

Regex

Security

About me

Renée Bäcker

RENEEB @
Github
CPAN

Perl-Services.de since 2011

Regex

A regular expression (shortened as regex or regexp;(...)) is a sequence of characters that specifies a search pattern. Usually such patterns are used by string-searching algorithms for "find" or "find and replace" operations on strings, or for input validation. It is a technique developed in theoretical computer science and formal language theory.

https://en.wikipedia.org/wiki/Regular_expression

Security

Regex

Security

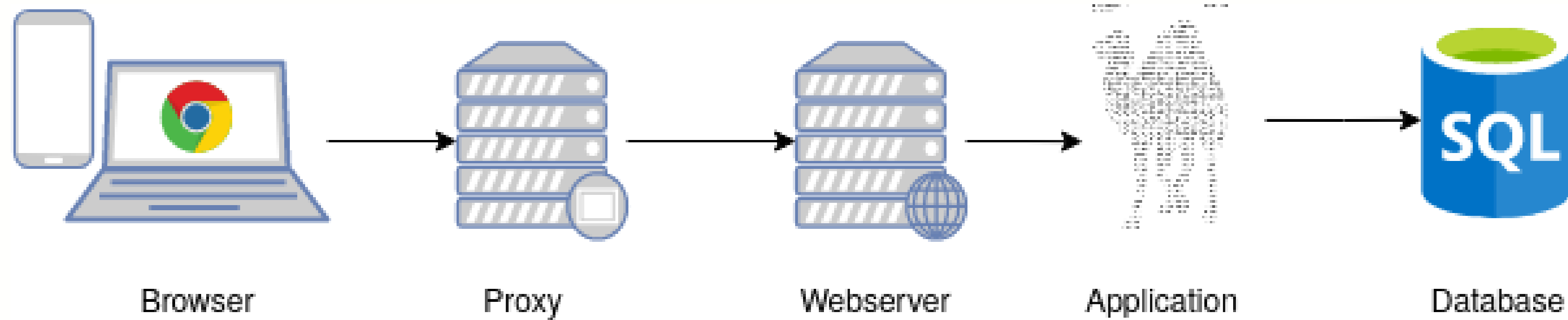
Security is freedom from, or resilience against, potential harm (or other unwanted coercive change) caused by others. Beneficiaries (technically referents) of security may be of persons and social groups, objects and institutions, ecosystems or any other entity or phenomenon vulnerable to unwanted change.

<https://en.wikipedia.org/wiki/Security>

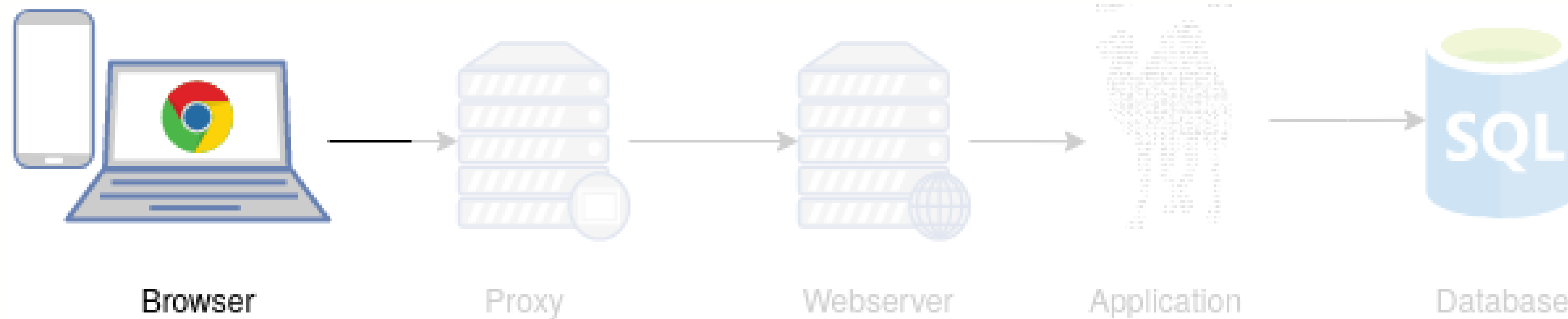
Regex

Security

Regexes are everywhere (e.g. Web)

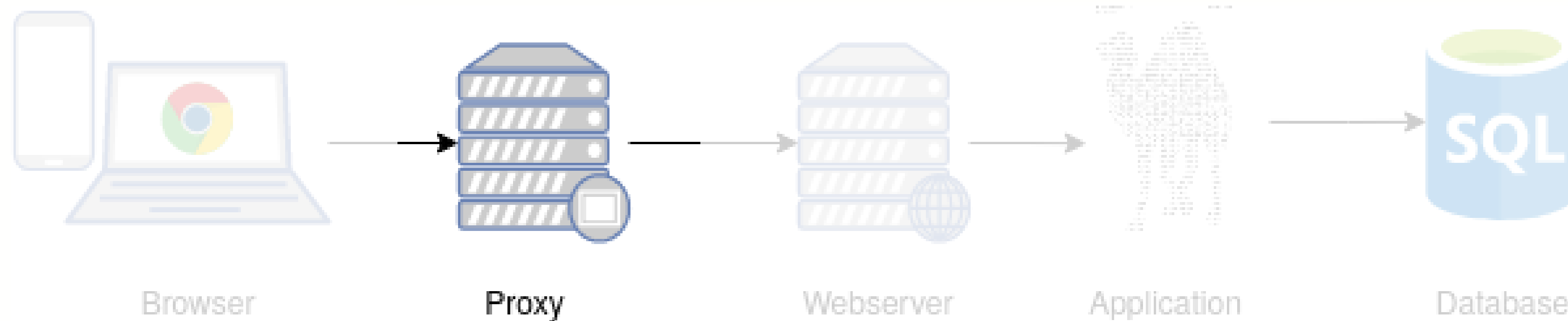


Regexes are everywhere (e.g. Web)



- Validate user input
- Browser detection
- Browser extensions
 - Chrome Regex Search

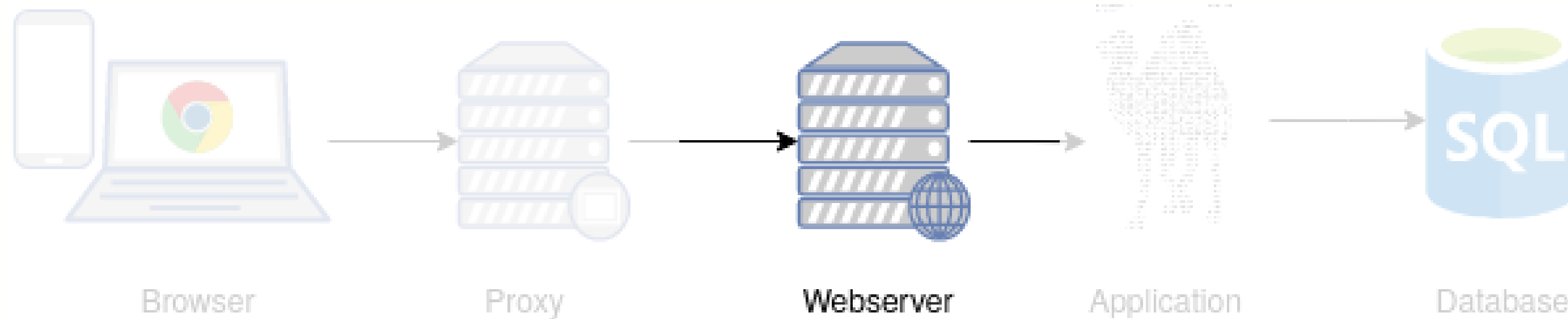
Regexes are everywhere (e.g. Web)



- Proxy redirects

```
proxy_redirect ~^(http://[^\:]+\):\d+(/.+)$ $1$2;  
proxy_redirect ~*/user/([^\:]+)/(.+)$  
               http://$1.example.com/$2;
```

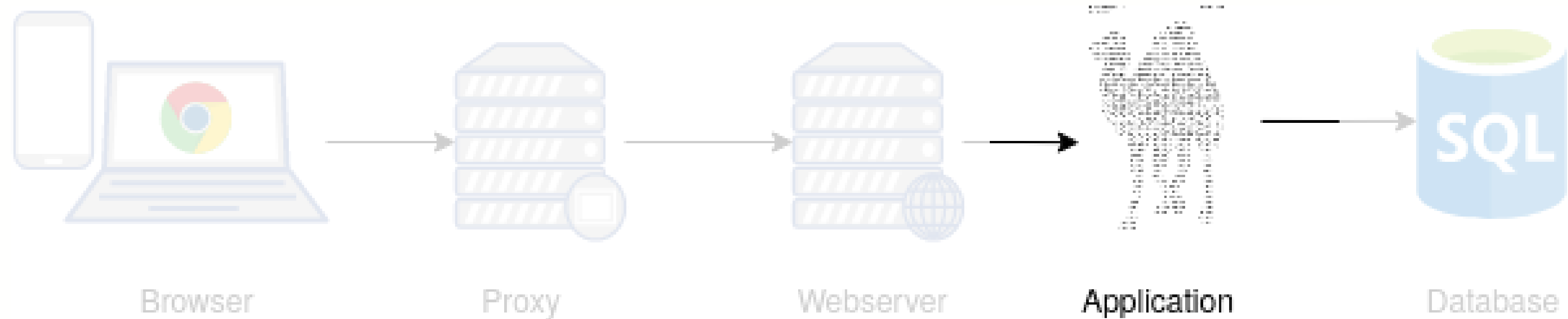

Regexes are everywhere (e.g. Web)



- Parsing logfiles
- Webserver configuration

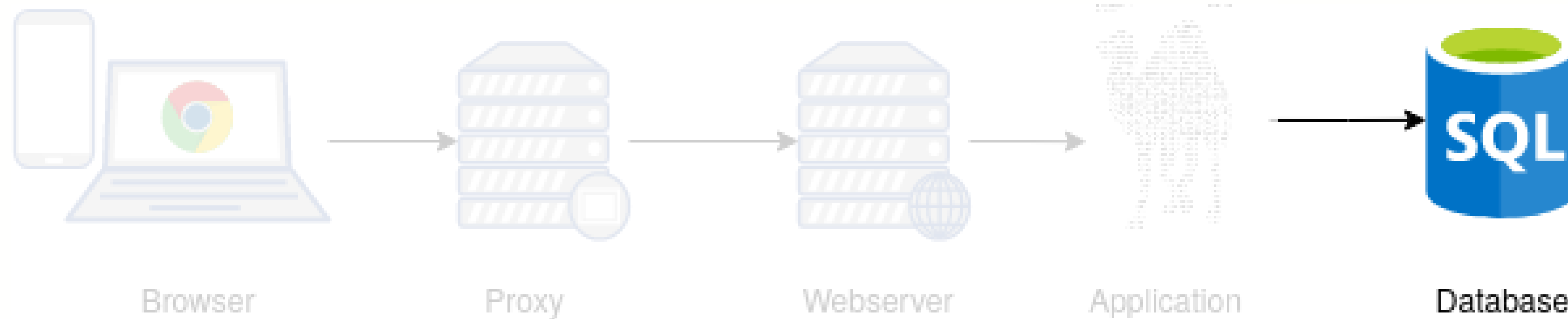
```
location ~* (?<begin>.*myapp)/(?<end>.+\.php)$ {  
...  
}
```

Regexes are everywhere (e.g. Web)



- We use Perl. We use Regexes everywhere ;-)

Regexes are everywhere (e.g. Web)



- Search for db entries

Regexes improve Security

Regexes improve Security

- Used to validate user input
- Parse logfiles to identify security threats (e.g. fail2ban)
- Used to identify persons/bots
 - Browser detection

```
if ( $username =~ m{\A[a-z]{8,}\z} ) {  
  
    ...  
  
}
```

Regexes improve Security

- Used to validate user input
- Parse logfiles to identify security threats (e.g. fail2ban)
- Used to identify persons/bots
 - Browser detection

Regexp::* modules

Data::Validate

Mojolicious::Validator

Regexes improve Security

- Used to validate user input
- Parse logfiles to identify security threats (e.g. fail2ban)
- Used to identify persons/bots
 - Browser detection

[Definition]

```
failregex = .* Login failed for host
```

```
ignoreregex =
```


Regexes improve Security

- Used to validate user input
- Parse logfiles to identify security threats (e.g. fail2ban)
- Used to identify persons/bots
 - Browser detection

```
if ( defined $robot_fragment ) {  
  
    # Examine what surrounds the fragment; that leads us to the  
    # version and the string (if we haven't explicitly set one).  
  
    if (  
        $self->{user_agent} =~ m{\s*                # Beginning whitespace  
        ([\w .:,\-\@\/]* # Words before fragment  
        $robot_fragment # Match the fragment  
        [\w .:,\-\@\/]*) # Words after fragment  
    }ix  
  
    ) {
```

From HTTP::BrowserDetect

Regexes reduce Security

Why?

We develop a debugger for Znuny / ((OTRS)) Community Edition / OTOBO / OFORK Postmaster Filters

Edit PostMaster Filter

* Name:

Setze Projekt

* Stop after match:

No

▼ Filter Condition (AND Condition)

Search header field:

To x

for value:

([w\.\++]+\@perl-services.de)

Negate:

☐

Search header field:

for value:

Negate:

☐

▼ Set Email Headers

Set email header:

X-OTRS-DynamicField-Project x

with value:

[**]

Set email header:

with value:

Why?

Users can upload those filters (regular expressions).

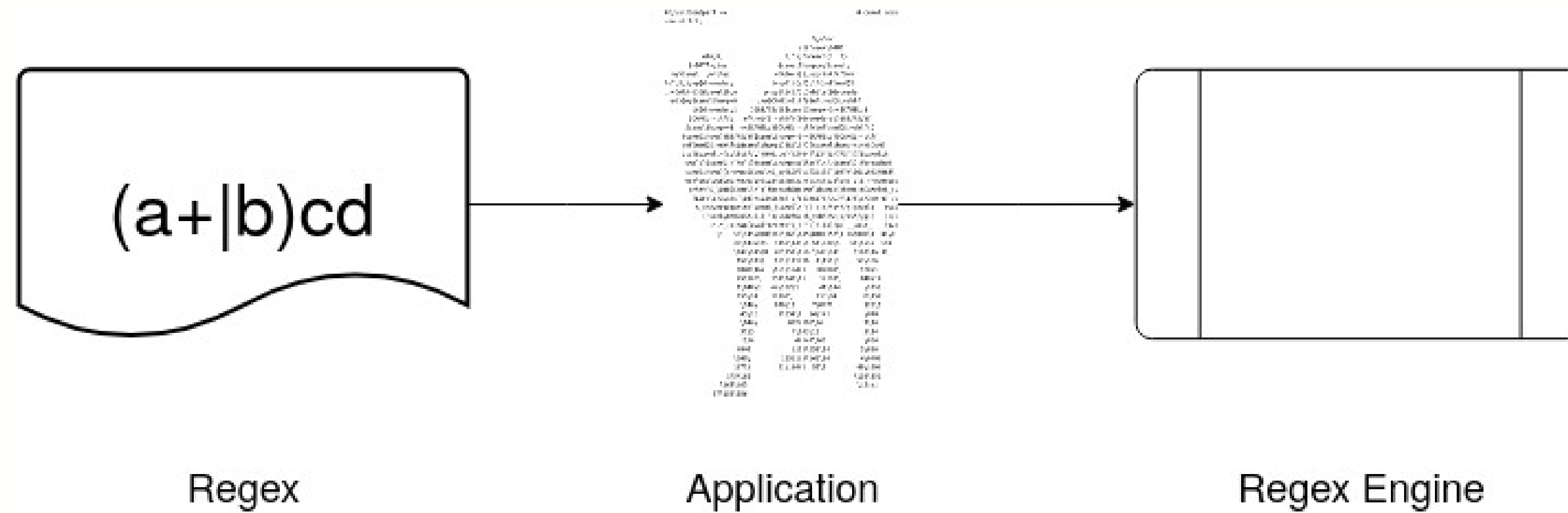
User defined regular expressions – what can go wrong?
(no matter if the ticket systems are affected)

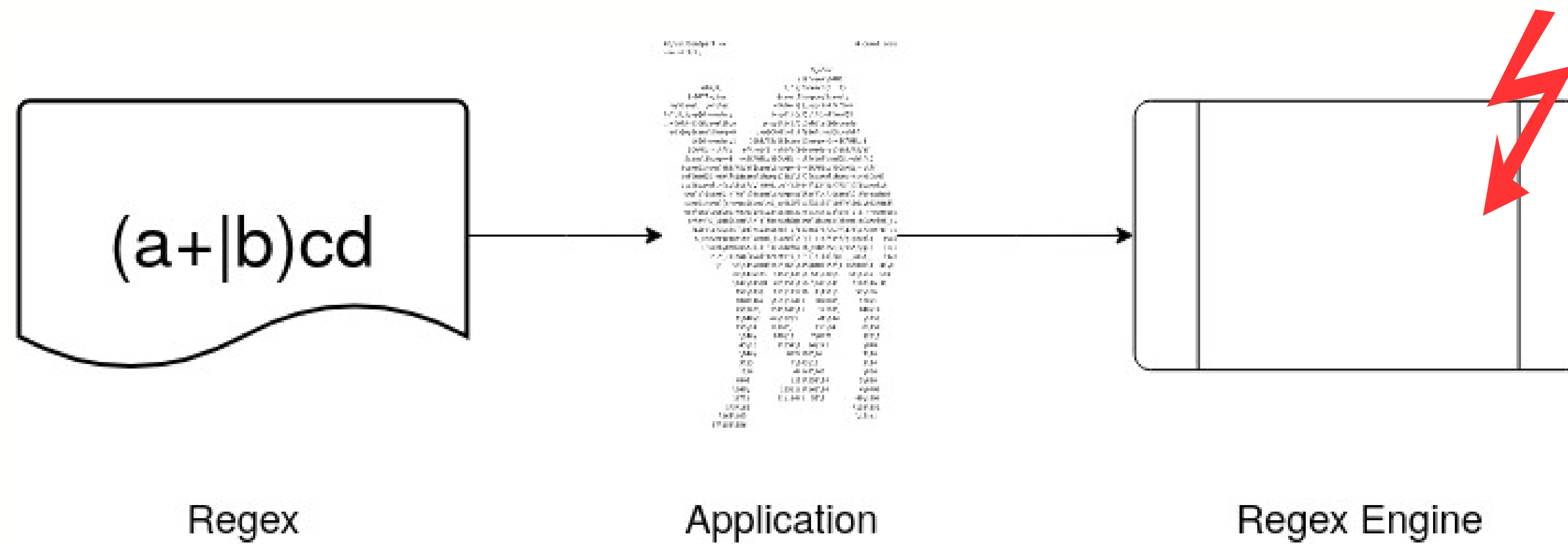
What black hats might want...

They want...

- to execute code on your machine
- access to your machine
- to get data
- high load on your machine

**They need to
identify
security issues**





The Regex engine

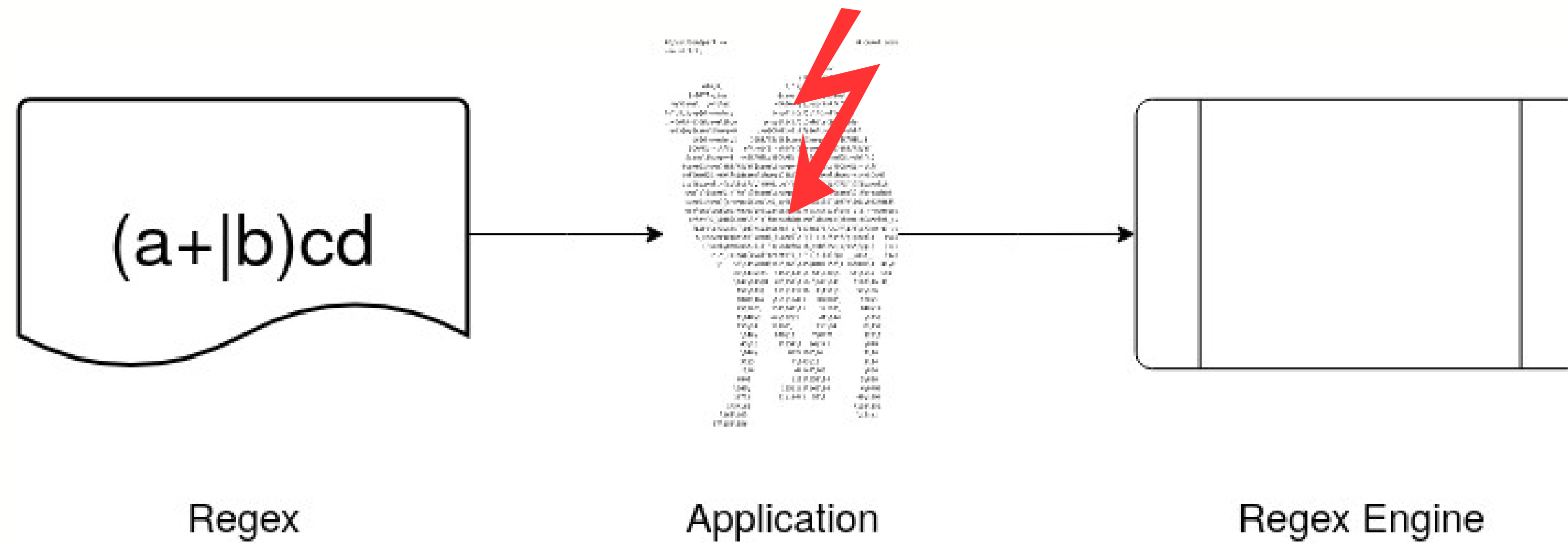
... can have security issues

<input type="checkbox"/>	<input type="checkbox"/> 6 Open ✓ 98 Closed	Author ▾	Label ▾	Projects ▾	Milestones ▾	Assignee ▾	Sort ▾
<input type="checkbox"/>	a1325b902d breaks build on OpenBSD Bug Severity High affects-blead distro-openbsd smoke type-core 12						
	#18536 by jkeenan was closed 20 days ago						
<input type="checkbox"/>	Matching fancy Unicode regex against an ASCII string leaks memory Severity High affects-5.31 type-core 8						
	#17140 by p5pRT was closed on Sep 3, 2019						
<input type="checkbox"/>	Assertion failure in S_scan_const (token.c:4063) Severity High affects-5.29 type-core 12						
	#16981 by p5pRT was closed on May 22, 2019 5.30.0						
<input type="checkbox"/>	Assertion failure in S_find_span_end_mask (regex.c:689) Severity High affects-5.29 type-core 9						
	#16937 by p5pRT was closed on May 22, 2019						

The Regex engine

... can have security issues

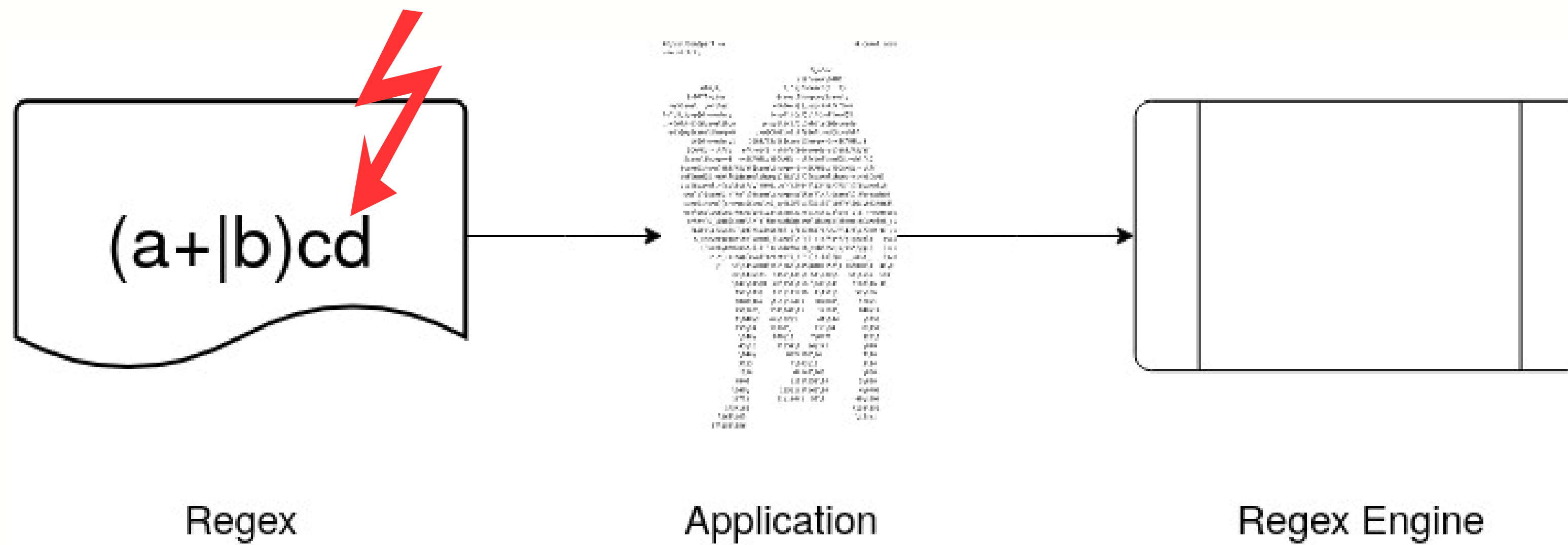
- Impact depends on the issue
- May lead to data leaks ...
- ... or application crash ...
- ... or anything else



The application

... can have security issues, too

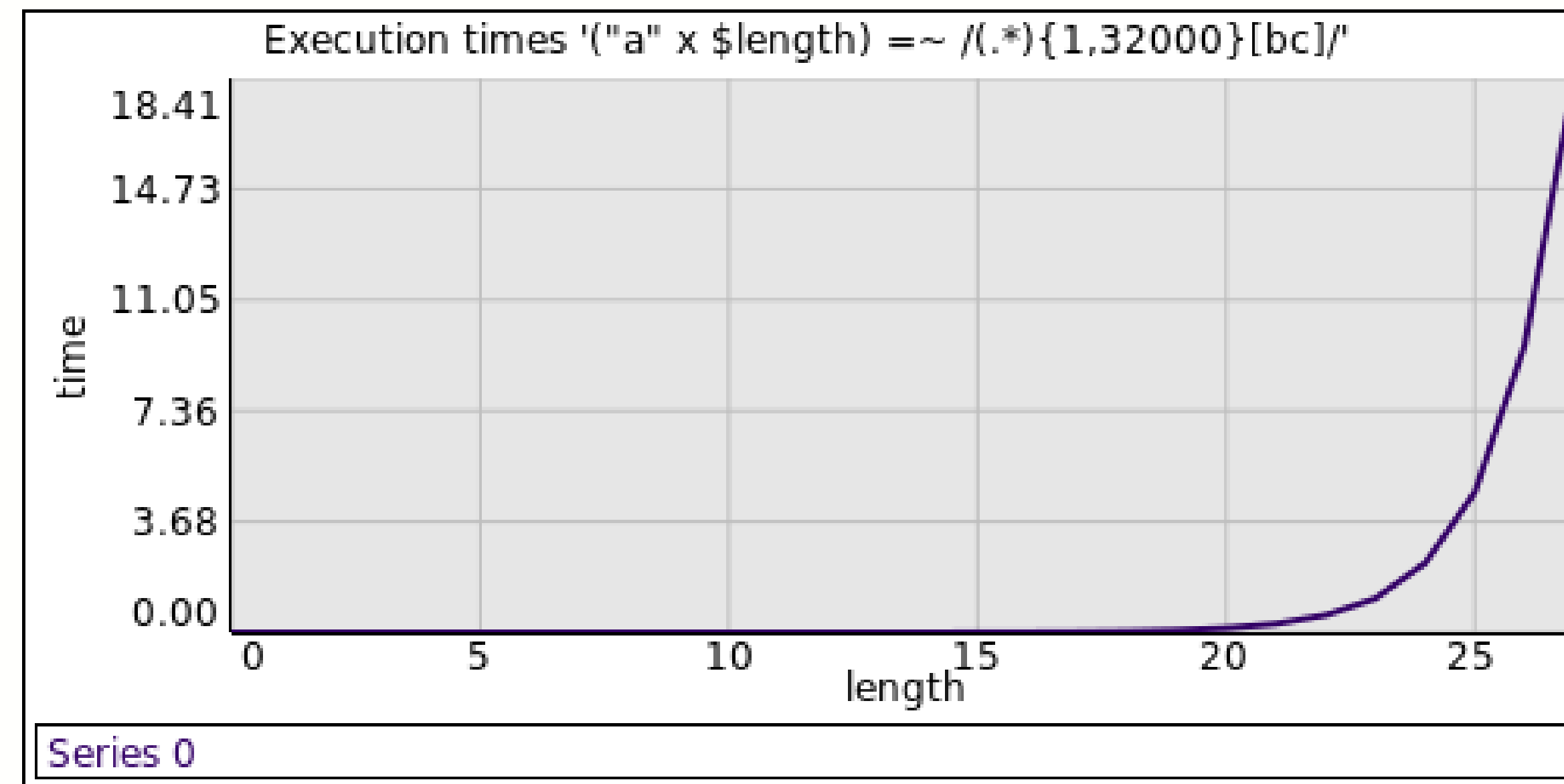
- Execute user-defined code
(and regexes)
- Show detailed error messages
to user
- ...



The regex

... can be painful...

- Regexes can have code elements
- Regexes with wrong input can lead to DoS → ReDoS



Regex from <https://perlgeek.de/blog-en/perl-tips/in-search-of-an-exponential-regexp.html>

ReDoS alarm

- Grouping construct with repetition
- Inside the repeated group there should appear
 - Repetition
 - Alternation with overlapping

```
1) (a+)+  
2) ([a-zA-Z]+)*  
3) (a|aa)+  
4) (a|a?)+  
5) (.a){x}    | for x > 10
```

Payload: “aaaaaaaaaaaaaaaaaaaaaX”

ReDoS alarm – examples

 CKEditor 4[Overview](#) [Demo](#) [Menu ▾](#)

Jan 26/2021

Security Updates:

- Fixed ReDoS vulnerability in the [Autolink](#) plugin.

Issue summary: It was possible to execute a ReDoS-type attack inside CKEditor 4 by persuading a victim to paste a specially crafted URL-like text into the editor and press Enter or Space.

Fixed ReDoS vulnerability in the [Advanced Tab for Dialogs](#) plugin.

Issue summary: It was possible to execute a ReDoS-type attack inside CKEditor 4 by persuading a victim to paste a specially crafted text into the Styles dialog.

```
▼ 2  plugins/autolink/plugin.js   
↑  
@@ -150,7 +150,7 @@  
150 150 * @since 4.11.0  
151 151 * @member CKEDITOR.config  
152 152 */  
153 - CKEDITOR.config.autolink_urlRegex = /^(https?|ftp):\/\/(-\.)?([^\s\/?\.#]+\.?)(\.[^\s\/?\.#]+)?([^\s\/?\.#]+)?$/i;  
153 + CKEDITOR.config.autolink_urlRegex = /^(https?|ftp):\/\/(-\.)?([^\s\/?\.#]+\.?)(\.[^\s\/?\.#]+)?([^\s\/?\.#]+)?$/i;  
154 154 // Regex by Imme Emosol.  
155 155
```

ReDoS alarm – examples

ZSA-2021-03

Date: 2021-03-10

Affected: all versions of OTRS Community Edition; Znuny LTS 6.0.32

Severity: medium

CVE: CVE-Pending

There is a denial of service (DOS) issue, when a mail with a special crafted url is received. This can lead to a maxout of the available server-CPU(s) and can reduce the quality of service or even bring the system to a halt.

The issue is fixed in the current stable release Znuny LTS 6.0.33. An update of Znuny LTS to the latest version is recommended.

↓	↑	@@ -44,7 +45,7 @@ sub Pre {	
44	45	(# \$2
45	46	(?:	# http or only www
46	47	(?: (?: http s? ftp) :\\ \\)	# http://, https:// and ftp://
47	-	(?: [a-z0-9\\-]* \\.	# allow for sub-domain or prefixes bug#12472
48	+	(?: [a-z0-9\\-]{0,255} \\.	# allow for sub-domain or prefixes bug#12472;
48	49	(?: www ftp) \\.	# www.something and ftp.something
49	50)	
50	51)	
↓			

How to (try to) avoid the pitfalls

The Regex engine

... can have security issues

- Keep your Perl up to date

Avoid information leaks

- Check regexes for syntax errors
- do not show error messages to the public

```
use v5.24;
```

```
use strict;  
use warnings;
```

```
my $pat = "(a|b)+cd";  
my $re = eval {  
    qr/$pat/;  
};
```

Avoid code execution

- User defined regex ****should**** be no problem
- Avoid using "use re 'eval'"

```
use v5.24;
```

```
use strict;  
use warnings;
```

```
my $pat = "a(?{ say 'yes' })";  
my $re = qr/$pat/;
```

```
> perl regex_code.pl  
Eval-group not allowed at runtime, use  
re 'eval' in regex m/a(?{ say 'yes' })/  
at regex_code.pl line xx.
```

Avoid ReDoS

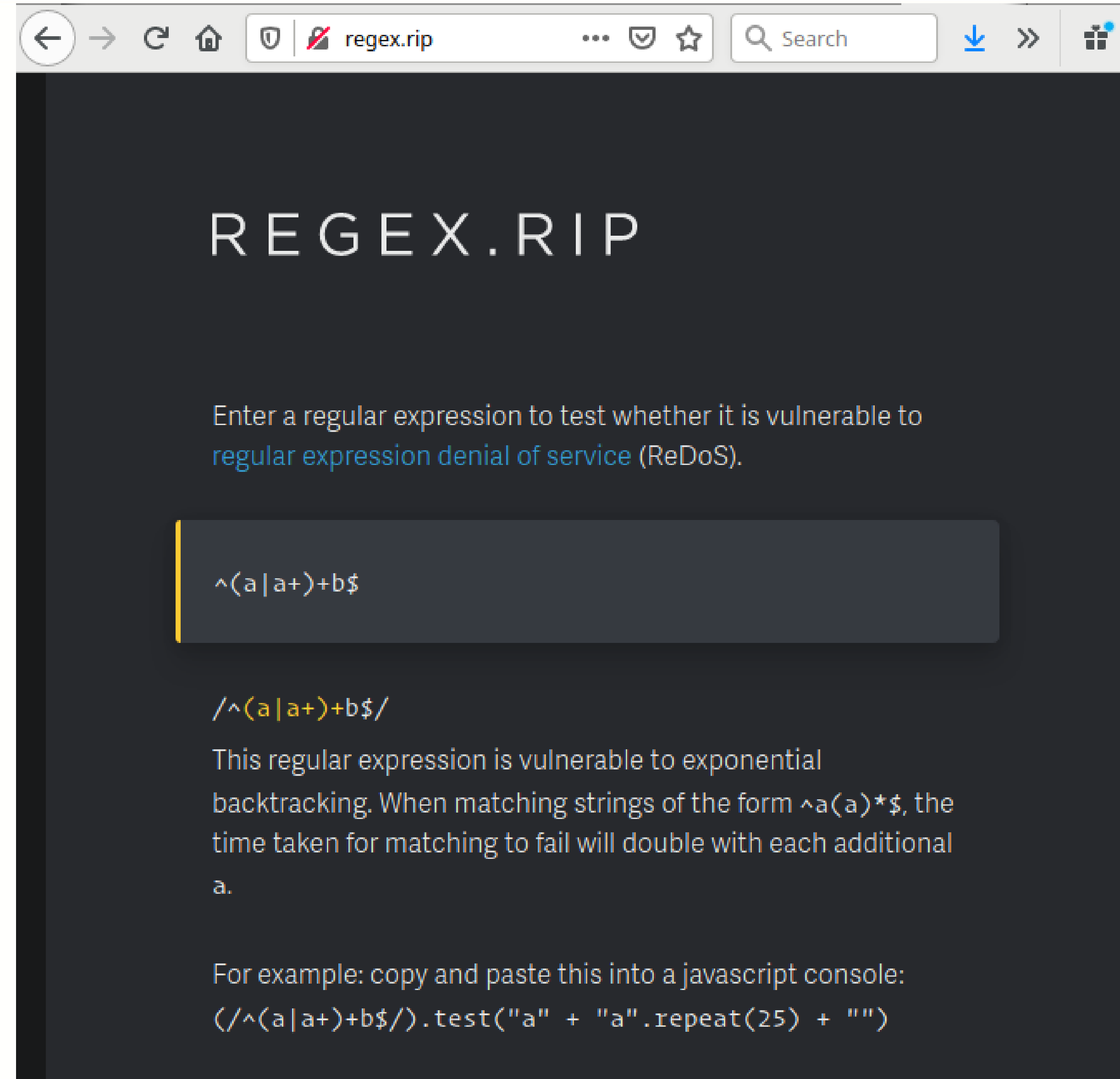
- Check user-defined regexes for „malicious“ stuff
- Use different regex engine
- Limit execution time

Avoid ReDoS

- Check user-defined regexes for „malicious“ stuff
- Use different regex engine
- Limit execution time
- Send regexes to
 - <http://regex.rip>
 - <http://redos-checker.surge.sh/>
- Most tools do not support full Perl regex syntax

Avoid ReDoS

- Check user-defined regexes for „malicious“ stuff
- Use different regex engine
- Limit execution time



The screenshot shows a web browser window with the address bar displaying 'regex.rip'. The page title is 'REGEX.RIP'. The main content area has a dark background and contains the following text:

Enter a regular expression to test whether it is vulnerable to regular expression denial of service (ReDoS).

A text input field contains the regular expression: `^(a|a+)+b$`

Below the input field, the regular expression is repeated: `/^(a|a+)+b$/`

The text continues: "This regular expression is vulnerable to exponential backtracking. When matching strings of the form `^a(a)*$`, the time taken for matching to fail will double with each additional `a`."

Finally, it provides an example: "For example: copy and paste this into a javascript console: `(/^(a|a+)+b$/).test("a" + "a".repeat(25) + "")`"

Avoid ReDoS

- Check user-defined regexes for „malicious“ stuff
- Use different regex engine
- Limit execution time



Avoid ReDoS

- Check user-defined regexes for „malicious“ stuff
- Use different regex engine
- Limit execution time
- Check with PPI and PPIx::Regex

Check Regexes with PPI and PPIx::Regexp

```
my $doc = PPI::Document->new( \'$var =~ m{foo}smx' );  
PPI::Dumper->new( $doc )->print;
```

```
PPI::Document  
  PPI::Statement  
    [...]   
      PPI::Token::Regexp::Match 'm{foo}smx'
```

Check Regexes with PPI and PPIx::Regexp

```
my $doc = PPI::Document->new( \'$var =~ s{foo}{bar}smx' );  
PPI::Dumper->new( $doc )->print;
```

```
PPI::Document  
  PPI::Statement  
    [...]   
      PPI::Token::Regexp::Substitute 's{foo}{bar}smx'
```

Check Regexes with PPI and PPIx::Regexp

```
my $doc = PPI::Document->new( \'$var = qr/test/' );  
PPI::Dumper->new( $doc )->print;
```

```
PPI::Document  
  PPI::Statement  
    [...]   
      PPI::Token::QuoteLike::Regexp 'qr/test/'
```

Check Regexes with PPI and PPIx::Regexp

```
my $perl_code = q!  
    my $var = 'hello foo';  
    $var =~ m{foo}smx;  
    $var =~ s{foo}{bar}xms;  
  
    $var = qr/hello/;  
!;  
  
my $doc = PPI::Document->new( \$perl_code );  
PPI::Dumper->new( $doc )->print;  
  
my $regex_nodes = $doc->find( sub {  
    $_[1]->isa( 'PPI::Token::QuoteLike::Regexp' ) ||  
    $_[1]->isa( 'PPI::Token::Regexp::Match' ) ||  
    $_[1]->isa( 'PPI::Token::Regexp::Substitute' )  
});
```


Check Regexes with PPI and PPIx::Regexp

```
my $re = PPIx::Regexp->new(
    '/^(?{ say "yes" })(?:a+|aa+)+$/smx'
);

PPIx::Regexp::Dumper->new( $re )
    ->print();
```

```
PPIx::Regexp    failures=0
PPIx::Regexp::Token::Structure ''
PPIx::Regexp::Structure::Regexp / ... /
PPIx::Regexp::Token::Assertion '^'
PPIx::Regexp::Structure::Code (? ... )
PPIx::Regexp::Token::Code '{ say "yes" }'
PPIx::Regexp::Structure::Modifier (?: ... )
PPIx::Regexp::Token::Literal 'a'
PPIx::Regexp::Token::Quantifier '+'
PPIx::Regexp::Token::Operator '|'
PPIx::Regexp::Token::Literal 'a'
PPIx::Regexp::Token::Literal 'a'
PPIx::Regexp::Token::Quantifier '+'
PPIx::Regexp::Token::Quantifier '+'
PPIx::Regexp::Token::Modifier 'smx'
```


Check Regexes with PPI and PPIx::Regexp

```
my $perl_code = q!  
    my $var = 'hello foo';  
    $var =~ m{foo}smx;  
    $var =~ s{foo}{bar}xms;  
  
    $var = qr/hello/;  
!;  
  
my $doc = PPI::Document->new( \$perl_code );  
PPI::Dumper->new( $doc )->print;  
  
my $regex_nodes = $doc->find( sub {  
    $_[1]->isa( 'PPI::Token::QuoteLike::Regexp' ) ||  
    $_[1]->isa( 'PPI::Token::Regexp::Match' ) ||  
    $_[1]->isa( 'PPI::Token::Regexp::Substitute' )  
});
```

Check Regexes with PPI and PPIx::Regexp

```
for my $re ( @{ $found || [] } ) {  
    my $pattern = $re->content;  
  
    my $ppi_re = PPIx::Regexp->new( $pattern );  
  
    warn "Skip Regex" if $ppi_re->find_first( 'PPIx::Regexp::Structure::Code' );  
}
```

Check Regexes with PPI and PPIx::Regexp

```
my $qr = qr/  
    ^(?&TEST)$  
    (? (DEFINE)  
        (?<TEST>test)  
    )  
/xms;  
  
my $re2 = PPIx::Regexp->new( "$qr" );  
PPIx::Regexp::Dumper->new( $re2 )  
    ->print();
```

```
PPIx::Regexp    failures=1  
PPIx::Regexp::Token::Unknown '(?^msx:  
    ^(?&TEST)$  
    (? (DEFINE)  
        (?<TEST>test)  
    )  
)'Tokenizer found illegal first characters
```

Avoid ReDoS

- Check user-defined regexes for „malicious“ stuff
- Avoid Backtracking
- Limit execution time

Avoid ReDoS

- Check user-defined regexes for „malicious“ stuff
- **Avoid Backtracking**
- Limit execution time
- Check if you can use a regex engine without backtracking (e.g. Google's RE2)
- Avoid Backtracking Perl regexes
- Other engines might not support the features you need

Use RE2 engine

```
my $length = 25;
my $string = "a" x $length;
my $pattern = '(.*){1,200}[bc]';

{
    my $start = time;
    $string =~ /$pattern/;
    say sprintf "It took $duration seconds", (time-$start);
}

{
    use re::engine::RE2;
    my $start = time;
    $string =~ /$pattern/;
    say sprintf "It took $duration seconds", (time-$start);
}
```

Use RE2 engine

```
my $length = 25;
my $string = "a" x $length;
my $pattern = '(.*){1,200}[bc]';

{
    my $start = time;
    $string =~ /$pattern/;
    say sprintf "It took $duration seconds", (time-$start);
}

{
    use re::engine::RE2;
    my $start = time;
    $string =~ /$pattern/;
    say sprintf "It took $duration seconds", (time-$start);
}
```


Use RE2 engine

```
my $length = 25;
my $string = "a" x $length;
my $pattern = '(.*){1,200}[bc]';

{
    my $start = time;
    $string =~ /$pattern/;
    say sprintf "It took $duration seconds", (time-$start);
}

{
    use re::engine::RE2;
    my $start = time;
    $string =~ /$pattern/;
    say sprintf "It took $duration seconds", (time-$start);
}
```


Use RE2 engine

```
my $length = 25;
my $string = "a" x $length;
my $pattern = '(.*){1,200}[bc]';

{
    my $start = time;
    $string =~ /$pattern/;
    say sprintf "It took $duration seconds", (time-$start);
}

{
    use re::engine::RE2;
    my $start = time;
    $string =~ /$pattern/;
    say sprintf "It took $duration seconds", (time-$start);
}
```

```
> perl regex_redos.pl
It took 4.11798501014709 seconds
It took 0.000406980514526367 seconds
```

Use RE2 engine

```
my @pattern = (  
    '(.*){1,200}[bc]',  
    '(.*){1,32000}[bc]',  
);  
  
use re::engine::RE2;  
  
for my $pattern ( @pattern ) {  
    my $re = qr/$pattern/;  
    say $pattern, ": ",  
        ( $re->isa('re::engine::RE2') ? 'yes' : 'no' );  
}
```

Use RE2 engine

```
my @pattern = (  
    '(.*){1,200}[bc]',  
    '(.*){1,32000}[bc]',  
);  
  
use re::engine::RE2;  
  
for my $pattern ( @pattern ) {  
    my $re = qr/$pattern/;  
    say $pattern, ": ",  
        ( $re->isa('re::engine::RE2') ? 'yes' : 'no' );  
}
```

```
> perl re2.pl  
(.*){1,200}[bc]: yes  
(.*){1,32000}[bc]: no
```

Avoid Backtracking in Perl

```
my $re = qr/(a|a+)+[bc]/xms;  
my $string = ('a' x 26);
```

```
$1 = ''
```

```
say "yes" if $string =~ $re;
```

```
Back-tracked within regex and recapture to $1
```

```
|  
V  
/(a|a+)+[bc]/
```

```
|  
V  
'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa;
```

```
[Visual of regex at 'avoid_redos.pl' line 10] [step: 121]
```

Avoid Backtracking in Perl

```
my $re = qr/(a|a+)+(*COMMIT)[bc]/xms;  
my $string = ('a' x 26);
```

```
$1 = ''
```

```
say "yes" if $string =~ $re;
```

```
Back-tracked within regex and recapture to $1
```

```
|  
V  
/(a|a+)+[bc]/
```

```
|  
V  
'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa'
```

```
[Visual of regex at 'avoid_redos.pl' line 10] [step: 121]
```

Avoid ReDoS

- Check user-defined regexes for „malicious“ stuff
- Avoid Backtracking
- Limit execution time
- Use `$SIG{ALRM}` to interrupt regex execution

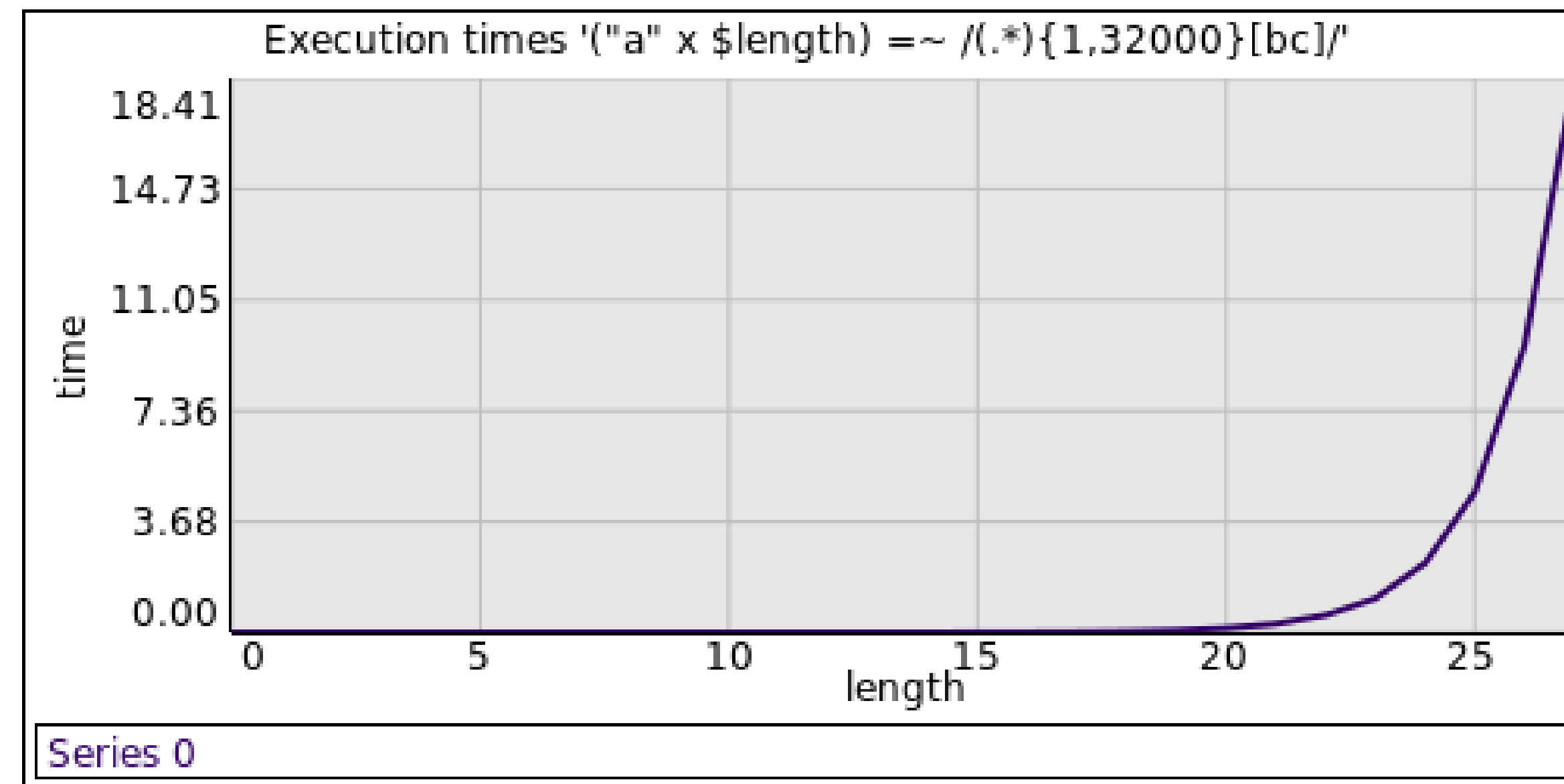
Limit execution time

```
use Time::HiRes qw(time);

my $string  = "a" x 25;
my $pattern = '(.*){1,32000}[bc]';

{
    my $start = time;
    $string =~ /$pattern/;
    say sprintf "It took $duration seconds", (time-$start);
}

eval {
    local $SIG{ALRM} = sub { die "Regex took too long" };
    $string =~ /( ?{ alarm 2 } )$pattern/;
    1;
} or warn $@;
```



Regex from <https://perlgeek.de/blog-en/perl-tips/in-search-of-an-exponential-regexp.html>

Limit execution time

```
use Time::HiRes qw(time);

my $string = "a" x 25;
my $pattern = '(.*){1,32000}[bc]';

{
    my $start = time;
    $string =~ /$pattern/;
    say sprintf "It took $duration seconds", (time-$start);
}

eval {
    local $SIG{ALRM} = sub { die "Regex took too long" };
    $string =~ /( ?{ alarm 2 } )$pattern/;
    1;
} or warn $@;
```

Limit execution time

```
use Time::HiRes qw(time);

my $string = "a" x 25;
my $pattern = '(.*){1,32000}[bc]';

{
    my $start = time;
    $string =~ /$pattern/;
    say sprintf "It took $duration seconds", (time-$start);
}

eval {
    local $SIG{ALRM} = sub { die "Regex took too long" };
    $string =~ /( ?{ alarm 2 } )$pattern/;
    1;
} or warn $@;
```

Limit execution time

```
use Time::HiRes qw(time);

my $string = "a" x 25;
my $pattern = '(.*){1,32000}[bc]';

{
    my $start = time;
    $string =~ /$pattern/;
    say sprintf "It took $duration seconds", (time-$start);
}

eval {
    local $SIG{ALRM} = sub { die "Regex took too long" };
    alarm 2;
    $string =~ /$pattern/;
    1;
} or warn $@;
```

Limit execution time

```
use Time::HiRes qw(time);

my $string = "a" x 25;
my $pattern = '(.*){1,32000}[bc]';

{
    my $start = time;
    $string =~ /$pattern/;
    say sprintf "It took $duration seconds", (time-$start);
}

eval {
    local $SIG{ALRM} = sub { die "Regex took too long" };
    $string =~ /( ?{ alarm 2 } )$pattern/;
    1;
} or warn $@;
```

Limit execution time

```
use Time::HiRes qw(time);

my $string = "a" x 25;
my $pattern = '(.*){1,32000}[bc]';

{
    my $start = time;
    $string =~ /$pattern/;
    say sprintf "It took $duration seconds", (time-$start);
}

eval {
    local $SIG{ALRM} = sub { die "Regex took too long" };
    $string =~ /( ?{ alarm 2 } )$pattern/;
    1;
} or warn $@;
```

```
> perl regex_redos.pl
Regex took too long at regex_redos.pl
    line 15.
It took 4.08752989768982 seconds
```

**There's no
such thing as
100% security**

**... but you can
try to get as
close as
possible**

Thank you!

Questions?

Slides available at <https://gitlab.com/perl-academy/talks/2021/gpw/>