RUBY

ОСОБЕНОСТИ ОТ ГЛЕДНА ТОЧКА НА СИГУРНОСТТА

НАБЪРЗО ЗА RUBY

Динамичен, обектно-ориентиран, general-purpose, но предимно използван за Web, език за програмиране

ВСИЧКО Е ОБЕКТ

```ruby
123.class
#=> Fixnum
```

```ruby
"Ruby".class
#=> String
```

```ruby
nil.class
#=> NilClass
```

```ruby
true.class
#=> TrueClass
```

ПРОМЕНЛИВИ

- глобални:

```
$FOO = "bar"
```

- на инстанцията:

```
@foo = "bar"
```

- на класа:

```
@@foo = "bar"
```

- локални:

```
foo = "bar"
```

Когато решите да дефинирате глобална променлива в Ruby, станете от компютъра, изпийте една студена вода и се разходете.

Ако когато се върнете, все още имате желание да дефинирате глобална променлива, прочетете (отново) тази статия:

http://c2.com/cgi/wiki?GlobalVariablesAreBad

НИЗОВЕ

Литералите за низ в Ruby са единични и двойни кавички

```
'Васил „не“ пие' # <--- Низ :)
```

UTF-8 кодирани, но част от операциите върху тях действат само върху ASCII символите им:

```
"Васил „не“ pie".upcase
#=> "Васил „не“ PIE"
```

Създаването на низ с двойни кавички предоставя интерполация

```
"1 + 2 = #{1 + 2}"
#=> "1 + 2 = 3"
```

СИМВОЛИ

Интернирани низове от символи

Интернирани?

Immutable, всяка уникална стойност остава в паметта до приключване на изпълнението. Използват се например за ключове за хешове и за други работи :).

```
animal = :crocodile
animals[animal] = Vasil.new
animals[:crocodile]
#=> #<Vasil:0x007fb7f049d180>

:crocodile.object_id
#=> 1390428

animal.object_id
#=> 1390428
```

КЛАСОВЕ И МЕТОДИ

Клас се дефинира така:

```
class Vasil
end
```

Метод се дефинира така:

```
class Vasil
  def doesnt_drink?
    false
  end
end
```

ОПАСНИТЕ НЕЩА

ПРЕПЪЛВАНЕ НА ПАМЕТТА

Преди Ruby 2.2.0 всички символи остават в паметта до приключване на изпълнението на програмата.

Това означава, че ако уникални символи се създават от потребителски данни, приложението може да препълни паметта на машината, на която работи.

След 2.2.0 динамично създадените символи се освобождават от garbage collection процеса.

В Ruby (почти) всичко може да бъде променено по време на изпълнение

Под почти всичко да се разбира, че дори константите могат да бъдат променяни

```
VASIL_DOESNT_DRINK = false
VASIL_DOESNT_DRINK = "true"
# warning: already initialized constant VASIL_DOESNT_DRINK
# warning: previous definition of VASIL_DOESNT_DRINK was here
VASIL_DOESNT_DRINK
=> "true"
```

ИЗВОДИ

Програмистите трябва да внимават да не променят неочаквано нещо... Което не е толкова трудно, ако програмистите спазват конвенциите.

МЕТАПРОГРАМИРАНЕ

- `eval`
- `instance_eval`
- `class_eval`
- `module_eval`

До тези методи никога не трябва да достигат данни под контрола на потребителите.

ДИНАМИЧНО ИЗВИКВАНЕ НА МЕТОДИ

- `Object#send` (извиква дори private методи)
- `Object#respond_to?`

Отново, ако аргументите на тези методи са под контрола на потребителите, you're gonna have a bad time.

GOOD OLD SHELL INJECTION

- `Kernel#system`
- `Kernel#exec`
- `Kernel#syscall`
- `%x(foo)`
- `` `foo` ``

Параметрите на тези команди винаги трябва да бъдат екранирани, ако са под контрола на потребителите

Защо?

```
system("rf -rf #{params[:file]}")

exec(params[:command])

`unlink #{params[:something]}`
```

Нека преди изпълнение на горните команди, сме дефинирали тази променлива:

```
params = {file: '/home/vasil',
          command: 'rm -rf /home/vasil',
        something: '/home/vasil/Desktop/passwords.txt'}
```

ОПЕРАЦИИ С ФАЙЛОВЕ

`Dir`, `File`, `IO`, `Kernel`, `Net::FTP`, `Net::HTTP`, `PStore`, `Pathname`, `Shell` и т.н.

Препоръчително е имената на файловете/директориите да не бъдат под контрола на потребителите ИЛИ да бъдат екранирани по правилния начин

А не така:

```
File.read "/home/application/upload/#{params[:file]}"

File.read '/home/application/upload/../../../etc/passwd'
```

РЕГУЛЯРНИ ИЗРАЗИ

Като навсякъде другаде, с едно доста важно изключение

За начало и край на низ не се използват ^ и $.

Използват се \A и \z.

Какво става, ако се объркаме?

Този регулярен израз:

```
/^https?:\/\/[^\n]+$/i
```

Ще разпознае този низ от символи:

```
javascript:exploit_code();/*
http://hi.com
*/
```

Повече за този вид атаки по-нататък в курса.

ПАРАЛЕЛНО ПРОГРАМИРАНЕ

MRI има global interpreter lock. Други имплементации на Ruby нямат.

С изключение на това, особеностите на паралелното програмиране с Ruby са стандартни.

УЯЗВИМОСТИ НА СТАНДАРТНАТА БИБЛИОТЕКА И RUNTIME-А

MRI е написан на C

Rubinius е написан на C++

Съществуват редица имплементации на Ruby.

Следете уязвимостите, асоциирани с имплементацията, която използвате.

УЯЗВИМОСТИ НА ИЗПОЛЗВАНИТЕ ОТ ПРОГРАМАТА БИБЛИОТЕКИ

Всеки може да пише джемове за Ruby.

Не всеки може да пише джемове, които нямат дупки в сигурността.

Rubygems има функционалност за подписване на пакети, но тя масово не се ползва.

Доскоро пакетите се трансферираха по HTTP.

ЗАКЛЮЧЕНИЕ

Следвайте конвенциите.

Екранирайте входящите данни.

Следете уязвимостите на платформата си.