RRABBIDS, un système de détection d'intrusion pour les applications Ruby on Rails

Romaric Ludinard, Loïc Le Hennaff, and Eric Totel

SUPELEC, Rennes, France, first_name.last_name@supelec.fr

Résumé RRABBIDS (Ruby on RAils Behavior Based Intrusion Detection System) est un système de détection d'intrusion au niveau applicatif pour des applications Web écrites avec le framework Ruby on Rails. Le but de cet IDS est de fournir un outil de détection des attaques contre les données dans le cadre d'applications Web. Cet IDS comportemental se fonde sur l'apprentissage d'invariants dynamiques pendant une phase d'observation du comportement normal de l'application pour définir son profil de référence. Dans une deuxième phase, ce profil est utilisé pour vérifier à l'exécution que le comportement normal de l'application est respecté, en instrumentant automatiquement l'application Web. Les premiers résultats sont encourageants et montrent que des classes d'attaques classiques contre les données (telles que des injections SQL, ou des modifications de paramètres de requêtes) sont détectées par les mécanismes mis en place.

1 Introduction

De nos jours, les serveurs informatiques forment les piliers des infrastructures de nos systèmes d'information. En tant que tels, ils sont aussi en première ligne vis-à-vis d'attaques visant à corrompre le système d'information. Aussi est-il nécessaire de mettre en place des outils visant à détecter ces attaques. Les systèmes de détection d'intrusion (ou IDS) sont utilisés dans ce but. Il s'agit de systèmes destinés à repérer des attaques ou des violations de la politique de sécurité à l'égard d'un système d'information. Ainsi cette entité informatique va observer le système et comparer les résultats de son observation par rapport à une base de connaissance afin d'établir un diagnostic.

On distingue classiquement deux types d'IDS. La différence fondamentale se situe dans la base de connaissance utilisée. Ainsi lorsque celle-ci constitue une connaissance d'attaques connues permettant d'amener l'entité observée dans un état illégal, on parle d'IDS par scénario. L'IDS observe alors les entrées du système et compare la séquence d'entrées observées aux séquences utilisés pour les attaques déjà connues et déclenche une alerte le cas échéant (c'est par exemple la méthode utilisée par l'IDS Snort qui repose sur une base de signatures des paquets réseau). Le deuxième type d'approche est dit "comportemental". La connaissance préalable du comportement normal de l'application constitue la base de

référence. Ainsi toute déviation sensible vis-à-vis du comportement attendu indique une intrusion potentielle. L'avantage de cette dernière approche réside dans sa capacité à détecter de nouvelles attaques sans pour autant devoir mettre à jour une base de signatures. Néanmoins, pour que la détection soit pertinente il faut être capable de se doter d'un profil de référence suffisamment complet et exact du comportement normal de l'application. C'est une approche de ce type que nous développons dans cet article.

Concernant les attaques contre les applications, la majorité des études menées considèrent qu'une attaque sur l'application entraîne une modification de son flot de contrôle (c'est-à-dire l'ordre dans lequel les actions sont réalisées). Ainsi, dans le cadre des applications web, l'observation de l'enchaînement des actions effectuées (l'enchaînement des pages par exemple) par l'application surveillée permet de détecter si le comportement observé dévie du comportement de référence. Cependant toutes les attaques n'altèrent pas nécessairement le flot de contrôle. Par exemple, dans le cas d'une injection SQL, l'enchaînement des actions réalisées par l'application est tout à fait légal. Cependant l'exécution de l'injection SQL peut permettre à l'attaquant d'interagir de manière incorrecte avec la base de données pour extraire de l'information confidentielle (violation de la confidentialité de certaines données) ou corrompre des données (violation de l'intégrité de l'application ou de ses données). Cela peut par exemple mener à l'obtention de privilèges supérieurs à ceux que l'utilisateur devrait avoir ou lui permettre d'effectuer des actions nécessitant des privilèges qu'il n'a normalement pas. Ceci illustre le fait que les attaques contre les données ne modifiant pas forcément le flot de contrôle de l'application ne doivent pas être ignorées.

Dans cet article, nous étudierons dans un premier temps la manière de caractériser le comportement normal de l'application dans le contexte des attaques contre les données, ensuite nous définirons les données critiques dans ce contexte, nécessaires à cette caractérisation. Dans une troisième partie, nous présenterons une implémentation de notre IDS comportemental dans le cadre d'applications web écrites avec le framework Ruby on Rails. Enfin, en dernière partie nous présenterons les résultats que nous avons d'ores et déjà obtenus avec notre outil.

2 Des contraintes invariantes de l'application

Pour fixer les idées, considérons une petite application web de type CMS. La majorité de ces applications utilisent une base de données contenant l'ensemble des billets ayant été rédigés par les administrateurs. Ce sous-ensemble d'utilisateurs dispose de privilèges pour modifier la base de données. Considérons encore que cette petite application jouet est perméable de diverses manières, en particulier vis-à-vis d'injections SQL. Le code utilisé dans l'application Ruby est donné dans le Listing 1.1.

Au lieu de fournir en entrée un identifiant correct, l'attaquant fournit une chaîne de caractère totalement différente des identifiants des utilisateurs

("'OR '1'='1"), et réussit une injection SQL, qui le mène à s'authentifier, sans pour autant avoir à fournir un couple identifiant/mot de passe correct.

Listing 1.1. Requête SQL pour l'authentification

```
user=User.authenticate(params[:user][:login], params[:user][:password])
```

Dans l'application concernée, le résultat de la requête SQL en comportement normal donne comme utilisateur l'utilisateur ayant pour identifiant *login*. Alors que dans le cas de l'injection SQL, l'identifiant du premier utilisateur retourné est forcément différent du *login* introduit. Ceci montre deux choses : en comportement normal, il existe un invariant qui est vérifié : $user == login$, alors que dans le cas de l'attaque, ce même invariant n'est pas vérifié.

Nous avons illustré ici notre approche pour le type d'attaque le plus commun qui existe pour les applications web, mais l'approche s'applique pour toutes les attaques qui visent à violer l'intégrité des données utilisées par l'application (comme la modification de paramètres de requêtes, la modification de données portées par des cookies, etc.). Le problème qui se pose est d'être capable de générer des invariants de manière automatique et sur des données pertinentes. C'est ce problème que nous abordons dans la section suivante.

3 Modèle de détection fondé sur les invariants

Nous proposons dans cette section de décrire l'approche que nous souhaitons utiliser pour la détection d'intrusion comportementale. Pour ce faire, nous allons construire un modèle de référence fondé sur les invariants issus du programme que l'on souhaite sécuriser. Notre détection se concentrera uniquement sur les attaques contre les données du programme. Comme nous l'avons vu précédemment, ces attaques exploitent du code valide de façon illégale de manière à faire sortir le programme de son exécution normale.

3.1 Données sensibles aux attaques

Le modèle de référence basé sur les invariants du programme repose sur la génération d'invariants caractérisant l'exécution normale du programme au travers des valeurs des variables du programme à différents instants de l'exécution. Cependant, il est bien évident que toutes les données de l'application ne sont pas pertinentes pour la construction de cet ensemble de référence.

Comme l'ont remarqué Sarrouy et al. [1] dans le cadre général d'applications quelconques, il semble tout d'abord nécessaire de prendre en compte

les données influençant les appels systèmes et leurs arguments. En effet, si une variable n'influence pas les appels systèmes ou leurs arguments, elle n'a que peu d'intérêt vis-à-vis de l'attaquant et donc par conséquent vis-à-vis de la détection d'intrusion. D'autre part, il faut aussi prendre en compte les données fournies par l'utilisateur ainsi que celles influencées par ces entrées. En effet, celles-ci constituent le seul moyen pour l'utilisateur d'interagir avec l'application et donc a fortiori d'attaquer l'application.

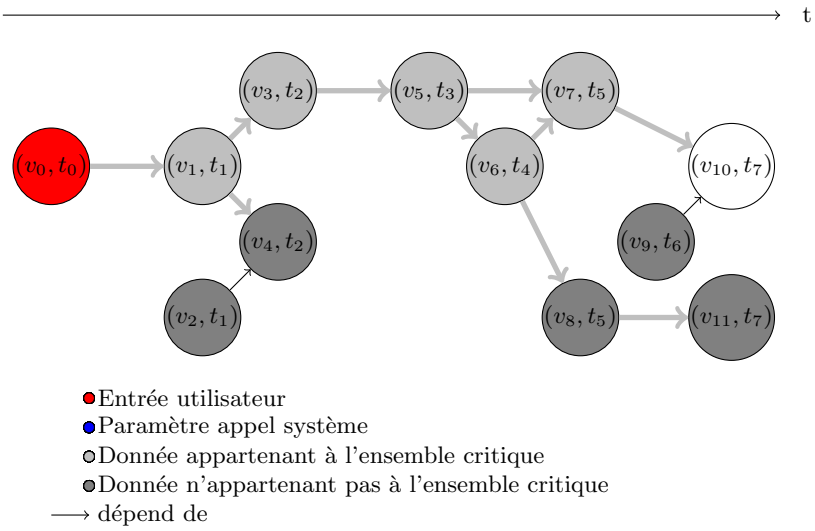

Figure 1. Ensemble des variables sensibles d'un programme

Au final, l'ensemble des données critiques pour le programme est donc situé dans l'union de l'ensemble des données influençant les appels systèmes et leurs arguments et de l'ensemble des données influencées par les entrées utilisateurs. Afin de réduire au maximum le nombre de données pour lesquelles on cherche à calculer des invariants, notre approche s'intéresse uniquement aux données influencées par les utilisateurs (voir la Figure 1). Cette approche en soit n'est pas nouvelle (elle est par exemple utilisée dans [2]) et consiste à *marquer* les données (*data tainting*) pour savoir si elles sont ou non dépendantes des entrées utilisateurs.

3.2 Génération automatique de contraintes

Daikon[3] est un outil développé par Michael Ernst pour la détection dynamique d'invariants. À partir de traces d'exécutions constituées de couples $(variable, valeur)$, Daikon est capable d'inférer des contraintes invariantes portant sur les données du programme qui a généré ces traces.

En particulier, Daikon peut exhiber des invariants sur une variable (constante, domaine de valeur, référence constante,...), ou sur un ensemble de valeurs (relation d'ordre entre deux variables, combinaison linéaire de variables, etc...). D'autre part, Daikon est capable d'appréhender certaines conditions temporelles, à savoir le lien entre les valeurs d'une même variable en entrée et en sortie d'une fonction, ou alors sur des conditions structurelles des données : il peut ainsi générer des invariants portant sur des objets ou sur des classes.

La découverte dynamique d'invariants se déroule en deux temps. Tout d'abord, Daikon possède un front-end qui instrumente le programme de manière à obtenir des informations sur ses variables internes, et la structure de celui-ci, de manière à générer un fichier de traces structuré que Daikon saura exploiter. L'instrumenteur est totalement dépendant du langage source puisqu'il doit être capable de s'interfacer avec le programme de manière à récupérer le maximum d'information (structure et valeurs) sur les données qui nous intéressent. Une fois ces traces collectées, on peut les fournir à Daikon lui-même. Le moteur d'inférence qui est utilisé pour extraire des invariants des traces d'exécution est totalement indépendant de l'instrumenteur.

4 Mise en œuvre avec Ruby on Rails

Nous allons détailler dans cette section les principales technologies et techniques mises en oeuvre dans le cadre de notre implémentation d'un système de détection d'intrusion comportemental.

4.1 Ruby et Ruby on Rails

Présentation de Ruby on Rails Ruby on rails [1] (ou RoR) est un framework web libre permettant le développement rapide d'applications web. Il est connu pour certains sites qui n'ont pas hésité à l'adopter, tels que Twitter [2] ou GitHub [3]. Ruby on Rails offre de nombreuses facilités pour implémenter notre approche.

Listing 1.2. Lecture de l'entrée standard et propagation du flag *tainted*

```
x=gets
x.tainted? -> True

y="toto"
y.tainted? -> False
y=x
y.tainted? -> True
```

1. http ://rubyonrails.org/
2. http ://twitter.com/
3. https ://github.com/

Identification des variables critiques Comme nous l'avons dit précédemment, la création de notre modèle de référence débute par l'identification des variables critiques de l'application. Le langage Ruby offre de manière native une fonctionnalité permettant d'identifier les variables du programme dont la valeur est influencée par les entrée utilisateur. En effet, tout objet Ruby contient un flag *tainted* positionné automatiquement dès que la valeur de l'objet dépend d'une entrée utilisateur. La valeur du flag est obtenue au moyen de la méthode *tainted ?* présente dans chaque objet. D'autre part, la propriété *tainted* se propage par transitivité. Ainsi, si une variable x est marquée *tainted*et si on initialise une autre variable y à partir d'un calcul faisant intervenir la valeur de x, celle-ci sera marquée *tainted* aussi, comme on peut le voir dans le listing 1.2.

Cependant, le flag *tainted* ne se propage pas par inclusion structurelle. En effet, si un objet contient un champ *tainted*, Ruby ne marque pas l'objet en question comme *tainted*. Il est donc nécessaire de parcourir en profondeur l'ensemble des objets présents en mémoire qui nous intéressent, à savoir les variables locales, d'instance et les variables de classes afin d'extraire ceux qui permettent d'atteindre un champ *tainted* et ainsi les marquer eux aussi comme *tainted*. Cette opération est rendue possible grâce aux grandes capacités d'introspection offertes par Ruby.

Collecte des données sensibles à l'exécution Ruby on Rails permet de créer des plugins pour une application. C'est par ce moyen que nous allons pouvoir créer nos traces d'exécution en interceptant l'exécution de l'interprétation du programme à chaque ligne. Lors de l'initialisation de l'application, Ruby on Rails lit automatiquement les fichiers *init.rb* des éventuels plugins en vue de leur intégration au sein de l'application. Un plugin RoR n'est autre qu'un fragment de code placé à l'endroit adéquat dans l'arborescence de l'application, il ne modifie donc pas le code de l'application, toute la réalisation que nous avons menée est indépendante de celle-ci. Nous allons voir plus en détail les deux plugins que nous avons réalisés, pour notre système de détection d'intrusion. Pour collecter les données, il est nécessaire d'intercepter le code Ruby qui s'exécute. Une fois encore, Ruby nous fournit toutes les fonctionnalités dont on a besoin. En effet au travers de son module *Kernel*, Ruby offre la fonction *set_trace_func*. Celle ci, lorsqu'elle est appelée avec un paramètre différent de *nil* permet de suspendre l'exécution courante (exécution d'instruction, appel de méthode, initialisation d'objet, ...) afin d'exécuter le code qu'on lui passe en paramètre avant de reprendre l'interprétation du code au point où elle avait été suspendue. Ainsi, grâce à cette méthode, on peut exécuter à chaque instant, un parcours en profondeur sur les objets en mémoire, de manière a obtenir tous les objets marqués comme *tainted* et ainsi générer un fichier de traces contenant les couples $(nomdevariable, valeurdevariable)$.

Il faut noter que dans une utilisation classique d'un front-end de Daikon, il n'y a pas, sauf cas particulier de relations entre les variables dans le temps. C'est-à-dire qu'on ne peut pas avoir d'invariants tels que 'la variable v1 a l'instant t0 est égale à la variable v2 à l'instant t1". Or dans

Listing 1.3. Exemple de trace fournie à Daikon

```
variable 10_1_@_session[user].@attributes[name]
  var-kind variable
  dec-type String
  rep-type java.lang.String
  enclosing-var @_session[user].@attributes
  comparability 0
variable 10_1_@_params[user].[name]
  var-kind variable
  dec-type String
  rep-type java.lang.String
  enclosing-var @_session[user].@attributes
  comparability 0
...
10_1_@_session[user].@attributes[name]
"evette"
1
10_1_@_params[user][name]
"evette"
1
```

Listing 1.4. Exemple d'invariant généré par Daikon

```
10_1_@_params[user][name] == 10_1_@_session[user].@attributes[name]
```

notre approche, puisqu'on intercepte l'exécution du programme ligne à ligne, on introduit implicitement la notion d'évolution du temps dans les traces qu'on génère (l'avancement d'une ligne à une autre correspondant à l'incrémentation du temps), en nommant les variables qui sont tracées en fonction de la ligne d'exécution (cf. Listing 1.3). Ceci signifie que Daikon pourra nous fournir des invariants plus pertinents que ceux qu'il produit en utilisant les front-ends associés aux langages pour lesquels il est normalement conçu.

4.2 Invariants Générés

Une fois que l'on a collecté suffisamment de traces, il suffit de les fournir en entrée de Daikon, et celui-ci va fournir en retour un ensemble d'invariants de l'application. Les types d'invariants que nous avons décidés de chercher sont les suivants :

- la valeur d'une variable (qui peut être une référence sur un objet) est constante ;
- la donnée est dans un ensemble fini d'éléments ;
- relation d'ordre entre variables comparables (inférieur, supérieur) ;
- relation d'égalité entre deux variables.

Il faut noter que la pertinence des invariants générés dépend étroitement de l'exhaustivité de la phase d'apprentissage. Nous avons expérimenté cette approche sur une application de e-commerce développée par Kereval [4] en Ruby on Rails dans le cadre du projet ANR DALI [5], et pour laquelle

4. http ://www.kereval.com
5. http ://dali.kereval.com

nous avons réalisé cet apprentissage. Cette phase consiste à automatiser le parcours de l'application web de manière à être le plus exhaustif possible dans son fonctionnement normal. Ceci s'apparente à effectuer un test fonctionnel de l'application, et nous avons donc utilisé un outil de tests d'applications web nommé Selenium [6]. Ceci nous a permis d'automatiser la collecte de traces d'exécution sur l'application.

4.3 Vérification des invariants à l'exécution

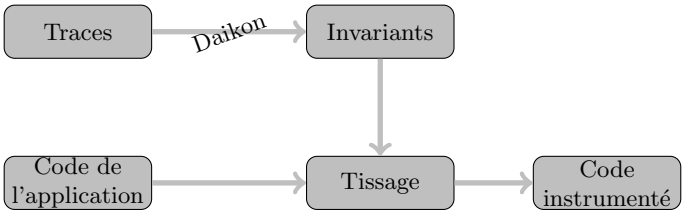

Figure 2. Les différentes étapes de la réalisation

La mise en place de l'IDS nécessite de tisser les invariants dans le code de l'application afin de vérifier à l'exécution si le comportement normal de l'application, appris pendant la phase d'apprentissage, est bien respecté (Figure ci-dessus).

Listing 1.5. Code Ruby initial

```
def signup
  if request.post?
    params[:user][:admin] =
      User.first(:conditions => {:admin => true}).nil? ? true : false
    @user = User.new(params[:user])

    if @user.save
      # whole user is stored in the session
      session[:user] = User.authenticate(@user.login, @user.password)
      flash[:notice] = 'You have been successfully signed up.'
      redirect_to :controller => '/user/home', :action => 'index'
    end
  end
end
```

Les invariants que nous obtenons portent sur des relations (identifiées précédemment) sur la valeur d'une ou plusieurs variables dans le temps. Afin de vérifier les invariants à l'exécution, il faut (1) être capable de retrouver ces valeurs et (2) insérer dans le code final de l'application la vérification de l'invariant. Ceci nous mène à tisser dans le code final deux

6. http ://seleniumhq.org/

Listing 1.6. Code Ruby instrumenté

```
def signup
  if request.post?
    params[:user][:admin] =
      User.first(:conditions => {:admin => true}).nil? ? true : false
    @user = User.new(params[:user])

    if @user.save
      # whole user is stored in the session
      session[:user] = User.authenticate(@user.login, @user.password)
#Auto-generated
@@vars["10_1_@_params[user][name]"] = @_params[user][name]

#Auto-generated
@@vars["10_1_@_session[user][attributes][name]"] =
  10_1_@_session[user].@attributes[name]

#Auto-generated
assertEquals(@@vars["10_1_@_params[user][name]"],
  @@vars["10_1_@_session[user].@attributes[name]"])

      flash[:notice] = 'You have been successfully signed up.'
      redirect_to :controller => '/user/home', :action => 'index'
    end
  end
end
```

types d'instructions : la sauvegarde de valeurs de variables afin qu'elles soient accessibles lorsqu'on en aura besoin, et la vérification de l'invariant lui-même à la ligne adéquate.

Listing 1.7. Exemple d'alerte

```
<Alert>
<To>
 <Request>http://localhost:3000/signup</Request>
 <Controller>users</Controller>
 <Action>signup</Action>
 <Invariant>
   <Description>
   10_1_@_params[user][name]==10_1_@_session[user].@attributes[name]
   </Description>
   <Line>10</Line>
   <Value1>'or '1'='1</Value1>
   <Value2>secret</Value2>
   <Datetime>Tue Jun 15 17:54:45 +0200 2010</Datetime>
 </Invariant>
</To>
</Alert>
```

On observe sur la Figure 1.6 ces deux étapes. Dans cet exemple, on détecte une injection SQL similaire à celle qui a été présentée Section 2, et une alerte est levée (Listing 1.7).

Dans la pratique, les relations entre variables peuvent porter entre deux exécutions successives de requêtes à l'application, de sorte qu'on ne retient dans le temps qu'une fenêtre de variables de deux exécutions. La sauvegarde des variables est réalisée sur le serveur entre ces deux

exécutions, et les invariants portant sur des variables entre deux exécutions successives peuvent être vérifiés.

5 Résultats

Comme dit précédemment, nous avons expérimenté notre approche sur une application de e-commerce développée dans le cadre du projet ANR DALI [7]. Sur cette application, nous avons pu générer plus de 10000 invariants qui ont été tissés dans son code.

Afin de tester si les mécanismes de détection détectent réellement des attaques, nous avons réalisé quatre types d'attaques contre les données :

- les injections SQL que nous avons réalisées mènent à la levée d'une alerte ;
- nous avons altéré les paramètres de requêtes vers l'application, et ces modifications ont été détectées ;
- nous avons modifié, grâce à un plugin Firefox, des cookies contenant des données de l'application, et ces modifications ont été détectées ;
- nous avons réalisé des attaques de type cross-site scripting. Ces attaques, bien qu'elles concernent des données de l'application, ne sont en fait pas utilisées par l'application serveur, mais par le navigateur client, et n'ont donc pas d'impact sur le fonctionnement de l'application côté serveur. Par conséquent, ces attaques n'ont pas été détectées.

Ces premiers résultats sont encourageants et seront, nous l'espérons, confirmés par une phase d'évaluation intensive de l'IDS applicatif qui sera menée dans le cadre du projet ANR DALI.

6 Conclusion

Ces travaux montrent qu'il est possible de générer automatiquement dans une application des mécanismes de détection d'erreurs qui permettent la détection d'attaques contre les données de l'application. De plus, ces mécanismes qui ont été expérimentés initialement sur des binaires, s'avèrent efficaces sur des applications aussi différentes que des applications web. Par contre, afin de pouvoir générer des invariants pertinents, il faut être capable de générer des traces des valeurs des variables utilisées par l'application pendant son exécution. Afin de pouvoir avoir un niveau d'introspection suffisant pour effectuer ces observations, nos expérimentations nous ont porté vers un langage et un environnement de développement fort apprécié : Ruby on Rails. L'application développée dans le cadre du projet afin de tester la faisabilité de l'approche nous a montré que les attaques classiques trouvées dans les applications web sont correctement détectées par notre IDS, à l'exception des attaques par cross-site scripting qui sont des attaques contre le client, et non contre le serveur. Enfin, les outils permettant de réaliser cet IDS dans le cadre d'applications Ruby on Rails seront prochainement disponibles en libre accès sur le web.

7. http ://dali.kereval.com

Références

1. Sarrouy, O., Totel, E., Jouga, B. : Application data consistency checking for anomaly based intrusion detection. In : The 11th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2009), Lyon (November 2009)
2. Cavallaro, L., Sekar, R. : Anomalous taint detection. Technical report, Secure Systems Laboratory, Stony Brook University (2008)
3. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C. : The daikon system for dynamic detection of likely invariants. Science of Computer Programming **69** (2007) 35–45