht diese Regex?

```
my $string = "bla\nfasel";
$string =~m{bla$\nfasel}m;
```

- Nein, da `$\` der Output Record Separator ist und bei einem Default von 'undef' die Regex zu folgendem wird:

```
$string =~m{blanfasel}m;
```

Namenskonflikte mit builtins

- manchmal sind eigene Methoden so genannt wie builtins, zB send, close. Das führt zu folgenden Problemen:

```
  1 package Test;
  2 sub close {}
  3 sub close_file { close(@_) }
  4 sub close_test { shift->close(@_) }

$perl -cw Test.pm
Ambiguous call resolved as CORE::close() ... line 3
```

- Die beste Lösung ist die Sache klarzustellen, d.h. die Methode umzubenennen.
- Die zweitbeste Möglichkeit ist klarzustellen, das das lokal deklarierte nur als Methode aufgerufen wird:

```
- sub close {}
+ sub close:method {}
```

0, "0", TRUE, FALSE,... - I

- Interpretation von Strings in numerischem Kontext

```
1 "0" == 0       # numerischer Vergleich -> TRUE
2 "0E0" == 0     # numerischer Vergleich -> TRUE
3 "1" == 1       # numerischer Vergleich -> TRUE
4 "1E1" == 10    # numerischer Vergleich -> TRUE
5 "1C1" == 1     # Nummer am Anfang (+warnings) -> TRUE
6 "5 1 2" == 5   # Nummer am Anfang (+warnings) -> TRUE
```

- und in boolean Kontext

```
1 !"A"           # Stringlänge>0 -> FALSE
2 !"0A"          # Stringlänge>0 -> FALSE
3 !"0"           # looks_like_number -> !(int("0")) -> TRUE
4 !"0E0"         # ! looks_like_number -> FALSE (Stringlänge)
```

0, "0", TRUE, FALSE,... - II

- typischer Fehler:

```
 1 while (<DATA>) {
 2     s/#.*//;       # Kommentar weg
 3     s/\s+$//;      # Trailing space
 4     next if ! $_;  # Leerzeilen weg
 5     print $_,"\n"; # Rest ausgeben
 6 }
 7 __DATA__
 8 Wir haben  # Zeile 1
 9            # Zeile 2
10 0          # Zeile 3
11 Pferde     # Zeile 4
```

- ergibt:

```
Wir haben
Pferde
```

- Lösung ist:

```
-    next if ! $_;
+    next if ! length $_; # length, aber nicht defined
```

next außerhalb von Loops

- Man kann ein next, redo oder last auch außerhalb eines (offensichtlichen) Loops aufrufen. Das sucht sich dann schon den nächsten umliegenden Loop, und sei dieser auch ganz weit weg.
- Typischer Fall nach Refactoring von Code, d.h. große Codeteile in kleinere Funktionen umschreiben:

```
1 my $i = 0;
2 while ($i<10) {
3     show( $i );
4     $i++;
5 }
6 sub show {
7     print "@_ ";
8     next;
9 }
```

- `0 0 0 0 0 0 0 0 0 0 0 0 ...`
- perl -cw zeigt kein Problem. Nur wenn man use warnings benutzt sieht man eine Warning, aber erst zur Laufzeit

|| statt //

- oft wird || benutzt aber nicht gemeint:

```
1 # cut -F..
2 getopts( 'F' ); # delimiter Parameter -F
3 $opt_F ||= "\t";
---
cut -F 0 ...  - cut nicht an '0' sondern "\t"
```

- i.A. ist in solchen Fällen `defined` gemeint

```
- $opt_F ||= "\t";
+ $opt_F = "\t" if ! defined $opt_F;
+ $opt_F //= "\t"; # ab perl5.9
```

- typisch auch für Fileno

```
# die() auch bei valider fn == 0
- fileno($fd) || die;
+ defined( fileno($fd) ) || die;
+ fileno($fd) // die; # ab perl5.9
```

AutoLoader vs. chroot

- AutoLoader und auch andere perl Module laden Code dynamisch nach, was problematisch ist in chroot Umgebungen, die nicht eine komplette perl Installation beinhalten:

```
1 use Carp 'croak';
2 chdir( '/cage' );
3 chroot( '/cage' ) || die $!;
4 croak 'bad';
---
Can't locate Carp/Heavy.pm in @INC
```

- auch bei Net::DNS, MIME::Decode, POSIX etc

- Workarounds je nach Art des Modules, mal das fehlende Modul manuell vorab laden, mal das *.al File des Autoloaders manuell requiren, mal mit eval versuchen....

```
require Carp::Heavy;     # Carp
require 'auto/exit.al';  # POSIX
eval { freeze() };       # Storable
```

Regex/Unicode vs. chroot

- Regexen laden evtl. dynamisch Code nach, wenn Unicode benutzt wird

```
1 chdir( '/tmp' );
2 chroot( '/tmp' ) || die $!;
3 print shift(@ARGV) =~ m/(\p{IsPrint}+)/;
---
Can't locate utf8.pm in @INC
```

- wenn man Pech hat hängt das von bestimmtem Input ab

```
1 # html decode: &#48; -> '0'
2 chdir( '/tmp' );
3 chroot( '/tmp' ) || die $!;
4 $_ = shift;
5 s{ &\#(\d+); }{ chr($1) }xeg;
6 print m{(\w+)};
---
test.pl 'bla&#48;' -> 'bla0'
test.pl 'bla&#480;' -> Can't locate utf8.pm in @INC...line 6
```

- Workarounds: utf8 explizit vorab laden (dann aber evtl. Problem mit bestimmten Codepages) oder Unicode vermeiden, indem man keine Unicode-Charsets benutzt und auch use bytes einschaltet.

- siehe auch mein Vortrag "Firewall in Perl" von Perl-Workshop 2007

Regex - Fettnäpfchen

- Perl Regex sind 'left most longest', d.h. der längste links stehende Match gewinnt

```
1 '123456789' =~ m/\d+7|\d+9/; print $&,"\n";
2 '123456789' =~ m/\d+9|\d+7/; print $&,"\n";
---
1234567
123456789
```

- Regex aus Userland sollten nicht direkt in die Regex gestopft werden, es sein denn man will es wirklich als Regex benutzen

```
my $user_rx = ...
my $rx_substr = qr/before\Q$user_rx\Eafter/; # als Text
my $user_rx_checked = eval { qr/$user_rx/ }; # überprüfen
```

- Regex sind unberechenbar und unkontrollierbar bzgl. Laufzeit und Speicherverbrauch (Beispiele folgen)

Probleme mit Regex, Beispiel 1

- Aufgabe: Erhöhe Spamlevel, wenn mindestens 3 Empfänger in einer To-Zeile (Addr mit Komma getrennt)

```
# To: bla@fasel.com, me@example.org,...
if ( $header =~m{ ^To: ([^,]+,){3,} }xim ) {
   ...
}
```

- Führte zu Out of memory bei 100en von Empfaengern, da versucht wird alle zu matchen.
- ein `{3}` statt `{3,}` wäre besser gewesen

Probleme mit Regex, Beispiel 2

- Aufgabe: Matche HTML-Tag mit Attributen.

```
# <a href=... title="...".. >
1 m{ <\w+  #Tag
2   (
3     \s+\w+   #Attribute Name
4     ( \s*=\s* (
5       "[^"]*"  # quoted Value
6       |[^"\s]+ # unquoted
7     )?)? # attr|attr=|attr=value
8   )*
9 \s*> }xs;
```

- Bei <a href=..100e spaces.. wird es exponentiell langsamer, da Regex Engine sich nicht zwischen Matches in Zeile 3 und Zeile 4 entscheiden kann.
- Lösung bestand daran nicht ganz so viel Müll zu akzeptieren

Regex - was sind /m, /s ?

- '/m' sorgt bei mehrzeiligem Input dafür, das '^' und '$' auf den Anfang bzw das Ende einzelner Zeilen matchen dürfen, '\A' und '\Z' matchen weiterhin nur auf Anfang bzw. Ende des kompletten Inputs.
- '/s' sorgt dafür, das '.' innerhalb der Regex auch auf newlines matcht
- Beispiel

```
 1 $_ = <<'EOF';
 2 Ich mache nicht nur leere Versprechungen,
 3 ich halte mich auch daran.
 4 -- Edmund Stoiber
 5 EOF
 6
 7 check ( qr{^ich halte}m,            1 );
 8 check ( qr{^Ich mache},             1 );
 9 check ( qr{^Ich mache.*daran},      0 );
10 check ( qr{^Ich mache.*daran}s,     1 );
11 check ( qr{^ich halte.*Stoiber$}m,  0 );
12 check ( qr{^ich halte.*Stoiber$}ms, 1 );
```

Regex - was macht /o ?

- perlop manpage:

  o Compile pattern only once.

```
1 $_ = 'eins zwei drei';
2 my @pat = qw( eins zwei drei );
3 for my $pat (@pat) {
4     print "!O: $1\n" if m/($pat)/;  # eins zwei drei
5     print " O: $1\n" if m/($pat)/o; # eins eins eins
6 }
```

- genutzt um immer wieder kehrende Compilation einer Regex zu verhindern, wenn deren Inhalt abhängig von einer Variablen ist. Macht auf konstanten Regex keinen Sinn. Besser ist es vorcompilierte Regex zu benutzen:

```
 1 my @rx_pat = map { qr/$_/ } qw( eins zwei drei );
 2 for (
 3     "eins zwei drei",
 4     "drei zwei",
 5     "drei zwei eins",
 6 ) {
 7     # eins zwei drei   zwei drei   eins zwei drei
 8     for my $rx (@rx_pat) {
 9         print "$1\n" if m/($rx)/;
10     }
11 }
```

Regex - was macht /g ?

- matcht genau einmal im scalaren Kontext, und setzt dann immer wieder auf

```
$_ = "10-12-45-35";
print "z:$1\n" while ( m/(\d+)/g ); # z:10 z:12 z:45 z:35
```

- matcht alles im Arraykontext

```
my @zz = m/(\d+)/g; print "zz:@zz\n"; # zz:10 12 45 35
```

- Prima auch bei mehreren Klammern

```
$_ = "otto=14 --- berta=10";
my %h = m/(\w+)=(\d+)/g;
print Dumper(\%h); # otto => 14, berta => 10
```

- aber immer Kontext beachten!

```
# otto=14 berta=10
while ( m/(\w+)=(\d+)/g )               { print "$1=$2\n" }
# otto=14 otto=14 otto=14 otto=14....
while ( my ($k,$v) = m/(\w+)=(\d+)/g ) { print "$k=$v\n" }
```

Regex - was macht \s

- matcht whitespace

```
1 my $mail = <<EOM;
2 From: me
3 Subject:
4 To: you
5 EOM

6 $mail =~m{^Subject:\s*(.*)$}mi || die;
7 print "Subject = '$1'\n";
```

- `Subject = 'To: you'`
- whitespace ist auch newline !!!

Regex - Performance

- $&, $' und $` kosten Performance wenn irgendwo im Programm verwendet
- statt (...) wo immer möglich (?:...) verwenden (kein Zwischenspeichern ungenutzter Strings)
- nicht versuchen den String durch ständige Substitutes zu verkürzen sondern lieber m/\G../gc sowie pos() verwenden.

```
1 my $html = ...
2 if ( $html =~s{^<\w+}{} ) {         # Tag
3   if ( $html =~s{^\s+\w+}{} ) {     # Attributename
4     ...
--- Besser:
1 my $html = ...
2 if ( $html =~m{\G<\w+}gc ) {        # Tag
3   if ( $html =~m{\G\s+\w+}gc ) {    # Attributename
4     ...
```

- Regexp::Common bietet getestete, performante Regex
- viele weitere Optimierungen möglich, siehe Literaturhinweise

Regex übersichtlicher schreiben

- Regexp::Common benutzen wenn möglich
- /x Modifikator benutzen, um Kommentare einzubetten
- aus Einzelteilen zusammenbauen

```
1 my $scalar = qr{^\$\w+};
2 my $array  = qr{^\@\w+};
3 my $hash   = qr{^\%\w+};
4 my $var = qr{$scalar|$array|$hash};
```

Gefährlicher Code

- open mit nur 2 Argumenten

```
open( my $fh,$file );
open( my $fh,">$file" );
open( my $fh,"ci -l $rcsfile" );
open( my $fh,"|sendmail -f $from $to" );
```

- Besser mit >=3 Argumenten

```
open( my $fh,'<',$file );
open( my $fh,'>',$file );
open( my $fh,'|-','ci','-l',$rcsfile );
open( my $fh,'-|','sendmail','-f',$from,$to );
```

- 3+ Argument-Form ist lesbarer und sicherer, man weiß nie was wirklich in den übergebenen Argumenten steckt. Außerdem gibt es immer mehr Leute, die Leerzeichen oder Sonderzeichen in Dateinamen benutzen. z.B:

```
$file = '/tmp; rm -rf /';
$rcsfile = 'Vertrag(Entwurf).tex';
$from = '; rm -rf /';
```

- gleiches gilt für system und exec oder IPC::Open2::open2 etc

```
- system( "co -l $file" );
+ system( "co","-l",$file );
```

Ist der Input wie erwartet ?

- bei Input aus unbekannter Quelle sollte man nie annehmen, das er einem bestimmtem Format genügt, bei Input aus bekannter Quelle muss das nicht immer besser sein
- Gern wird versucht den Input zeilenweise mit <> bzw. getline einzulesen, weil es sollte sich ja **eigentlich** um MIME, HTTP etc handeln. Wenn dann die Zeile leider sehr lang ist kommt es zum out of memory.
- Es ist auch nicht immer klar, welche Annahmen die verwendeten Module über den Input treffen:
    - MIME::Parser liest den Input zeilenweise (was es allerdings auch explizit sagt).
    - Net::Cmd (und damit Net::POP3, Net::SMTP, Net::FTP ...) liest zwar blockweise aber solange, bis eine Zeile voll ist, d.h. kein Limit auf die Zeilenlänge.
- Insb. Regex werden in der Komplexität oft unterschätzt und werden nur mit erwartetem Input getestet (s.o.)
- Daher nur Annahmen über den Input treffen, die garantiert werden können oder mit dem Risiko leben

Wie schreibt man wartbaren Code

- genau so wie es in anderen Sprachen möglich ist unverständlichen Code zu schreiben, kann man in perl auch lesbaren Code schreiben
- in diesem Zusammenhang sei jedem "Perl Best Practices" von Damian Conway empfohlen
- damit der Code verständlich ist sollte man keine Konstrukte (bzw. Module) verwenden, die von den anderen Mitarbeitern nicht verstanden werden.
- das heißt allerdings nicht, das man auf forgeschrittene Feature der Sprache verzichten sollte (was in vielen Fällen schade wäre)
- sondern das die Mitarbeiter entsprechend geschult werden.
- Im folgenden werden einige Wege beschrieben, die helfen lesbareren und wartbaren Code zu schreiben

strict und warnings verwenden

- use strict fordert das Variablen.. deklariert werden, was den Code sicherer macht
- use warnings kann sehr nervig sein (insb. bzgl uninitialized Variablen) aber Erfahrungen bei uns zeigen, das es sehr viele potientielle Probleme zeigt, die es Wert sind gefixt zu werden.
- Für strict und warnings existieren haben wir für unseren Code Migrationsstrategien entwickelt, um Produktivsysteme mit relaxteren Einstellungen zu betreiben als Testumgebungen. In Standalone Tests produziert eine Warning einen Abbruch, im Testsystem eine Warnung und beim Kunden wird sie unterschlagen.

  ```
  use mystrict;
  use mywarnings;
  ```

Standardmodule verwenden

- perl kommt mit vielen Modulen, die bei jeder Installation dabei sind. Diese sind gut dokumentiert und getestet und sollten daher statt aufwendigen und fehlerträchtigen Eigentwicklungen bevorzugt werden. Dazu gehören insb.:
    - File::Temp bietet tempfile, tempdir...
    - File::Copy bietet komfortable copy und move
    - File::Path bietet mkpath und rmtree
    - File::Basename, File::Compare
- Getopt::Long statt Getopt::Std verwenden. Beides sind CORE Module aber Getopt::Long kann mehr (u.a. lange Argument, wie z.B. --help) und ist vor allem wesentlich lesbarer.
- Util::List::first statt grep:

```
- $found = 1 if grep { .. } @array;
+ $found = 1 if first { .. } @array;
```

Hashkeys schützen

- Hashes werden in perl für viele Zwecke benutzt
- Ein Problem ist, das die Keys von einem Hash nicht wie z.B. Variablennamen verifiziert werden, obwohl sie oft eine ähnliche Funktion haben:

```
1 my %hash = ( src => '10.0.3.4', dst => '10.0.5.6' );
2 ...
3 print $hash{source}; # undef
```

Hashkeys schützen - fields

- eine Möglichkeit ist bei Objekten fields zu benutzen:

```
 1 package Test;
 2 use fields qw( src dst socket );
 3 sub new {
 4   my ($class,%args) = @_;
 5   my $self = fields::new($class);
 6   %$self = %args; # die() wenn %args invalide Keys hat
 7   return $self;
 8 }
 9 sub get_src {
10   my Test $self = shift;
11   return $self->{source}; # Syntaxfehler zur Compilezeit
12 }
13 sub get_dst {
14   my $self = shift;
15   return $self->{destination}; # die() zur Runtime
16 }
```

- Vorteil von fields ist, das ein Teil bereits zur Compiletime verifiziert werden kann
- Allerdings kann man fields nur für Objekte verwenden und auch nicht bei mehreren Basisklassen

Hashkeys schützen - Hash::Util

- Bis perl5.8 konnte man für standalone Hashes fields::phash verwenden. Das ist depreceated und verschwindet in perl5.10 !
- Seit perl5.8 gehört Hash::Util zum perl Core welches Zugriff auf restricted Hashes bietet

```
 1 use Hash::Util qw( lock_keys lock_hash lock_value );
 2 my %hash = ( dst => '10.0.3.4', src => '10.0.4.5' );
 3 $hash{proto} = 'tcp';    # OK
 4 lock_keys( %hash );
 5 print $hash{source};     # die()
 6 $hash{proto} = 'udp';    # OK
 7 lock_value( %hash, 'proto' );
 8 $hash{proto} = 'tcp';    # die()
 9 lock_hash( %hash );
10 $hash{src} = '10.0.5.6'; # die()
```

- es existieren auch unlock_* Funktionen um den Zugriff wieder kontrolliert freizugeben
- das ganze funktioniert praktisch ohne Performanceeinbußen
- Wichtig: bei Objekten erst bless, dann lock_keys.

tief verschachtelte Hashzugriffe auflösen

- Tief verschachtelte Hashzugriffe sind nicht nur schwieriger zu lesen, sondern auch langsamer:

```
my $laddr = $relay->{fds}{s}{csock};
my $raddr = $relay->{fds}{s}{rsock};
my $in    = $relay->{fds}{s}{bytes_in};
```

- Besser:

```
my $fds   = $relay->{fds}{s};
my $laddr = $fds->{csock};
my $raddr = $fds->{rsock};
my $in    = $fds->{bytes_in};
```

Performance I

- Es ist einem oft nicht bewußt, was perl vorab optimiert und was nicht
- B::Deparse kann helfen einen Blick darauf zu werfen:

```
$ perl -MO=Deparse -e 'print 5*4'
print 20;
```

- constant kann genutzt werden um Konstanten zu deklarieren

```
$ perl -MO=Deparse -e 'use constant PI2 => 2*3.1415; print PI2'
use constant ('PI2', 6.283);
print 6.283;
```

- Regex ohne Variablen werden zur Compilezeit berechnet, mit Variablen nicht

Performance II

- anon Subs ohne Abhängigkeiten werden nur einmal gebunden, mit Abhängigkeiten werden sie jedesmal neu gebunden, was sehr teuer ist

```
 for ( 1,2,3 ) {
   my $i = $_;
-  my $sub = sub { print $i }; # immer wieder binden
-  $sub->();
+  my $sub = sub { print @_ }; # einmalig binden
+  $sub->($i);
 }
```

- usemymalloc ??

  in *hints/openbsd.sh*:

  ```
  # OpenBSD has a better malloc than perl...
  test "$usemymalloc" || usemymalloc='n'
  ```

  Drastischer Performanceeinbruch mit OpenBSD>=3.8 durch neue malloc Implementierung, die Speicher wieder freigibt (schlecht bei Objektorientiertheit).

Schon gewusst?

- perl kann Exception Handling. $@ kann Objekte transportieren:

```
eval {
   ...
   die myError::IO->new( file => ..., reason => $! );
   ...
};
$@->show_error() if $@ && UNIVERSAL::isa( $@,'myError' );
```

- Unterscheiden zwischen Strings und precompiled Regex in Argumenten

```
1 sub mygrep {
2   my ($pattern,$fd) = @_;
3   my $rx = UNIVERSAL::isa( $pattern,'Regexp' )
4     ? $pattern
5       : qr/\Q$pattern/i;
6   return grep { /$rx/ } <$fd>
7 }
```

Kann man über mehrere Ebenen exportieren?

- Schön wäre so was:

```
use ExportAll; # exportiert alles aus Utils,POSIX,..
```

- So nicht, aber man kann re-exportieren

```
1 package myUtils;
2 use Utils qw( debug );
3 use List::Util qw( first );
4 use base 'Exporter';
5 our @EXPORT = qw( debug first );
```

- Oder auch mit einem Sourcefilter:

```
# myUtils.pm
1 package myUtils;
2 use Filter::Macro; # CPAN
3 use Utils qw(debug);
4 use List::Util qw(first);
5 ...
---
# test.pl
use myUtils;
```

Crossreferenzen

- Das funktioniert so nicht:

```
 1 package Relay;
 2 my @allRelays; # Liste aller aktiven Relays;
 3 sub new {
 4   ...
 5   push @allRelays, $self;
 6   return $self
 7 }
 8 sub find_relay {
 9   my $sub = shift;
10   return first { $sub->($_) } @allRelays;
11 }
12 sub DESTROY {
13   # wird solange nicht aufgerufen, wie $self in @allRelays ist
14 }
```

- aber so geht es

```
-   push @allRelays, $self;
+   push @allRelays, $self;
+   Scalar::Util::weaken( $allRelays[-1] );
...
- sub DESTROY {
-   # wird solange nicht aufgerufen, wie $self in @allRelays ist
+ sub DESTROY {
+   my $self = shift;
+   @allRelays = grep { $_ != $self } @allRelays;
```

Literatur

- Conway 'Best Perl Practices'
- Conway, Poe, Chromatic: 'Perl Hacks'
- mein Vortrag "Firewall in Perl" auf dem Perl-Workshop 2007
- Friedl: 'Mastering Regular Expressions'
- perldoc perltrap

Dankeschön!

Fragen?