УДК 519.683.8

О.В. Дробнич, М.П. Дробнич, В.М. Різак
Ужгородський національний університет
88000, Ужгород, вул. Волошина, 54
e-mail: adrobnych@gmail.com

ПРОБЛЕМА ВИКОНАННЯ СТОРОННЬОГО КОДУ У ГРІДАХ, ЩО ПОБУДОВАНІ НА ВІРТУАЛЬНИХ МАШИНАХ RUBY

Декларується проблема побудови безпечної “пісочниці” для виконання зовнішнього коду в грідах на основі мови програмування Ruby. Аналізуються шляхи вирішення цієї проблеми.

Ключові слова: розподілені розрахунки, методи Монте-Карло, моделювання з перших принципів, Ruby, метапрограмування, sandbox, ObjectSpace, грід.

Вступ

У роботах [1, 2] обґрунтовується актуальність застосування динамічних високорівневих мов програмування, і, зокрема, мови Ruby як основи для розробки і постановки розподілених чисельних експериментів. У нашій попередній роботі [2] ми описали спосіб запуску розподіленого додатку (розрахунки методом Монте-Карло в двовимірній гратці Ізінга [4]) без прив’язки до інфраструктури гріда [3]. У даній роботі ми розбираємо один із аспектів роботи гріда — безпечне виконання стороннього коду.

Проблема

Мова Ruby, як і деякі інші сучасні мови програмування (Objective C, Objective J) походять від мови програмування SmallTalk, що характеризується потужною об'єктно-орієнтовною концепцією (передача повідомлень замість викликів функцій, зміна поведінки об'єктів після їх створення) та динамічністю (динамічні типи даних та метапрограмування). Найбільш цікавою особливістю для побудови гріду є метапрограмування — тобто програмна обробка коду (замість даних). Дійсно, віртуальна машина Ruby має велику кількість засобів запуску стороннього коду. Це дає можливість відокремити та спростити розробку інфраструктури такого гріда. Наприклад, ми можемо сконцентруватись на керуванні атрибутами безпеки, керуванням нодами і ресурсами незалежно від коду, який власне буде проводити розподілені розрахунки (який є власне клієнтським кодом у термінах архітектури програмних систем).

Що стосується клієнтського коду, то він без перешкод може завантажуватись і виконуватись на нодах гріда. Очевидним негативними моментом є повна відсутність системи безпеки, яка мала би блокувати небезпечний код.

Одже, нашою задачею є побудова так званої „пісочниці” (sandbox), що ізолює клієнтський код від коду інфраструктури гріда. У ході даної статті ми розглядатимемо різні способи реалізації такої „пісочниці”.

Рівні безпеки Ruby

Змоделюємо ситуацію, коли замість коду розрахунків грід отримує код для так званої атаки грубої сили (Brute Force) на певний вебсайт. Цей код просто викликає команду HTTP GET, але якщо цей виклик зациклити і розповсюдити на сотню нодів гріда, то вебсайт може бути виведений з ладу.

Розглянемо приклад небезпечного коду на лістінгу L1. Звернемо увагу на конструкцію open(url) у сендвіч паттерні. Вона власне і робить код небезпечним.

Тепер вставимо наступну конструкцію перед викликом open:

```
$SAFE = 4
```

При виконанні модифікованого коду замість контенту з вебсайту ми бачимо помилку виконання і скрипт зупиняється:

```
>ruby test-with-safe.rb
C:/Ruby192/lib/ruby/1.9.1/open-uri.rb:
706:in `include?': Insecure operation
`include?' at level 4 (SecurityError)
```

Тут ми бачимо використання вбудованої системи безпеки мови Ruby. Ця система будується навколо поняття “зіпсованих” (tainted) об'єктів — об'єктів, що надійшли із зовнішніх джерел по відношенню до поточного оточення виконання. Існує 5 рівнів безпеки, які задаються глобальною змінною $SAFE:

$SAFE = 0

– не виковнується жодних перевірок стосовно зіпсованих об'єктів. Це є режимом по замовчанню для програм на мові Ruby.

$SAFE = 1

– Ruby забороняє використання зіпсованих об'єктів для потенційно небезпечних операцій.

$SAFE = 2

– додатково до попереднього рівня Ruby забороняє завантажувати програмні файли із публічних джерел.

$SAFE = 3

– додатково всі новостворені об'єкти розглядаються як зіпсовані.

$SAFE = 4

- додатково до попередніх рівнів Ruby ефективно розділяє виконання програми на два простори об'єктів. Незіпсовані об'єкти не можуть модифікуватися. Інфраструктура виконання стороннього коду будується із використанням нижчих рівнів безпеки і поставляє безпечні об'єкти. Потім рівень безпеки підвищується до 4 щоб уникнути наступних змін у цьому оточенні.

Слід відмітити, що вбудована система безпеки мови Ruby знаходиться у постійному розвитку і на даний момент не роглядається як абсолютно безпечний спосіб запуску стороннього коду. Головні її недоліки – це занадто спрощена і жорстка схема правил. В ідеалі, автор стороннього коду має власноруч розставити модифікатори безпеки у своєму коді, що порушує логіку побудови надійної „пісочниці”.

Система безпеки мови Java

Цікавим трендом є поява і розвиток вбудованих мов програмування, зокрема мови JRuby. Це повноцінна імплементація віртуальної машини Ruby, що працює як окремий легкий процес у віртуальній машині Java. Це дає доступ коду на мові Ruby до самої потужної екосистеми бібліотек і фреймворків, розроблених на мові Java. В тому числі до надійного і перевіреного роками фреймворку безпеки мови Java, що називається Java Security Manager.

Можливо, у деякий момент у майбутньому це буде хорошим вирішенням проблеми пісочниці у мові Ruby, але на даний момент лінійна швидкість JRuby значно поступається як Ruby так і Java і є недостатньою для виконання розподілених розрахунків.

Трасування виконання

Розглянемо код на лістінгу L2. В результаті запуску цього коду отримаємо:

```
>ruby trace2.rb
c-return trace2.rb:7 set_trace_func Kernel
    line trace2.rb:9
    line trace2.rb:10
    line trace2.rb:12
  c-call trace2.rb:12 inherited Class
c-return trace2.rb:12 inherited Class
   class trace2.rb:12
u can't create a class
    end trace2.rb:12
```

Зверніть увагу на виклик методу set_trace_func, що є системним колбеком, тобто при виконанні кожного рядку програми Ruby буде автоматично викликати блок коду, що був переданий цьому методу.

Імплементація „пісочниці” може бути побудована на подібному трасуванні стороннього коду. Однак, як видно із результатів запуску прикладу коду, даний колбек має доволі посередні можливості в розборі і аналізі коду, що виконується. Наприклад, у програмі L2 вдається розпізнати і зупинити лише спробу перевантаження класу String.

Побудова власної системи безпеки на основі властивостей ObjectSpace

Немаючи готової вбудованої системи для „пісочниці”, Ruby надає потужні атомарні операції для метапрограмування. Ми можемо сканувати простір пам'яті його віртуальної машини і визначати які класи і об'єкти на даний момент є завантаженими (оскільки класи також є об'єктами, то їх обробка не відрізняється від обробки інших об'єктів). Після чого ми можемо заборонити створення нових і перевантаження існуючих класів. Маючи список імен класів ми можемо відслідкувати операції створення об'єктів і відправлення повідомлень на них.

На лістінгу L3 представлено прототип імплементації „пісочниці” на основі властивостей ObjectSpace. У результаті запуску цього прикладу отримуємо:

```
>ruby os_analyzer.rb
x is some token
= is some token
'hello' is some token
y is some token
= is some token
String is a class
new is some token
x is some token
shuffle is some token
x is some token
= is some token
x is some token
+ is some token
100 is some token
to_s is some token
```

Тобто, ми створили нескладний аналізатор коду, що зміг відслідкувати момент створення нового об'єкту. Слід зазначити, що це лише прототип і слід прикласти додаткові зусилля для перетворення його на повнофункціональну імплементацію „пісочниці”.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. E. Ong. MPI Ruby: Scripting in a Parallel Environment. //Computing in Science and Engineering, 4(4):78–82, 2002.
2. Дробнич О.В., Дробнич М.П., Матяшовський Т.В., Різак В.М.: Використання мови Ruby та технологій DRB і Rinda для проведення розподілених розрахунків із використанням ґраток Ізінга // Науковий вісник Ужгородського Університету. Серія Фізика. – № 27. – 2010, С. 14-20.
3. Bates M. Distributed programming with Ruby. - Addison-Wesley Professional Ruby Series, 2009. – 273 p.
4. Onsager, Lars. Crystal statistics. I. A two-dimensional model with an order-disorder transition", Phys. Rev. (2) 65: 117–149, 1944.

Стаття надійшла до редакції 29.08.2011

Додатки

Лістинг L1

```
require 'open-uri'
require 'pp'

open('http://www.ganttzilla.com/') do |f|
  # hash with meta information
  pp  f.meta
  pp "Content-Type: " + f.content_type
  pp "last modified" + f.last_modified.to_s

  no = 1
  # print the first three lines
  f.each do |line|
    print "#{no}: #{line}"
    no += 1
    break if no > 4
  end
end
```

Лістинг L2

```
set_trace_func proc { |event, file, line, id, binding, classname|
  printf "%8s %s:%-2d %10s %8s\n", event, file, line, id, classname
  if event == "class" then
    puts "u can't create a class"
    exit
  end
}

s = "scscccscs"
s = s + "sxcscss"

class S
  def initialize
     puts "hurray"
  end
end

s = S.new
s.class

puts "happy end"
```

Лістинг L3

```
#this script has to be launched recursively over string interpolations also
#this code has to use more wisdom splitting to tokens. f.e. x=x+1 will be one token now, what is error

class Analyzer
   def initialize
            @known_classes = []
             ObjectSpace.each_object {|o| @known_classes = @known_classes + [o] if o.class == Class }
          end

          def is_a_class? token
             @known_classes.select{|c| c.to_s == token.to_s} != []
          end
```

```
        def is_a_literal? token
           # to do: create Regexp filter for Ruby's literals
        end

  def is_a token
          if is_a_class? token then
             "a class"
          elsif is_a_literal? token then
             "a literal of class #{@literal_of_class}"
          else
             "some token"
    end
  end

end

sample = """
x = 'hello'
y = String.new
x.shuffle
x = x + 100.to_s
"""

#search for classes

analyzer = Analyzer.new

sample.split.each do
  |token|
  token.split(".").each do
     |token|
     #puts token
     puts "#{token} is #{analyzer.is_a token}"
  end
end
```

O.V. Drobnych, M.P. Drobnych, V.M. Rizak
Uzhhorod National University
88000, Uzhhorod, Voloshin Str., 54
e-mail: adrobnych@gmail.com

PROBLEM OF EXECUTION OF EXTERNAL CODE ACROSS GRID SYSTEMS BUILT ON TOP OF RUBY VIRTUAL MACHINES

Problem of creation of safe sandbox for execution of external code inside Ruby grids was stated. Conducted analysis of possible solution of the problem.
Key words: distributed calculations, Monte-Carlo methods, AB-Initio simulation, Ruby, metaprogramming, sandbox, ObjectSpace, grid.

А.В. Дробнич, М.П. Дробнич, В.М. Ризак
Ужгородский национальный университет МОН Украины
88000, Ужгород, ул. Волошина, 54
e-mail: adrobnych@gmail.com

ПРОБЛЕМА ИСПОЛНЕНИЯ ВНЕШНЕГО КОДА В ГРІДАХ, ПОСТРОЕННЫХ НА ВИРТУАЛЬНЫХ МАШИНАХ RUBY

Декларируется проблема построения безопасной “песочницы” для выполнения внешнего кода в гридах на основе языка программирования Ruby. Анализируются пути решения этой проблемы.
Ключевые слова: распределенные расчеты, методы Монте-Карло, моделирование из первых принципов, Ruby, метапрограммирование, sandbox, ObjectSpace, грид.