Détection d'attaques contre les données dans les applications web

Loïc Le Henaff

Détection d'attaques contre les données dans les applications web

Loïc Le Henaff
Encadreur : Eric Totel
Equipe SSIR, Supélec

Juin 2010

Master Recherche en Informatique 2009-2010

Résumé

Un système de détection d'intrusion (IDS) est un système permettant de détecter des activités anormales ou suspectes dirigées contre un système d'information. Plusieurs types d'IDS existent : certains permettent d'analyser un réseau, d'autres un hôte tandis que d'autres encore sont hybrides. Par ailleurs, deux grandes approches coexistent dans la détection d'intrusion. La première est l'approche par signature. La deuxième est l'approche comportementale. Nous nous intéressons ici à la détection d'intrusion comportementale au niveau applicatif. Cette approche est divisée en deux grandes phases. Tout d'abord, un modèle du comportement normal de l'application est créé. Ensuite, ce modèle est utilisé pour détecter des attaques à l'exécution.

Le stage mené dans l'équipe SSIR à Supélec cette année s'intéresse à la détection d'attaques contre les données dans le contexte particulier des applications web. Ce type d'application est de plus en plus utilisé tandis que le nombre d'attaques augmente de pair. Notre objectif est de modéliser le comportement de l'application par des invariants sur les variables critiques. A l'exécution, si un des invariants est violé, nous considérons que nous sommes en présence d'une intrusion. Nous avons implémenté notre approche et obtenu des résultats encourageants qui ont permis de détecter les attaques connues contre notre application test.

Abstract

An intrusion detection system (IDS) is a system allowing to detect anomalous activities against an information system. Several kinds of IDS exist : some of them are applied at a network level, others deal with the activity on a host whereas others mix the two approaches. In addition, there are two different approaches in intrusion detection. The former is misuse detection. The latter is anomaly detection. We are interested in anomaly detection at the application level. This approach is divided in two parts. Firstly, we model the application normal behaviour. Then, we use this model to detect attacks when the application is executed.

This year, in SSIR team in Supelec, we investigated intrusion detection against non-control-data in the specific context of web applications. This kind of application is more and more used whereas attacks against it are growing. We learn application normal behaviour by looking for properties, called invariants, on critical variables. When we execute the application, if one of the invariants is broken, we will conclude an intrusion occured. We implemented our approach and obtained results that proved to be successful in detecting attacks against our test application.

Table des matières

Remerciements **3**

1 Introduction **4**

2 Etat de l'art **6**
2.1 Généralités sur la détection d'intrusion 6
2.1.1 Les sources de données 6
2.1.2 Les méthodes de détection 7
2.1.3 Les autres caractéristiques 8
2.2 Détection d'intrusion au niveau applicatif 8
2.2.1 L'approche en boîte noire 8
2.2.2 L'approche en boîte grise 9
2.2.3 L'approche en boîte blanche 11
2.3 Modèle de détection fondé sur des invariants 13
2.3.1 Construction du modèle par analyse statique 13
2.3.2 Construction du modèle par analyse dynamique 14
2.4 Spécificités des applications web 15

3 Contexte de travail **18**
3.1 Objectifs 18
3.2 Technologies et outils utilisés 19
3.2.1 Ruby 19
3.2.2 Ruby on Rails 20
3.2.3 Daikon 20
3.3 Exemples d'attaques contre les applications web 21
3.3.1 Les injections SQL 21
3.3.2 L'altération des requêtes 22
3.3.3 Les attaques XSS 23

4 Détection d'attaques contre les données dans les applications web **24**
4.1 Présentation du modèle 24
4.2 Définition des données utiles à la construction du modèle . . 25
4.3 Log des variables sensibles aux intrusions 27

4.3.1 Log à l'exécution d'une action 27
4.3.2 Log entre deux actions consécutives 28
4.4 Génération des invariants 28
4.5 Détection d'intrusions 31

5 Implémentation **32**
5.1 Tracer et extraire les variables avec Ruby 32
5.2 Interfaçage avec Ruby on Rails 34
5.3 Apprentissage 35
5.4 Génération des invariants 36
5.5 Détection d'intrusions 38

6 Résultats **40**

7 Travaux futurs **42**

8 Conclusion **43**

Remerciements

Je souhaite tout d'abord remercier mon encadreur Eric Totel pour son accueil, son aide et sa disponibilité tout au long de ce stage. Je remercie toute l'équipe SSIR de Supélec pour leur accueil et leur bonne humeur quotidienne qui m'ont permis de réaliser mon stage dans un environnement convivial. Je remercie également Romaric Ludinard pour sa relecture de mon rapport et ses commentaires. Enfin, je tiens à remercier les communautés Ruby, Ruby on Rails et Daikon pour leurs réponses à mes interrogations.

Chapitre 1

Introduction

Dans l'environnement économique actuel, les systèmes d'information jouent un rôle primordial et abritent souvent des informations confidentielles. La sécurité est ainsi devenue en quelques années une problématique centrale. Paradoxalement, les systèmes d'information que nous utilisons tous les jours sont touchés par de nombreuses vulnérabilités. Si on emprunte un point de vue traditionnel, il devrait être possible de résoudre ces failles de sécurité en utilisant des méthodes formelles et de meilleurs logiciels. Bien qu'en théorie concevable, on rencontre rapidement les limites de cette proposition dans la pratique. Les ordinateurs et les systèmes d'information évoluent sans cesse, changent de configuration, sont reliés à des réseaux, etc. Les vérifications formelles dans un environnement aussi dynamique que celui-ci sont inutilisables. Pour répondre aux problématiques de sécurité, de nombreux outils ont ainsi été proposés : mise en place de politique de sécurité, anti-virus, protocoles cryptographiques, systèmes de détections d'intrusion (IDS)... C'est de ces derniers systèmes dont il est question dans ce document.

Un système de détection d'intrusion est un système qui permet de détecter des activités anormales ou suspectes dirigées contre un système d'information. Plusieurs types d'IDS existent, certains permettent d'analyser un réseau, d'autres un hôte tandis que certains sont hybrides. Par ailleurs, de nombreuses approches coexistent aujourd'hui, certaines se focalisant sur certaines problématiques ou types d'attaques particuliers. L'enjeu réside dans le choix de propriétés suffisamment discriminantes permettant la détection d'attaques réelles. En effet, un des problèmes récurrents dans les IDS aujourd'hui est l'important taux de faux positifs, c'est à dire de fausses détections.

Une approche populaire de la détection d'intrusion a été initiée par les travaux de Forest et al. [1, 2], qui consiste à surveiller l'enchaînement des appels système d'un programme et à détecter les séquences qui pourraient être incorrectes. Il s'est avéré que cette approche est efficace pour détecter les attaques qui modifient le flot de contrôle d'un programme, mais est insuffisante lorsqu'on considère des attaques contre les données, légitimes d'un

point de vue du système. C'est de ces attaques contre les données dont il a été question dans ce stage.

Durant le stage, nous avons continué les travaux initiés dans l'équipe SSIR à Supelec. L'approche a consisté à définir le comportement légitime d'un programme par des contraintes - ou invariants - sur les variables sensibles et à détecter par la suite des infractions. Nous nous sommes focalisés cette année sur les attaques contre les données dans les applications web. Nous avons pu obtenir des premiers résultats encourageants. Par ailleurs, ce stage s'inscrit dans le cadre du projet ANR, nommé DALI, en partenariat avec Kereval, Telecom Bretagne et le LAAS-CNRS. Ce projet a pour objectif de spécifier et développer de nouveaux mécanismes pour détecter des intrusions sur les sites Internet. En particulier, l'entreprise Kereval a développé une application vulnérable, que nous avons pu utiliser pour implémenter notre approche en Ruby et Ruby on Rails.

Dans la première partie de ce document, nous dresserons un état de l'art de la détection d'intrusion. Nous présenterons quelques généralités telles que la classification des IDS, leurs caractéristiques et leurs fonctionnements. Nous poursuivrons ensuite sur les différentes approches réalisées à ce jour au niveau d'une application, en les classant dans trois grandes catégories : les approches dites « boîtes noires », « boîtes grises », et « boîtes blanches ». Dans une deuxième partie, nous décrirons le contexte de ce stage et ses objectifs avant de rentrer dans le vif du sujet dans une troisième partie où nous présenterons notre démarche en détails. Nous continuerons sur une description de notre implémentation et par un commentaire des résultats obtenus. Nous verrons dans une dernière partie les axes de recherche futurs avant de conclure.

Chapitre 2

Etat de l'art

Nous présentons dans ce qui suit un état de l'art sur la détection d'intrusion. Nous allons décrire dans une première partie quelques généralités sur la détection d'intrusion et voir quelles sont les caractéristiques essentielles pour classer les différents types d'IDS. Nous nous focaliserons dans la suite sur la détection d'intrusion au niveau applicatif uniquement et nous verrons les différentes approches existantes. Nous verrons ensuite différents modèles de détection fondés sur les invariants d'un programme et quelques approches de la détection d'intrusion pour les applications web, domaine concerné par le sujet de ce stage.

2.1 Généralités sur la détection d'intrusion

Un système de détection d'intrusion est un système destiné à repérer des attaques ou des violations de la politique de sécurité à l'égard d'un système d'information. La détection d'intrusion est l'étape qui consiste à surveiller les évènements arrivant sur un réseau ou sur une machine hôte et à identifier une menace. En résultat, un IDS peut produire des notifications à l'administrateur de sécurité, produire des rapports et parfois, tenter de prévenir l'attaque. De nombreuses approches ont été présentées au fil des ans, et certaines d'entre elles sont encore aujourd'hui du domaine de la recherche. Il existe plusieurs manières pour caractériser un IDS ; nous identifions ci-dessous les grandes classes.

2.1.1 Les sources de données

Plusieurs sources de données peuvent être envisagées pour surveiller un système informatique. Une source particulière doit être suffisamment riche en informations à analyser pour que l'IDS ait un faible taux de faux négatifs (les attaques non détectées). Par ailleurs, les informations doivent être raisonnablement précises de manière à réduire le taux de faux positifs (les

fausses détections).

Les paquets circulant sur un réseau forment une bonne source de données à analyser pour la détection d'intrusion. L'entête et le payload d'un paquet sont ainsi parcourus en vue de trouver des comportements malveillants. Snort[1], IDS très répandu, se base sur cette source pour analyser les intrusions sur un réseau.

Une autre source peut être les données fournies par le système d'exploitation comme les appels systèmes [1, 2]. Les données applicatives, comme les variables d'un programme, peuvent également être utilisées pour mettre en oeuvre des IDS sur certains types d'applications, comme les applications web.

2.1.2 Les méthodes de détection

Les méthodes de détection d'intrusion sont divisées en deux grandes écoles : l'approche par scénarios et l'approche comportementale.

Les IDS basés sur une approche par scénario utilisent la modélisation du comportement jugé interdit de l'entité surveillée. En effet, une intrusion peut être considérée comme une suite d'évènements révélatrice d'une attaque. Ainsi, ces IDS utilisent une base de signatures - ou scénarios - identifiant les attaques et vulnérabilités connues, et les comparent à un ensemble de données à surveiller (paquets réseaux, logs systèmes...). Cette approche est relativement similaire à ce qui est employé dans la détection virale. L'inconvénient majeur de cette approche est qu'elle nécessite d'avoir une connaissance préalable quant à la nature des intrusions possibles. Les IDS basés sur une approche par scénario ne seront pas en mesure de détecter de nouvelles intrusions, puisque les signatures caractéristiques ne seront pas présentes dans la base de données. L'efficacité de cette technique repose donc totalement sur la capacité à entretenir une base de données des signatures à jour. Par ailleurs, ces IDS détectent en général de très nombreux faux positifs car l'écriture de signatures réellement discriminantes est difficile.

En contraste, *les IDS fondés sur une approche comportementale* modélisent le comportement normal d'une entité. La nature d'une intrusion est inconnue dans cette approche. Néanmoins, une intrusion provoquera un comportement différent du système, par rapport à celui observé normalement. Ainsi, toute déviation par rapport au modèle de référence indique une possible intrusion. L'avantage de cette approche est qu'elle permet de détecter les nouvelles attaques sans intervention additionnelle sur l'IDS. L'enjeu de cette technique est de créer un modèle de référence suffisamment complet et précis pour détecter les déviations et réduire le nombre de faux positifs. Nous verrons dans le deuxième chapitre certaines techniques employées à ce sujet. En règles générales, il y a deux étapes dans la détection d'intru-

[1]Snort est un IDS réseau open source. Plus d'informations sur http ://www.snort.org.

sion comportementale. La première consiste à créer une base de données du comportement normal d'une entité, c'est à dire, le modèle de référence. La deuxième étape consiste à utiliser la base de données précédemment construite pour surveiller le comportement du système.

2.1.3 Les autres caractéristiques

Plusieurs autres caractéristiques peuvent être employées pour classifier les IDS. Par exemple, plusieurs architectures sont possibles. Un IDS peut être centralisé, c'est-à-dire que son exécution est réalisée sur une seule machine, ou bien distribué sur plusieurs machines. Nous pouvons également caractériser le comportement d'un IDS après la détection d'une intrusion. En général, il fournit un rapport d'intrusion ou une alerte à l'administrateur de la sécurité. Des travaux ont toutefois été publiés dans le cadre d'IDS défensif (aussi nommé Intrusion Prevention System - IPS), qui vont adopter un comportement adéquat sur le système en fonction de l'intrusion détectée.

Nous avons vu dans ce chapitre les différents types d'IDS et quelles sont leurs caractéristiques. Dans le chapitre suivant, nous allons nous focaliser sur les solutions existantes au niveau applicatif, en faisant un état de l'art des solutions de détection d'intrusion comportementales actuelles.

2.2 Détection d'intrusion au niveau applicatif

Nous allons voir dans ce chapitre quelques approches populaires de la détection d'intrusion comportementale au niveau applicatif. Nous avons choisi d'organiser cet état de l'art selon trois visions différentes : une approche «boîte noire», «boîte grise» et «boîte blanche». Ces trois approches de la détection d'intrusion jouent sur des niveaux d'analyse différents, chacun d'eux étant basé sur le type d'informations disponibles pour construire le modèle de référence.

2.2.1 L'approche en boîte noire

Les approches de type boîte noire n'utilisent aucune information interne du programme, d'où le nom. Aujourd'hui, elles se basent pour la plupart sur l'analyse des enchaînements des appels systèmes des processus, donc sur des informations externes au programme. Historiquement, nous devons ces premiers travaux à Forest et al. dans [1] et développés dans [2]. L'expérience a en effet montré que de courtes séquences d'appels système génèrent une signature stable pour modéliser le comportement normal d'un processus par rapport à son environnement. Précédemment à ces travaux, des approches modélisant le comportement normal d'un utilisateur ont été proposées [3, 4]. Ces dernières modélisent des profils en analysant les logs du système. Il s'est avéré que l'analyse des appels système est plus avantageuse en tout point de

vue. En effet, le nombre de comportements différents d'un programme est borné, contrairement à celui d'un utilisateur qui est susceptible de générer un très grand nombre d'actions différentes.

Dans les travaux de Forest et al., seuls les appels système et leur ordre temporel sont considérés, les arguments étant ignorés. La phase d'apprentissage construit une base de données des séquences normales. Les nouvelles exécutions sont par la suite comparées aux traces présentes dans la base de données. Si certains appels système caractéristiques pour une exécution donnée n'apparaissent pas dans la séquence surveillée, on considère qu'il s'agit d'une attaque. La principale interrogation liée à cette démarche concerne la longueur des séquences d'appels système à analyser. En effet, il s'agit de trouver un juste milieu entre la précision, où des séquences longues semblent plus avantageuses, et le stockage de ces dernières, auquel cas on recherche plutôt à avoir des séquences courtes.

Cette technique permet de détecter des attaques visant à modifier le flot de contrôle de l'application. Par exemple, un attaquant peut cibler une adresse de retour sur la pile et la modifier de manière à exécuter du code malveillant injecté quelque part dans la mémoire. Cela aura pour effet de provoquer des appels système illégaux la plupart du temps.

Cette approche a cependant des faiblesses. En effet, elle est sujette aux attaques par mimétisme, dans lesquelles un attaquant imite le comportement attendu de l'application surveillée avec pour objectif d'agir sur le flot de contrôle de l'application. Cela aura pour conséquence de générer des appels système valides du point de vue de l'IDS et donc de ne pas détecter l'attaque. Cette approche est également encline aux attaques contre les données. Ces dernières visent l'intégrité des données sans forcément modifier le flot de contrôle de l'application surveillée. Pour tenter de pallier ces attaques, différents travaux ont proposé d'utiliser des informations additionnelles sur l'état interne de l'application et de les ajouter au modèle basé sur les séquences d'appels système. Ces approches sont appelées approches en boîte grise.

2.2.2 L'approche en boîte grise

Tout comme l'approche en boîte noire, l'approche en boîte grise est fondée sur les séquences d'appels système. Cependant, elle extrait des informations additionnelles du processus, notamment en utilisant la mémoire. Comme nous l'avons évoqué précédemment, l'approche en boîte noire est vulnérable à plusieurs types d'attaques. Les approches en boîte grise se veulent plus complètes de manière à accroître la difficulté d'exploitation de ces attaques. Nous allons voir ci-dessous un état des lieux de cette approche.

L'expérience a montré que la présence d'une attaque se manifeste souvent dans les arguments des appels systèmes. Se basant sur ce constat, Kruegel et al. [5] proposent de prendre en compte les arguments des appels système

pour améliorer la technique introduite par Forest et al. [1, 2]. Pour cela, les arguments des appels système sont analysés suivant plusieurs modèles et chacun de ces modèles est instancié pour chaque appel système. Parmi ces modèles, nous pouvons citer la distribution de la taille d'un argument, la distribution de ses caractères, sa structure grammaticale et la présence de valeurs énumérées. Les auteurs démontrent ainsi qu'il est possible de réduire considérablement la capacité d'un attaquant à contourner le système de détection. Néanmoins, leur travail est fondé sur deux hypothèses importantes. La première étant que si une attaque est menée, alors elle aura un impact sur les arguments des appels système. La deuxième hypothèse impose quant à elle qu'un paramètre utilisé pour mener une attaque diffère substantiellement des valeurs observables lors d'une exécution normale de l'application. Si ces deux conditions ne sont pas vérifiées, alors la capacité de détection de cette approche n'est évidemment plus assurée.

Une approche complémentaire à celle décrite précédemment a été étudiée par Mutz et al. [6]. Au lieu d'avoir une instanciation des modèles pour chaque appel système, les auteurs proposent de considérer les différentes phases d'un programme. Par exemple, les arguments d'une phase d'initialisation sont fortement susceptibles d'être différents de ceux d'une phase de production. Cette différenciation du comportement du programme va être réalisée en prenant en compte le contexte d'un appel système, c'est à dire sa pile d'appel. Les auteurs démontrent alors que la sensibilité ajoutée grâce au contexte d'un appel système offre une meilleure performance de détection. En particulier, certaines attaques contre les données ont pu être détectées alors que l'approche se basant uniquement sur les arguments des appels système les avait ignorés. Les auteurs ont démontré également que la sensibilité au contexte d'appel apporte une granularité plus fine et permet ainsi de réduire le nombre de faux positifs.

Dans l'article [7], Gao et al. introduisent un nouveau modèle de détection, nommé graphe d'exécution. L'objectif de ce travail est de se rapprocher le plus possible d'une analyse de type boîte blanche (que nous décrirons dans la partie suivante) sans que cela requière une analyse statique des sources du programme. Par conséquent, les auteurs proposent cette solution lorsqu'une approche boîte blanche n'est pas envisageable. Par exemple, lorsqu'on ne possède pas les codes sources du programme (l'analyse statique sur des fichiers binaires étant complexe) ou lorsque le programme est protégé par obfuscation. Le modèle de référence créé se veut le plus exhaustif possible pour correspondre au mieux au graphe de flot de contrôle de l'application. Toutefois, certaines branches du programme sont susceptibles de ne pas être connues si leurs exécutions ne sont présentes dans aucune observation. Ainsi, pour créer le graphe d'exécution, la structure des appels de fonctions est extraite des observations. Enfin, les auteurs assurent deux propriétés intéressantes à leur modèle. Premièrement, les séquences d'appels système acceptées par le graphe d'exécution sont un sous ensemble de celles acceptées

par le graphe de flot de contrôle du programme. Deuxièmement, le langage accepté par le graphe d'exécution est maximal par rapport aux données d'entrainement fournies.

Nous avons vu dans ce chapitre que l'approche boîte grise améliore l'approche boîte noire, en lui associant des informations additionnelles sans toutefois utiliser les sources du programme. En effet, nous avons soit considéré l'environnement d'exécution, soit considéré les observations de l'exécution de l'application. Toutefois, des attaques sont toujours possibles comme les attaques contre les données. Nous allons voir dans le chapitre suivant comment utiliser les sources du programme pour améliorer la détection d'intrusion.

2.2.3 L'approche en boîte blanche

Dans une approche en boîte blanche, toutes les informations présentes dans les sources du programme peuvent être utilisées pour construire un modèle de détection d'intrusion au niveau applicatif. En effet, nous considérons ici le code du programme que nous allons analyser par analyse statique ou dynamique afin d'en dériver un modèle approprié. Ainsi, comme nous allons le voir dans la suite, cette approche permet de détecter à la fois des attaques contre le flot de contrôle de l'application et des attaques contre les données. En particulier, des attaques contre les données peuvent avoir pour conséquence de modifier le flot de contrôle de l'application.

Attaques contre le flot de contrôle

Wagner et al. [8] montrent comment une analyse statique du code source peut être utilisée pour construire un modèle du comportement normal de l'application. Cette technique se focalise sur la détection d'attaques contre le flot de contrôle de l'application, comme nous avons pu le voir dans l'approche en boîte noire par exemple. Pour réaliser ce travail, les auteurs proposent de dériver les spécifications du programme à partir d'une analyse statique de son code source. Ainsi, le comportement de l'application est modélisé par un système de transition d'états. Durant la phase de détection, si une séquence d'appels système est incompatible avec ce système de transition, on peut alors considérer qu'il s'agit d'une attaque. L'objectif de ce travail a été notamment d'automatiser la construction du modèle. Pour cela, les auteurs proposent différentes approches. La dernière retenue est fondée sur la génération d'un automate à pile, exprimant les séquences valides d'appels système. Des optimisations ont également été proposées pour affiner la détection, comme la prise en compte des arguments des appels système. Les résultats obtenus convergent vers ceux observés dans l'approche en boîte grise. Cependant, cette technique ne permet pas de détecter des attaques contre les données qui ne modifient pas le flot de contrôle de l'application.

Attaques contre les données

Contrairement aux approches en boîte noire et en boîte grise, l'approche en boîte blanche a les capacités d'agir de manière efficace envers les attaques contre les données. En particulier, Chen et al. démontrent dans l'article [9] que ces attaques sont une véritable menace, bien qu'elles soient plus difficiles à mettre en oeuvre. Elles reposent en effet sur la connaissance de la sémantique du programme ciblé, imposant à l'attaquant une étude approfondie de ses vulnérabilités. Ces attaques ciblent en particulier les fichiers de configuration et les paramètres d'entrée de l'application, ainsi que les données concernant les prises de décision (par exemple une attaque contre la valeur d'un booléen pour accéder à une branche particulière d'une conditionnelle). Ainsi, des informations sur le flot de données du programme peuvent être intéressantes pour détecter ce type d'attaques.

Motivés par le travail de Chen et al. [9], Cavallaro et al. [10] proposent une approche qui allie la technique de taint tracking et la détection d'intrusion comportementale. Le principe du taint tracking est de marquer toutes les variables susceptibles d'être modifiées par l'utilisateur extérieur comme contaminées (« tainted » en anglais). Ainsi, si une variable contaminée est utilisée dans une expression qui initialise une seconde variable, cette dernière est aussi marquée comme contaminée. On obtient au final un ensemble complet de variables potentiellement influencées par une intervention extérieure. Ainsi, la propriété de contamination concerne les données. L'approche proposée se focalise donc en particulier sur les arguments des appels système. Par ailleurs, l'analyse est sensible au contexte, comme on a pu le croiser dans certaines techniques en boîte grise. Comme dans toute approche comportementale, la démarche est divisée en deux parties. Dans une première partie, le programme étudié va être instrumenté de manière à marquer comme contaminé certains des arguments des appels système. Les propriétés des arguments contaminés vont ensuite être modélisées durant une phase d'apprentissage dans laquelle on va inférer les distributions de leurs tailles et de leurs structures. Pendant la phase de détection, un modèle de comportement de l'application va être créé de manière dynamique. Si ce modèle de comportement est différent de celui appris durant la phase précédente, une alarme sera déclenchée. L'implémentation de cette approche a montré qu'il était avantageux de considérer l'ensemble des variables contaminées plutôt que la globalité des variables du programme car cela permet de réduire le nombre de faux positifs.

Castro et al. [11] proposent une démarche différente qui permet comme précédemment de détecter des attaques contre les données. Cette technique consiste à assurer l'intégrité du flot de données de l'application. Dans cet objectif, les auteurs proposent une approche en trois temps. Dans la première phase, une analyse statique du programme est réalisée pour calculer un graphe du flot de données de l'application. Dans la deuxième phase, le

programme est instrumenté de manière à garantir que le flot de données à l'exécution est autorisé par le graphe précédemment calculé. Enfin, la dernière phase concerne la détection : le programme instrumenté est exécuté et une alerte est émise si l'intégrité du flot de données n'est plus assurée. Cette technique a l'avantage de ne pas avoir de faux positifs. En effet, lors de la construction du modèle, l'analyse statique calcule une sur-approximation du comportement de l'application. Ainsi, on peut affirmer avec certitude que si une alarme est émise, il s'agit effectivement d'une attaque. Nous pouvons cependant observer des faux négatifs. En effet, une attaque peut être ignorée dû à la sur-approximation du comportement de l'application.

Nous avons vu à travers ces quelques techniques comment le code source du programme pouvait être utilisé pour avoir des modèles précis permettant de détecter une vaste panoplie d'attaques. Dans la suite, nous nous focalisons en particulier sur une autre technique de l'approche boîte blanche, que nous utiliserons également pour le stage de cette année.

2.3 Modèle de détection fondé sur des invariants

Nous proposons dans ce chapitre de décrire deux approches boîtes blanches de la détection d'intrusion comportementale. Ces dernières ont pour particularité de construire un modèle de référence fondé sur les invariants d'un programme. Par ailleurs, elles se concentrent uniquement sur la détection d'attaques contre les données d'un programme. Comme nous l'avons vu précédemment, ces attaques [9] exploitent du code valide mais de façon illégale de manière à corrompre les données saines d'un programme. Le stage de cette année se positionnera également suivant cette approche.

2.3.1 Construction du modèle par analyse statique

Les travaux de Chen et al. [9] ont montré que des attaques contre les données avaient pour conséquence de mettre l'état interne du programme visé dans un état incohérent par rapport à ses spécifications. Dans l'article [12], Demay et al. proposent d'exploiter cet état interne du programme et de vérifier la cohérence des données pour détecter des appels système intrusifs.

La manière la plus naturelle de construire un tel modèle est de connaître les spécifications du programme et de vérifier si l'exécution est en accord avec ces dernières. Toutefois, en leurs absences, les spécifications peuvent être dérivées du programme en analysant son code source par analyse statique. Dans cette approche, l'état interne du programme est représenté par un ensemble de variables qui ont une influence sur les appels système. C'est à dire que nous nous intéressons aux variables utilisées pour calculer les arguments d'un appel système et les variables qui ont contribué à l'exécution de cet appel. Les propriétés sur ces variables sont des invariants et sont à

la base de ce modèle. Par exemple, une variable x peut avoir pour invariant d'appartenir au domaine [0,9] dans toutes les exécutions du programme. Rappelons que l'objectif d'une attaque contre les données est de modifier la valeur de certaines variables. Ainsi, si tous les invariants d'un programme sont préservés durant l'exécution, aucune intrusion n'est détectée. Cependant, si un seul des invariants est violé, on peut dire avec certitude qu'il s'agit d'une intrusion. En effet, comme nous l'avons vu de façon similaire dans l'article [11], les invariants forment une sur-approximation du comportement de l'application.

La construction du modèle de comportement est réalisée en trois temps. La première étape consiste à découvrir l'ensemble des variables que le système doit surveiller. Cette détermination est réalisée par analyse statique sur le code source du programme. La deuxième étape a pour objectif de déduire du code source des invariants sur cet ensemble de variables. Dans cette implémentation, les invariants générés concernent uniquement le domaine de variation des variables. Enfin, pour détecter des violations d'invariants durant l'exécution, le programme est instrumenté. Pour cela, des assertions sont insérées à différents points du programme (notamment avant des appels système) et vérifient que les invariants sont toujours valides.

D'après les tests effectués, les ressources additionnelles induites par la mise en place d'un tel système sont négligeables. Par ailleurs, la précision d'un tel système dépend totalement des assertions générées. Les invariants établis concernent uniquement le domaine de variation des variables. Des variables ayant un domaine de définition défini à l'exécution ne pourront pas être surveillées. Ainsi, cette approche est encline aux faux négatifs.

2.3.2 Construction du modèle par analyse dynamique

Les travaux initiés par Sarrouy et al. dans [13] vont plus loin dans le modèle de détection fondé sur les invariants du programme que précédemment [12]. En effet, la détermination des invariants est réalisée ici de manière dynamique.

Le principe de détection est similaire à la technique précédente. En effet, le modèle de référence repose sur la détermination des invariants caractérisant les variables sensibles du programme. Comme nous avons pu le voir également précédemment, le succès d'une telle approche repose fortement sur la capacité à obtenir cet ensemble d'invariants sur les données de l'application.

Toutes les données de l'application ne sont pas nécessaires. Les données sensibles sont l'ensemble des variables influençant les appels système et leurs arguments. En effet, si une variable n'influence pas les appels système ou leurs arguments, elle n'a que peu d'intérêt pour l'attaquant et donc pour la détection d'intrusion. On peut également considérer comme données sensibles les entrées fournies par l'utilisateur. La première étape du travail a

donc été de trouver une façon d'extraire les données internes que l'on veut observer. La solution retenue ici consiste à émuler un processeur contrôlé par le système de détection d'intrusion et à exécuter l'application dans ce contexte. Pour cela, l'environnement d'instrumentation dynamique de binaire Valgrind[2] a été utilisé. En particulier, un plugin pour Valgrind a été développé, nommé Fatgrind. Ce dernier est capable de déterminer dynamiquement les données sensibles aux intrusions et de générer les traces d'exécution associées.

Après avoir obtenu un certain nombre de traces d'exécutions normales, il s'agit d'en déduire automatiquement les invariants caractérisant les données sensibles. Dans cet objectif, l'outil Daikon [14] a été utilisé. Cet outil est une implémentation de la détection dynamique d'invariants [15]. Ainsi, à partir des traces d'exécution, Daikon déduit dynamiquement les invariants les plus probables. Contrairement à la technique précédente où l'on ne considérait que des invariants portant sur le domaine de variation des données, les invariants calculés par Daikon peuvent être relativement complexes. Il peut déterminer des invariants sur une donnée unique. Par exemple, le fait qu'une variable soit une constante, qu'elle soit non nulle, qu'elle appartienne à un ensemble fini de valeurs, etc. Il peut également déterminer des invariants sur plusieurs données. Par exemple, il peut déduire que plusieurs variables vérifient une relation linéaire ou une relation d'ordre. La qualité des invariants trouvés par Daikon dépend bien évidemment de l'exhaustivité de la phase d'apprentissage du comportement normal de l'application.

Les invariants extraits par cette approche ont été validés en les confrontant à des attaques connues contre les données. Ces résultats confirment en effet que la prise en considération de ces invariants à l'exécution empêcherait la réalisation de telles attaques. Ainsi, les résultats obtenus prouvent la viabilité d'une telle approche et sont encourageants pour la continuité de ces travaux. Nous verrons plus en détails cette approche dans la suite de ce rapport.

2.4 Spécificités des applications web

Durant le stage, nous nous sommes intéressés au cas particulier des applications web. Ce type d'application est devenu en quelques années très utilisé. Et à mesure que le nombre d'applications web a grandi, le nombre d'attaques à leur encontre a augmenté de pair. Les attaques peuvent se porter contre les serveurs web aussi bien que vers les applications clientes. Nous présentons dans la suite trois approches répondant à des problématiques différentes sur la sécurité des applications web.

Dans l'article [16], Vigna et al. s'intéressent à la détection d'intrusion sur

[2]Valgrind est un framework d'instrumentation dynamique de programme. Plus d'informations sur http ://valgrind.org

les serveurs web. Leur travail se place dans le cadre de la détection d'intrusion par scénarios et a débouché sur la réalisation d'un IDS nommé WebSTAT. Les attaques sont dans un premier temps modélisées dans un langage de haut niveau, grâce au framework STAT[3] puis automatiquement compilées pour être utilisées comme signature lors de la détection d'intrusion. STAT leur permet d'exprimer des attaques complexes en terme d'états et de transitions. WebSTAT se positionne ainsi comme un IDS à états. Il a en effet la capacité de détecter des attaques temporelles, en prenant en compte l'historique des évènements passés. Lors de la détection d'attaques, WebSTAT a la particularité de prendre en compte différentes sources d'évènements et de les corréler entre elles pour améliorer la détection. Ainsi, des évènements bas niveaux issus des paquets réseaux et des logs du système d'exploitation sont mis en relation avec les logs du serveur. Cette caractéristique permet de gagner en efficacité en réduisant le nombre de faux positifs.

Robertson et al. se sont intéressés aux problèmes récurrents de la détection d'intrusion comportementale dans l'article [17]. Ils présentent deux techniques permettant à un administrateur de gagner du temps en connaissant la nature des intrusions et leurs criticités. En effet, contrairement à la détection d'intrusion comportementale, l'approche par scénarios présente l'avantage d'avoir moins de faux positifs et de connaître le type de l'attaque détectée, grâce à la base de signature. Robertson et al. désirent se rapprocher de ces caractéristiques pour la détection d'intrusion comportementale. Ils proposent tout d'abord une étape de généralisation, qui à partir d'une requête web, crée une signature. Les caractéristiques telles que la taille des paramètres ou la distribution des caractères de la requête web sont utilisées pour créer un modèle afin de pouvoir regrouper ensuite les requêtes présentant des propriétés similaires. Par ailleurs, cette approche propose une étape d'inférence des classes d'attaques des intrusions. Les intrusions sont confrontées à des heuristiques pour savoir s'il s'agit d'une attaque visant un débordement de buffer, une exploitation XSS, une injection SQL, etc.

Dans l'article [18], Cova et al. proposent pour la première fois de surveiller la valeur des variables en session en des points critiques de l'application. Cette approche, nommée Swaddler, s'intéresse à l'état interne de l'application, qui est surveillé avec différents modèles. Durant la phase d'apprentissage, l'application est dans un premier temps instrumentée avec du code pour extraire les valeurs des variables en différents points critiques. L'instrumentation est ainsi réalisée au début de chaque bloc de base ; un bloc de base étant un bloc de code sans possibilités d'arrêt ou de branchements. Les valeurs récupérées sont ensuite analysées pour créer un profil en différents points de l'application. Plusieurs propriétés sont ainsi capturées : des propriétés sur des variables uniques mais aussi sur des relations entre

[3]Pour plus d'informations sur STAT, consulter
http ://www.cs.ucsb.edu/~seclab/projects/stat/software/stat_framework.html.

différentes variables. Comme notre approche, Swaddler introduit la notion d'invariant et utilise en partie l'outil Daikon pour établir des relations complexes entre variables. Les profils créés sont ensuite utilisés pour détecter des attaques à l'exécution. L'implémentation de Swaddler a été réalisé en PHP et d'après les tests effectués, deux facteurs sont déterminants sur les performances de l'application surveillée : le nombre de variables analysées dans chaque bloc de base et le nombre de blocs de base.

Durant le stage, notre approche a consisté à définir le comportement légitime d'un programme par des contraintes - ou invariants - sur les variables critiques. Nous nous sommes servi ensuite de la modélisation de ce comportement normal pour détecter des infractions à l'exécution du programme. Contrairement à Swaddler, nous ne nous concentrons pas uniquement sur les variables en session. Nous identifions les variables qui peuvent potentiellement être utilisées dans une attaque et nous considérons tous les types de variables : attributs d'un objet, variables locales, sessions, cookies... Par ailleurs, le traçage des variables que nous effectuons est lui aussi beaucoup plus fin comme nous le verrons dans la suite.

Chapitre 3

Contexte de travail

Dans ce chapitre, nous allons décrire notre contexte de travail. Nous expliciterons tout d'abord les objectifs du stage. Nous poursuivrons sur la présentation des outils utilisés, car notre démarche tire un avantage important de ces technologies. Nous illustrerons enfin les attaques les plus fréquentes contre les applications web.

3.1 Objectifs

Les fondements de notre approche reposent sur les travaux initiés par Sarrouy et al. dans [13] que nous avons présenté précédemment dans la partie 2.3.2. Cette approche repose sur la découverte de contraintes - ou invariants - sur les variables du programme et sur la pertinence de ces invariants. Ces invariants caractérisent les relations que peuvent avoir les variables du programme entre elles. Toutes les variables du programme ne sont pas nécessaires. Comme nous le verrons de manière détaillée dans la suite, nous considérons uniquement les variables critiques de l'application. Par ailleurs, notre approche se concentre sur les attaques contre les données. La première étape consiste à tracer les variables critiques efficacement et à générer des invariants à partir de ces traces avec Daikon. Dans une deuxième étape, les invariants générés sont utilisés lors de l'exécution de l'application. Si un invariant est violé, on supposera que nous sommes en présence d'une intrusion.

Les travaux de Sarrouy et al. dans [13] ont abouti sur une implémentation de cette approche au niveau binaire jusqu'à la génération d'invariants. L'étude des invariants générés a pu montrer qu'ils auraient pu permettre la détection des attaques étudiées. Durant le stage de cette année, nous avons voulu étudier cette approche à un niveau plus haut, celui d'un langage interprété et vérifier que les invariants générés permettent de détecter effectivement certaines attaques en pratique. Nous nous sommes ainsi placés dans le contexte particulier du langage Ruby, et des applications web, et plus particulièrement

des applications écrites avec le framework Ruby on Rails. Le nombre d'applications web est de plus en plus important aujourd'hui, tout comme les attaques à leurs encontres. Par ailleurs, un langage interprété permet d'avoir d'avantages d'informations que le niveau binaire sur les variables, par exemple en considérant leurs types. Ainsi, nous pouvons espérer obtenir des invariants plus détaillés qu'au niveau binaire. L'objectif du stage a donc été d'adapter l'approche de Sarrouy et al. dans [13] au contexte des applications web sur un langage interprété et de vérifier sa pertinence. Nous désirons également que l'implémentation de cette approche puisse être distincte de l'application surveillée, c'est à dire, qu'il ne soit pas nécessaire de modifier son code source.

3.2 Technologies et outils utilisés

Nous présentons dans la suite les principaux outils et technologies que nous avons été amené à utiliser durant le stage. Leur choix a eu un impact important sur l'approche que nous avons choisi d'étudier.

3.2.1 Ruby

Ruby[1] est un langage interprété, libre et fortement orienté objet. En effet, en Ruby, tout est objet. Toutes les fonctions sont des méthodes et toute variable est une référence à un objet. Par ailleurs, il existe deux moyens pour structurer son code : les classes et les modules. Les modules peuvent être inclus dans des classes pour leur ajouter des fonctionnalités supplémentaires. Ruby met l'accent sur une syntaxe élégante, simple et axée sur une meilleure productivité.

En particulier, le langage Ruby offre de grandes possibilités du côté de la méta programmation. Ainsi, il est possible par exemple de modifier des classes dynamiquement et d'ajouter ou modifier des méthodes durant l'exécution du programme. Une classe peut être réouverte à tout moment pour y définir de nouvelles méthodes. Il est possible par exemple de rajouter des fonctionnalités aux classes de base comme la classe String. Ruby introduit également de nombreuses fonctionnalités réflexives, c'est-à-dire la possibilité d'avoir un regard introspectif sur le code exécuté. Il est ainsi possible de récupérer les différentes variables à un point du programme, connaître les méthodes d'une classe, etc.

[1] Pour plus d'informations sur le langage Ruby, se référer à http ://www.ruby-lang.org

3.2.2 Ruby on Rails

Ruby on Rails[2] est un framework open source écrit en Ruby. Il permet de créer des applications web rapidement en imposant un cadre de travail et une structure de développement aux programmeurs. Ruby on Rails fournit des fonctionnalités de haut niveau, permettant d'abstraire les tâches rébarbatives au programmeur. Le framework est basé sur deux principes fondamentaux : ne pas se répéter dans le code grâce aux capacités de Ruby et privilégier la convention sur la configuration. En effet, Ruby on Rails propose des comportements par défaut pour une grande majorité de ses fonctionnalités. Par ailleurs, Ruby on Rails est fortement extensible et il est aisé d'y ajouter des plugins et d'étendre ses fonctionnalités.

Ruby on Rails utilise le motif de conception MVC (Modèle Vue Contrôleur). Le modèle concerne les données. Ainsi, un enregistrement présent dans une base de données correspond à une instance d'une classe du modèle. Le rendu de ces données sur le navigateur Internet est réalisé par les vues. Enfin, les contrôleurs font le lien entre les modèles et les vues. Prenons un exemple particulier avec la figure 3.1. Le navigateur envoie une requête à l'application web (1). Par exemple, nous voulons afficher une liste de produits dans le cas d'une boutique de e-commerce. La requête est reçue par un contrôleur qui va demander au modèle (2) de lui retourner la liste des produits. Le modèle dialogue avec la base de données (3 et 4) et renvoie la liste des produits au contrôleur (5) qui va les passer à une vue (6). Finalement, la vue va être envoyée au navigateur pour être affichée (7) à l'écran de l'utilisateur.

Nous avons vu que les contrôleurs ont une place centrale dans une application Ruby on Rails. Plus précisément, un contrôleur est composé de méthodes appelées actions. Les URL de l'application Ruby on Rails sont composées par défaut d'un nom de contrôleur et d'un nom d'action. Par exemple, si un utilisateur désire accéder à une page qui a pour URL *http ://www.domaine.fr/produits/liste*, l'action *liste* du contrôleur *produits* sera appelée.

3.2.3 Daikon

Daikon[3] [14] est une implémentation de la détection dynamique d'invariants [15]. A partir des traces d'exécution (en particulier les noms des variables et leurs valeurs), Daikon est capable d'inférer les invariants les plus probables entre les différentes variables tracées. Par exemple, il peut déterminer à la fois des invariants sur une donnée unique (constante, domaines de valeur...) ou sur plusieurs données (relation d'ordre, relation

[2]Pour plus d'informations sur le framework Ruby on Rails, se référer à http ://rubyonrails.org/

[3]Pour plus d'informations sur Daikon, se référer à
http ://groups.csail.mit.edu/pag/daikon/

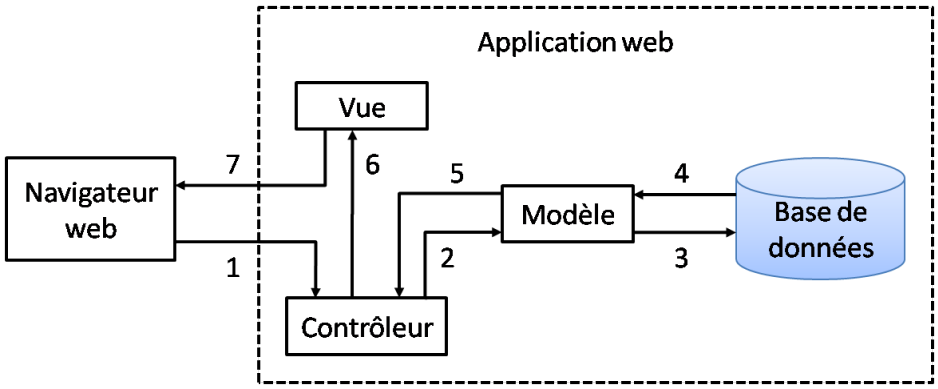

FIG. 3.1 – Cycle d'une requête dans Ruby on Rails : les contrôleurs sont toujours au centre des échanges.

linéaire...). La découverte dynamique d'invariants se déroule en deux étapes. Dans un premier temps, l'instrumenteur (ou traceur) récupère des informations sur les variables du programme et génère un fichier de trace. L'instrumenteur est dépendant du langage source du programme puisqu'il doit s'interfacer avec ce dernier pour en extraire les variables. Dans un deuxième temps, Daikon utilise ces fichiers de trace pour découvrir des invariants. Le moteur d'inférence qui calcule les invariants est indépendant de tout instrumenteur et est écrit en Java.

3.3 Exemples d'attaques contre les applications web

Différents types d'attaque peuvent menacer les applications web. A titre de rappel, notre approche se concentre sur les attaques contre les données. Nous présentons ci-dessous trois types d'attaques différents et commentons à chaque fois si nous pensons que notre approche est capable ou non de détecter ces attaques.

3.3.1 Les injections SQL

Considérons une application web qui nécessite une authentification par mot de passe pour accéder à l'administration. Nous pouvons envisager une requête SQL telle que ci-dessous :

```
1  SELECT login, password
2  FROM users
3  WHERE password = ' + pass + ';
```

Si la valeur de la variable *pass* est affectée à *' OR '1'='1*, la requête devient :

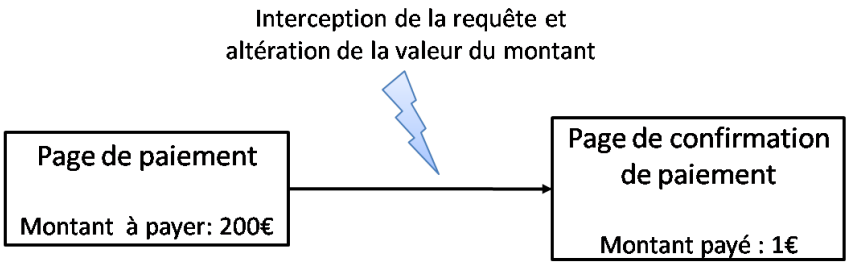

FIG. 3.2 – Altération d'une requête lors d'un paiement. L'altération est effective s'il n'y a pas de vérification côté serveur.

```
1 SELECT login, password
2 FROM users
3 WHERE password = '' OR '1' = '1';
```

Ainsi, en l'absence de mesure de sécurité, la deuxième partie de la condition est remplie et n'importe quel utilisateur malveillant peut s'authentifier de cette manière sans connaître le véritable mot de passe. Nous pensons que notre approche est capable de détecter ce genre d'attaque. Dans ce cas particulier, nous devons trouver un invariant imposant que le mot de passe retourné doit être égal au mot de passe envoyé en paramètre de la requête SQL.

3.3.2 L'altération des requêtes

Imaginons une application de vente en ligne où des utilisateurs peuvent acheter des produits. Le processus de paiement se déroule généralement en deux étapes ou plus. L'utilisateur doit remplir un formulaire avec les informations de paiement puis valider. Les données de paiement sont alors envoyées au serveur qui effectue le paiement auprès de la banque et renvoie une confirmation sur une nouvelle page au client. Imaginons un cas d'étude où lors de l'envoi des données du formulaire, la requête venait à être modifiée, et qu'il n'y ait pas de vérification des données du côté serveur. Un attaquant pourrait par exemple changer le montant des produits à payer comme illustré sur la figure 3.2. La modification des paramètres d'une requête est facilement réalisable avec le plugin Tamper Data[4] pour le navigateur Firefox. Nous pensons là aussi que notre approche est capable de détecter ce genre d'attaque si on parvient à générer un invariant (en restant sur l'exemple du montant lors de l'achat de produit) qui définit que le montant doit rester égal entre la page de paiement et la page de confirmation.

[4]Pour plus d'informations, se référer à
https ://addons.mozilla.org/en-US/firefox/addon/966/

3.3.3 Les attaques XSS

Lors d'une attaque XSS (Cross-Site Scripting), un attaquant force le navigateur d'un utilisateur à exécuter du code fourni par l'attaquant. Il s'agit typiquement de code Javascript. Un attaquant insère un bout de code malicieux dans une partie du contenu d'une application web, par exemple dans un commentaire sur un blog. Lorsqu'un utilisateur normal va accéder à la page, et que le blog en question n'a pris aucune mesure pour empêcher ce genre d'attaque, le code malveillant va être exécuté du côté client et va pouvoir accéder potentiellement à des informations sensibles conservées par le blog sur l'utilisateur, comme les valeurs des cookies par exemple. Notre approche ne pourra pas détecter ce genre d'attaque car il ne nous est pas possible de générer des invariants caractérisant la normalité d'un texte. En effet, notre approche est particulièrement efficace si nous pouvons trouver des relations entre les variables. Cependant, dans le cadre du projet DALI, dans lequel s'insère nos travaux, des contrats issus de la spécification du code HTML généré permettent de détecter ce genre d'attaques.

Chapitre 4

Détection d'attaques contre les données dans les applications web

Nous décrivons en détail dans ce chapitre la démarche de notre approche. Tout d'abord, nous présenterons notre modèle formellement. Nous verrons ensuite quelles sont les données utiles à la construction du modèle et quelles données nous traçons au sein d'une application Ruby on Rails. Nous nous intéresserons à la génération des invariants puis à l'utilisation de ces invariants pour la détection d'attaques.

4.1 Présentation du modèle[1]

L'exécution d'un programme peut se caractériser en terme d'états. Nous définissions l'état s comme l'ensemble des valeurs des variables présentes à un point donné de l'exécution du programme. L'ensemble des états globaux d'un programme, noté $\mathcal{S}$, est un ensemble dont les éléments dépendent des domaines de définition des variables. Par exemple, avec le langage Ruby, une variable entière n'a en théorie pas de limites (notamment grâce à la classe Bignum qui englobe les entiers très larges). Cependant, lors de l'exécution d'un programme, tous les états globaux ne sont vraisemblablement pas atteints. Nous pouvons alors définir un sous ensemble de $\mathcal{S}$ constitué des états globaux légitimes et atteignables que l'on notera $\mathcal{A}$. Par exemple, une variable entière peut prendre ses valeurs uniquement dans l'intervalle `[0, 5]` durant l'exécution du programme ou dépendre d'une relation particulière avec d'autres variables lui imposant un ensemble de valeurs particulières. Par ailleurs, si une attaque est possible contre le programme, cela signifie

[1] La description du modèle d'origine peut être consultée dans les travaux initiés par Sarrouy et al. dans [13].

qu'il est possible d'amener ce programme dans un état certes problématique, mais atteignable. L'ensemble $\mathcal{A}$ est donc défini comme l'ensemble des états globaux pour une utilisation normale du programme, c'est à dire, sans attaques.

Nous proposons d'abstraire la définition des états globaux de $\mathcal{A}$ par l'expression de contraintes - ou invariants - sur les valeurs autorisées et sur les relations que peuvent avoir des variables du programme entre elles. Nous pensons que l'ensemble des états globaux de $\mathcal{A}$ peut être explicitement défini par un ensemble d'invariants, vérifiés lorsque le programme est dans un état $s \in \mathcal{A}$. Nous pouvons alors considérer que l'application est dans un état anormal, c'est à dire dans un état $s \in \mathcal{S} \backslash \mathcal{A}$, lorsqu'un des invariants n'est plus vérifié. Une attaque peut donc être décrite comme l'observation d'une action conduisant le programme d'un état $s_i \in \mathcal{A}$ à un état $s_f \in \mathcal{S} \backslash \mathcal{A}$ en violant un ou plusieurs invariants.

Ainsi, le modèle que nous considérons ici repose sur la faculté à construire l'ensemble des invariants caractérisant les états $s \in \mathcal{A}$. Lorsque cet ensemble d'invariants est obtenu, il suffit d'observer que l'exécution du programme respecte ces invariants et de lever une alerte lorsque l'un d'entre eux est violé.

4.2 Définition des données utiles à la construction du modèle

La construction de l'ensemble des invariants nécessite l'extraction des valeurs des variables à l'exécution du programme. Cependant, toutes les variables ne sont pas forcément pertinentes. Nous allons voir dans ce qui suit quelles sont les variables critiques dans un programme puis nous nous focaliserons plus particulièrement sur les applications web écrites avec le framework Ruby on Rails.

L'ensemble des variables critiques d'un programme, que nous nommerons *ISDS* pour *Intrusive Sensitive Data Set*, appartient à deux ensembles. Tout d'abord, comme l'ont remarqué Kruegel et al. dans [5] auparavant, les attaques se manifestent souvent dans les arguments des appels système. Nous considérons donc que l'ensemble des données sensibles est un sous-ensemble des données pouvant influencer directement ou indirectement un appel système ou ses arguments. En effet, si une variable n'influence pas les arguments des appels système (par exemple pour une connexion à une base de données, une ouverture d'un fichier...), elle n'a que peu d'intérêt pour un attaquant. Le deuxième ensemble est celui des données influencées par l'extérieur, c'est-à-dire, par les entrées fournies par l'utilisateur. En effet, il s'agit là de la porte d'entrée d'un grand nombre d'attaques. Par exemple, il peut s'agir des données issues d'un formulaire, d'un cookie ou encore des paramètres d'une URL. Ces données sont dites *marquées* - « *tainted* » en

anglais - car elles sont le moyen par lequel les attaques se propagent dans le programme.

L'influence que peut avoir une variable sur les autres peut être illustrée à l'aide du concept de dépendance causale. Notons *(v,t)* la valeur de la variable v à l'instant t. Nous pouvons représenter la dépendance causale de *(v,t)* par rapport à *(v',t')* par *(v',t')*$\rightarrow$*(v,t)*, où $t' \leq t$ et $\rightarrow$ est une relation transitive. Ainsi, nous pouvons définir le cône de causalité de *(v,t)* comme l'ensemble des points *(v',t')* qui influence directement ou indirectement *(v,t)* :

$$cause(v,t) = \{(v',t')/(v',t') \rightarrow (v,t)\}$$

Par exemple, sur la figure 4.1, le cône de causalité de *(v_3,t_3)* est l'ensemble {*(v_1,t_1), (v_2,t_2)*}. De manière identique, le cône de dépendance de *(v,t)* est l'ensemble des variables *(v',t')* qui sont influencées par *(v,t)* :

$$dep(v,t) = \{(v',t')/(v,t) \rightarrow (v',t')\}$$

Sur la figure 4.1, le cône de dépendance de *(v_1,t_1)* est l'ensemble {*(v_2,t_2), (v_3,t_3), ...*}.

Nous pouvons désormais exprimer formellement l'ensemble des variables critiques *ISDS*. Comme nous l'avons dit auparavant, les variables critiques appartiennent au sous ensemble des données pouvant influencer les appels système. En notant AS l'ensemble des appels système, nous obtenons :

$$ISDS \subseteq cause(AS)$$

L'ensemble *ISDS* appartient également à l'ensemble des données influencées par l'extérieur, noté *UI*. Nous pouvons écrire :

$$ISDS \subseteq dep(UI)$$

Au final, l'ensemble des données critiques du programme est limité à l'intersection des données influencées par l'extérieur et des données ayant une incidence sur les appels système, donc à l'intersection entre le cône de dépendance des entrées utilisateurs et le cône de causalité des appels système (voir l'ensemble *ISDS* sur la figure 4.1) :

$$ISDS = cause(AS) \cap dep(UI)$$

Dans une application Ruby on Rails, nous avons fait le choix de considérer ces variables critiques dans les contrôleurs de l'application uniquement. Comme nous l'avons vu précédemment dans la partie 3.2.2, les contrôleurs sont le lien entre les modèles et les vues. C'est un passage obligatoire pour toutes les requêtes de l'application web. Si nous sommes en présence d'une attaque contre des données, elle se reflètera dans une des actions d'un contrôleur. Nous pensons donc qu'il est suffisant de s'intéresser particulièrement aux

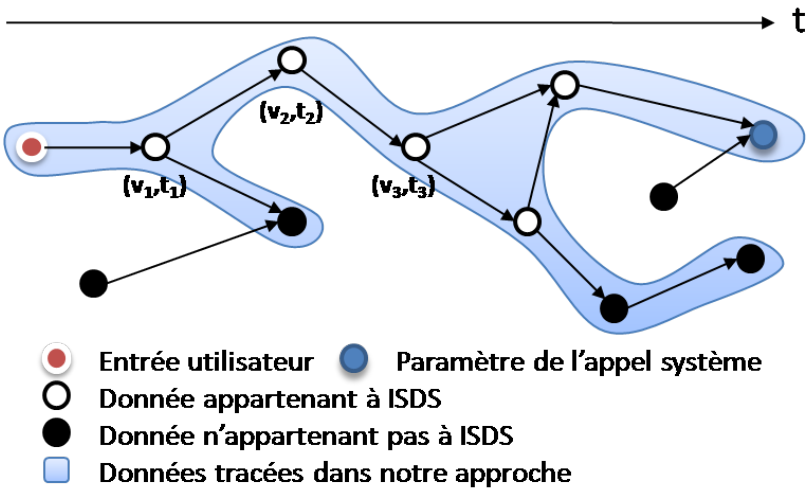

FIG. 4.1 – Ensemble des variables sensibles d'un programme.

variables critiques des contrôleurs pour créer le modèle. Par ailleurs, pour le moment, l'ensemble des variables critiques *ISDS* du programme est limité aux seules variables influencées par l'extérieur, c'est-à-dire au cône de dépendance des entrées utilisateurs *dep(UI)*. Nous ne tenons pas compte du cône de causalité des appels système *cause(AS)* tel que décrit précédemment. C'est une concession raisonnable dans le cas particulier des applications Ruby on Rails qui aboutit à une légère sur-approximation de l'ensemble *ISDS*, comme nous pouvons le voir sur la figure 4.1. En effet, les actions des contrôleurs sont de petite taille en général (une vingtaine de lignes de code habituellement) et les variables influencées par l'extérieur aboutissent généralement sur des appels système, comme paramètres d'une requête à une base de données par exemple.

4.3 Log des variables sensibles aux intrusions

Nous avons considéré deux approches complémentaires pour construire un modèle précis de l'application. Ainsi, nous allons tracer les variables dans deux situations différentes : au sein d'une action et entre deux actions consécutives.

4.3.1 Log à l'exécution d'une action

Durant l'étape d'apprentissage, nous traçons les variables présentes dans le corps des actions des contrôleurs de l'application. Toutes les variables critiques - « tainted » - de chaque action de chaque contrôleur vont être extraites. Ces données peuvent être définies formellement par le cône de

dépendance des entrées de l'utilisateur, *dep(UI)*, tel que décrit précédemment. En particulier, la notion de temps dans les dépendances causales entre les variables revient à considérer les lignes exécutées dans le cas où chaque ligne est exécutée une seule fois. Ainsi, *(v,l)* représente désormais la valeur de la variable v à la ligne l. Le cône de dépendance de *(v,l)* est alors :

$$dep(v, l) = \{(v', l')/(v, l) \rightarrow (v', l')\}$$

où l, l' sont des numéros de lignes de l'action exécutée et $l \leq l'$.

A l'aide de ces traces, l'objectif sera de trouver des relations que peuvent avoir des variables critiques sur des lignes différentes au sein d'une même action. Prenons l'exemple du listing 4.1 en supposant que la variable *reponse* appartienne à l'ensemble des données critiques. A la ligne 3, la variable *reponse* est logguée, ainsi qu'à la ligne 5. Ainsi, il va être possible d'établir des relations entre *(reponse,3)* et les autres variables, et *(reponse,5)* et les autres variables. Ainsi, la granularité à laquelle nous traçons les variables critiques nous permettra de trouver des relations entre les différentes occurences d'une même variable et les autres variables de l'action.

```
1 def methode(param1, param2)
2   ...
3   reponse = requete1(param1, param2)
4   ...
5   reponse = requete2(param1, param2)
6   ...
7 end
```

Listing 4.1 – Action d'un contrôleur

4.3.2 Log entre deux actions consécutives

La deuxième situation où nous traçons les variables concerne le passage d'une action à une autre. C'est typiquement le cas lorsqu'un internaute change de page. Les occurrences des variables critiques en fin de la première action et au début de l'action suivante vont être extraites. Comme nous pouvons l'observer sur la figure 4.2, l'objectif est de trouver des relations entre le cône de dépendance des entrées utilisateurs de l'action 1 et celui de l'action 2.

4.4 Génération des invariants

Une fois l'extraction des variables critiques terminée selon les deux situations présentées précédemment, nous utilisons Daikon pour générer les relations entre les variables critiques, appelées invariants. Le moteur d'inférence

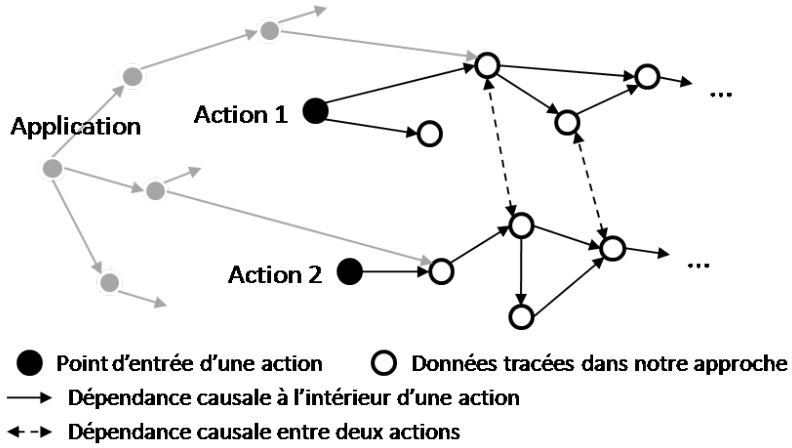

FIG. 4.2 – Dépendances causales entre des variables de deux actions différentes.

de Daikon accepte un format de trace unique en entrée et produit des invariants en sortie. L'idée principale derrière le fonctionnement du moteur d'inférence de Daikon est l'utilisation de l'algorithme *generate-and-check*. Dès le départ, Daikon suppose que tous les types d'invariants sont vrais entre toutes les variables tracées, puis confronte chacun d'eux aux traces des variables. Daikon reporte au final les invariants n'ayant pas été rejetés. Par ailleurs, plusieurs optimisations viennent s'ajouter à cet algorithme. Ces optimisations sont fondées sur le fait que l'algorithme *generate-and-check* fournit plus d'invariants que nécessaire et qu'un grand nombre d'entre eux est redondant. A titre d'exemple, si on obtient un invariant tel que $x>y$, alors cela implique aussi que $x\geq y$. Ce dernier invariant peut donc être supprimé. Pour plus de détails sur le moteur d'inférence, nous vous conseillons la lecture de l'article [14] de Ernst et al.

Dans ce qui suit, nous appelons « fichier de traces » un fichier contenant des variables critiques et leurs valeurs. Comme nous pouvons le voir sur la figure 4.3, à chaque exécution différente du programme correspond un fichier de trace. Dans un fichier de traces, Daikon impose de regrouper les variables que nous souhaitons mettre en relation, en blocs. Sur la figure 4.3, nous voulons générer des invariants propres à l'action 1, d'autres propres à l'action 2 et d'autres encore entre l'action 1 et l'action 2. Par exemple, à chaque exécution, les variables critiques de l'action 1 sont extraites et regroupées dans le bloc 1. Une fois que nous disposons d'un nombre suffisant d'exécution de l'application, Daikon utilise les blocs 1 de chaque fichier de traces pour générer les invariants propres à l'action 1.

Pour construire les invariants que nous voulons, nous avons dû détourner d'une certaine manière l'utilisation de Daikon. En effet, dans une grande majorité de langages, les instrumenteurs réalisent l'extraction des variables en

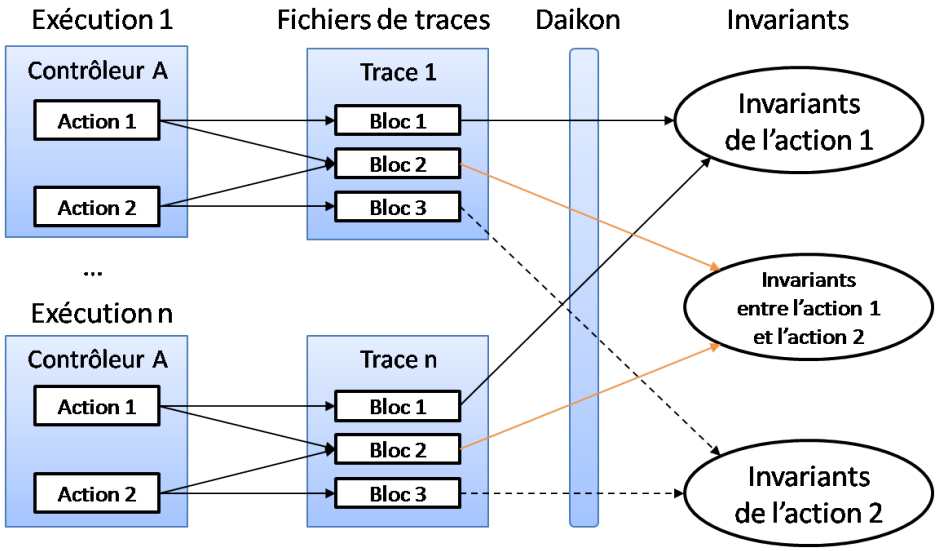

FIG. 4.3 – Processus de génération des invariants. Un fichier de traces est associé à chaque exécution. Dans un fichier de traces, un bloc correspond aux variables que nous souhaitons confronter les unes aux autres. Ici, les différentes occurrences des variables d'une même action sont regroupées dans un même bloc. Daikon utilise les blocs similaires pour générer les invariants.

début et en fin de méthodes. Daikon fournit des facilités dans son format d'entrée pour couvrir ces cas. Daikon impose également qu'un même identificateur de variables (son nom en l'occurrence) apparaisse une seule fois dans un bloc de fichier de traces. Notre situation est ainsi plus sensible. En effet, prenons le cas où nous souhaitons générer des invariants pour une action particulière. Il est probable qu'une même variable apparaisse sur plusieurs lignes différentes. Pour pallier la limitation de nommage de Daikon au sein d'un bloc, nous avons dissocié le nom de chaque variable en autant d'identificateur que le nombre de ligne où elle apparait. Cela nous permet par ailleurs de savoir quelle occurrence d'une variable est concernée par l'invariant dans lequel elle apparait.

Par ailleurs, puisque dans une utilisation classique de Daikon, seules les variables en entrées et en sorties d'une méthode sont tracées, l'outil ne gère pas le cas des boucles qui peuvent apparaître dans le code. Nous avons dû alors trouver un moyen pour gérer ce cas. Nous avons décidé de dissocier chaque tour de boucle en autant de fichiers de traces différents, comme s'il s'agissait d'exécution différente à chaque tour de boucle. Cela nous permet de passer outre les limitations de Daikon tout en arrivant à nos fins.

Les logs effectués dans la première situation, décrite dans la partie 4.3.1, permettent d'obtenir des invariants portant sur les variables critiques au sein d'une même action. Ces invariants répondent entre autre à la problématique

des injections SQL décrite en 3.3.1.En effet, il s'agissait de trouver idéalement une relation du paramètre *pass* avec des variables accessibles dans la même action de manière à détecter l'injection de faux mots de passe en environnement de production. Par ailleurs, les logs effectués dans la deuxième situation, décrite dans la partie 4.3.2, permettent d'obtenir des invariants portant sur les variables critiques entre deux actions. Ces invariants répondent quant à eux à la problématique de l'altération des données décrite en 3.3.2.

4.5 Détection d'intrusions

Lorsque la création du modèle de l'application est terminée, c'est-à-dire lorsque nous avons obtenu l'ensemble des invariants pour une utilisation normale de l'application, nous devons les utiliser pour détecter des attaques à l'exécution. Chacun des invariants doit être vérifié à un certain moment de l'exécution et si l'un d'entre eux est violé, nous devons émettre une alerte.

Dans une première étape, nous analysons le fichier d'invariants fourni par Daikon pour pouvoir l'utiliser dans la détection d'intrusion. Nous avons réalisé un compilateur pour transformer le fichier de sortie de Daikon dans une représentation exploitable en Ruby.

Nous pouvons envisager deux approches différentes pour la vérification des invariants à l'exécution du programme. La première consiste à instrumenter le code de l'application surveillée par analyse statique comme l'avaient réalisé Demay et al. dans [12]. Des assertions doivent être insérées à certains points du programme pour vérifier les invariants à l'exécution. L'inconvénient de cette approche est la difficulté de l'implémentation et la nécessité de modifier le code source du programme surveillé. La deuxième approche, celle que nous avons retenu, tire avantage des capacités du langage Ruby et des fonctionnalités offertes par le framework Ruby on Rails. Les fortes possibilités de méta programmation et les capacités d'introspection du code nous offrent la possibilité d'instrumenter dynamiquement le code exécuté et de vérifier à la volée les invariants sans avoir à modifier le code source du programme. Nous verrons plus en détails ces aspects dans le prochain chapitre.

Chapitre 5

Implémentation

Nous allons détailler dans ce chapitre l'implémentation de notre approche et les caractéristiques importantes des technologies que nous avons utilisées. La démarche est rappelée sur la figure 5.1. Nous verrons tout d'abord les spécificités du langage Ruby. Nous continuerons ensuite sur la façon dont nous avons intégré notre approche au sein du framework Ruby on Rails. Nous détaillerons comment nous avons réalisé l'apprentissage du modèle et la génération des invariants. Nous décrirons enfin la façon dont nous nous sommes servi du modèle créé pour détecter des attaques contre l'application.

Dans le cadre du projet DALI, Kereval a développé une application vulnérable écrite avec le framework Ruby on Rails. L'application est un prototype d'un site de e-commerce. Elle inclut les failles présentées dans la partie 3.3. Nous nous sommes appuyés sur cette application pour développer et tester notre approche.

5.1 Tracer et extraire les variables avec Ruby

Comme nous l'avons détaillé dans le paragraphe 4.2, la création du modèle débute par l'identification des variables critiques de l'application que nous serons amené à tracer. Le langage Ruby présente l'avantage de disposer nativement de fonctionnalités permettant d'identifier et de mani-

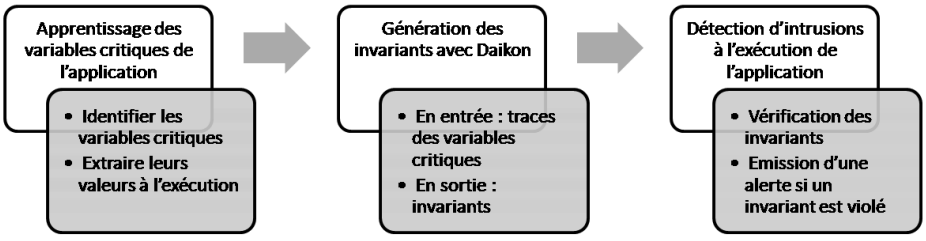

Fig. 5.1 – Synthèse de l'implémentation

```
1  x = gets
2  x.tainted?     # Retourne true
3  y = x
4  y.tainted?     # Retourne true
```

Listing 5.1 – Lecture de l'entrée standard et propagation de la propriété tainted.

```
1  mon_objet.instance_variables
```

Listing 5.2 – Retourne l'ensemble des noms des attributs d'un objet.

puler les variables influencées par l'extérieur. En effet, un flag « tainted » est associé à chaque objet du programme. Il est positionné à vrai si l'objet a pu être modifié ou influencé de l'extérieur. Chaque objet de Ruby possède une méthode booléenne « tainted ? » permettant de récupérer la valeur du flag « tainted ». Par ailleurs, la propriété « tainted » se propage. Ainsi, si une variable x influencée par l'extérieur est affectée à une variable y, la variable y héritera de la propriété « tainted », tel qu'illustré sur le listing 5.1. Comme nous l'avons justifié dans la partie 4.2, nous nous limitons à l'ensemble des variables influencées par l'extérieur pour la création du modèle, c'est-à-dire au cône de dépendance des entrées utilisateurs.

Le langage Ruby a également de grandes capacités d'introspection. En particulier, il existe des méthodes pour accéder aux différentes variables présentes dans une classe. Il est ainsi possible de récupérer l'ensemble des noms des variables statiques, des attributs d'une classe (voir listing 5.2) et des variables locales. Le cas des variables locales est un peu particulier. En effet, une variable locale est associée à un contexte et non pas à un objet comme c'est le cas pour les attributs d'une classe. Ainsi en Ruby, il est possible de récupérer le contexte d'exécution à un point donné du programme. Un contexte contient en particulier la valeur et le nom des variables accessibles à un moment précis (voir listing 5.3). Lorsque les noms des variables ont été récupérés, nous pouvons obtenir leur valeur et leur propriété « tainted » grâce à la méthode *eval*, qui évalue une chaîne de caractère dans un contexte donné.

Maintenant que nous savons comment récupérer dynamiquement les différents types de variables d'une classe, nous pouvons les utiliser pour construire le modèle. Comme nous l'avons énoncé dans la partie 3.1, nous désirons réaliser notre IDS sans avoir à modifier le code source de l'application surveillée. Nous avons dû trouver un moyen d'exécuter du code supplémentaire à chaque nouvelle ligne exécutée du programme sans modifier ses sources. Ruby fournit une méthode, *set_trace_func*, qui prend un bloc de code en paramètre, et une fois définie, est appelée à chaque nouvelle ligne de code exécutée. Le bloc de code passé en paramètre est alors exécuté à chaque

```
1   class Foo
2     def bar
3       i = 1
4       p = 2
5       return binding
6     end
7   end
8
9   t = Foo.new
10  contexte = t.bar
11  eval('local_variables', contexte)
12  # Retourne ['i', 'p']
13
14  eval('i', contexte)              # Retourne 1
15  eval('i.tainted?', contexte)     # Retourne False
```

Listing 5.3 – Le mot clé binding permet de récupérer le contexte à un point du programme. La méthode bar retourne le contexte d'exécution à la fin de la méthode. Nous évaluons ensuite la méthode local_variables sur le contexte récupéré pour connaître les noms des variables locales présentes. Connaissant le nom d'une variable nous obtenons sa valeur et sa propriété « tainted » dans le contexte.

nouvelle ligne. Cette méthode permet également d'accéder à de nombreuses informations très utiles : l'évènement de la ligne (appel de méthode, retour de méthode...), le nom du fichier, le numéro de la ligne exécutée, le nom de la classe et de la méthode dans laquelle nous nous trouvons et surtout, le contexte associé à la ligne courante. Connaissant le contexte à chaque nouvelle ligne de code exécuté, nous utilisons les techniques présentées ci-dessus pour récupérer les variables critiques à chaque nouvelle ligne.

5.2 Interfaçage avec Ruby on Rails

Comme nous l'avons expliqué auparavant dans la partie 4.2, nous pensons qu'il est suffisant de tracer les variables présentes dans les actions des contrôleurs d'une application Ruby on Rails. Pour cela, nous devons activer la fonction *set_trace_func* à chaque fois que nous entrons dans une action d'un contrôleur de l'application. Pour réaliser cela, introduisons la notion de filtre qui est une caractéristique de Ruby on Rails. Un filtre est une méthode qui va être appelée systématiquement avant ou/et après chaque action d'un contrôleur. L'idée est donc de définir des filtres dans la classe *ApplicationController*, dont hérite systématiquement tous les contrôleurs de l'application, de manière à ce que chaque action de chaque contrôleur bénéficie des filtres. Ainsi, le filtre antérieur à une action d'un contrôleur activera la fonction *set_trace_func*, tandis que le filtre postérieur la désactivera.

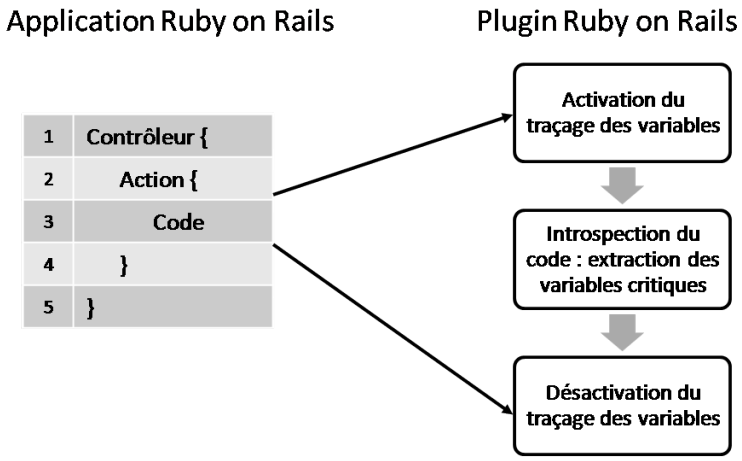

FIG. 5.2 – Interfaçage avec Ruby on Rails

Une des contraintes que nous nous étions fixé était d'appliquer notre approche aux côtés d'une application sans avoir à toucher à son code source. Ce choix est rendu possible grâce aux systèmes de plugins de Ruby on Rails. Ces derniers doivent suivre une convention particulière, notamment celle d'avoir un fichier nommé *init.rb* et placé à la racine du dossier contenant le plugin. Lors de l'initialisation de l'application, Ruby on Rails lit automatiquement les fichiers *init.rb* des éventuels plugins pour les intégrer à l'application. En particulier, les capacités de Ruby nous permettent d'étendre la classe *ApplicationController* dans le plugin pour définir les filtres. Ainsi, la seule chose à faire pour ajouter notre approche à une application existante est d'y installer un plugin, ce qui a le mérite d'être simple et de laisser intact l'application étudiée. L'interfaçage avec Ruby on Rails est illustré sur la figure 5.2.

5.3 Apprentissage

Un apprentissage minutieux et automatisé est essentiel pour obtenir un modèle de comportement normal le plus juste possible. En effet, plus nous obtenons de traces différentes des variables critiques, plus les invariants générés seront justes et indépendants d'une exécution donnée de l'application. Ainsi, il est important de parcourir plusieurs fois le plus d'actions possibles de chaque contrôleur pour obtenir un maximum de traces. Pour cette raison, l'automatisation de l'apprentissage est nécessaire. Nous avons opté pour l'utilisation de Selenium[1] pour automatiser l'apprentissage. Selenium regroupe une série d'outils destinés à tester les applications web. Nous nous en sommes servi pour parcourir au mieux l'application et simuler le plus fidèlement possible le comportement des utilisateurs normaux pour

[1]Pour plus d'informations sur Selenium, veuillez consulter http ://seleniumhq.org/

construire notre modèle. En particulier, grâce aux possibilités offertes par Selenium, nous pouvons piloter l'application en écrivant des scénarios en Ruby. Par exemple, dans le cas d'une page d'inscription sur notre application de test, nous avons écrit un scénario qui pioche automatiquement un pseudonyme et un mot de passe dans un dictionnaire pour créer un nouvel utilisateur. Selenium est également capable d'exécuter du Javascript. Nous nous servons de cette possibilité pour récupérer dynamiquement les liens de la page courante et ainsi créer le graphe des dépendances entre les pages au fur et à mesure de notre propagation dans l'application. Nous nous servons de ce graphe pour enregistrer le nombre de passages sur chaque lien et choisir à chaque fois le lien qui a été le moins visité pour couvrir convenablement l'ensemble de l'application.

5.4 Génération des invariants

Une fois que nous disposons d'un ensemble conséquent de variables tracées, nous pouvons générer des fichiers de traces pour Daikon. Comme nous l'avons décrit dans la partie 4.4, à chaque exécution correspond un fichier de traces et les variables que nous souhaitons confronter les unes aux autres sont regroupées dans des blocs. Daikon impose deux parties dans chaque bloc d'un fichier de traces : une partie où nous allons déclarer les variables et une autre où nous allons fournir les traces des variables. En ce qui concerne les traces associées à une action particulière, nous avons renommé chaque occurrence de variable en préfixant leur nom par le numéro de ligne et le type de la variable (variable de sessions, paramètres...) comme nous pouvons le voir sur le listing 5.4. De la même manière, pour les traces entre deux actions consécutives, nous avons ajouté un signe "-" devant les variables issues de la première action et un signe "+" devant les variables issues de l'action suivante. Nous introduisons ainsi une sémantique dans le nom des variables tracées, qui nous servira notamment dans l'analyse des invariants générés.

En sortie de Daikon, nous obtenons des invariants pour chaque action de chaque contrôleur. Par exemple, dans le listing 5.5, dans l'action *login* du contrôleur *users*, l'invariant nous informe que le champ *password* de l'objet *user* en paramètre, à la ligne 17 doit être égal au champ *password* de l'objet *user* en session à la ligne 21. Nous obtenons également des invariants associés au passage d'une action à une autre tel qu'illustré dans le listing 5.6. L'invariant est associé au passage de l'action *new* à l'action *create* du contrôleur *orders*. Il impose que la variable *amount* en paramètre dans l'action *new* (préfixé par un moins) doit être égale à la variable *amount* en paramètre dans l'action *create* (préfixé par un plus).

```
% Declarations (noms, types...)
ppt users.login:::POINT
ppt-type point
  variable 17_p_user.password
    var-kind variable
    dec-type String
    rep-type java.lang.String
  variable 21_s_user.password
    var-kind variable
    dec-type String
    rep-type java.lang.String

% Traces (noms et valeurs)
users.login:::POINT
17_p_user.password
"my_pass"
21_s_user.password
"my_pass"
```

Listing 5.4 – Bloc d'un fichier de traces. Il concerne le contrôleur users et l'action login.

```
  users.login:::POINT
  17_p_user.password == 21_s_user.password
```

Listing 5.5 – Invariant au sein d'une même action.

```
  orders-orders.new-create:::POINT
  -p_amount == +p_amount
```

Listing 5.6 – Invariant entre deux actions consécutives.

```
1 users.signup:::POINT
2 3_p_user.name == 11_s_user.name
3 3_p_user.login == 8_user.login
4 8_user.password == 11_s_user.password
```

Listing 5.7 – Exemple de format de sortie de Daikon.

```
1  {'users'=>
2    {'signup'=>
3      {'8'=>
4        [{:op=>'==',
5          :var1=>'3_p_user.login',
6          :var2=>'8_user.login'}],
7       '11'=>
8        [{:op=>'==',
9          :var1=>'3_p_user.name',
10         :var2=>'11_s_user.name'},
11        {:op=>'==',
12         :var1=>'8_user.password',
13         :var2=>'11_s_user.password'}]
14     }
15   }
16 }
```

Listing 5.8 – Exemple de format exploitable par Ruby après compilation du fichier de sortie de Daikon.

5.5 Détection d'intrusions

Après l'étape de génération des invariants, nous obtenons un fichier contenant une liste d'invariants pour chaque action de chaque contrôleur. Ce fichier a un format précis (voir listing 5.7), qu'il est nécessaire de retravailler pour que les invariants soient exploitables en Ruby pour la détection d'intrusion. Pour cela, nous avons réalisé une analyse syntaxique et sémantique du fichier avec le plugin Treetop[2], pour Ruby. Après cette analyse, les invariants ont été stockés dans une table de hâchage correctement indexée pour pouvoir naviguer facilement et rapidement dans cette base d'invariants. Le listing 5.8 présente cette table de hâchage dans le cas des invariants associés à une action particulière. Nous procédons de la même manière pour les invariants entre deux actions consécutives, seule la table de hâchage obtenue est légèrement différente.

Une fois que nous avons obtenu les invariants dans un format exploitable en Ruby, nous les utilisons pour détecter des intrusions à l'exécution de l'application en environnement de production. Nous reprenons le même principe

[2]Pour plus d'informations sur Treetop, veuillez consulter http ://treetop.rubyforge.org/

que pour le traçage des variables : nous utilisons la méthode *set_trace_func* pour exécuter du code à chaque nouvelle ligne du programme. Au sein d'une action, nous vérifions si des invariants sont présents dans la table de hâchage pour le contrôleur, l'action et la ligne courante. Si tel est le cas, nous extrayons comme auparavant les valeurs des variables critiques et testons les invariants un à un. Si l'un d'entre eux est violé, nous émettons une alerte. Le principe est similaire pour vérifier les invariants entre deux actions consécutives. Nous gardons en mémoire les variables critiques de l'action précédente et nous vérifions les invariants au début de l'action suivante. Nous avons choisi d'exporter les alertes dans un fichier au format XML.

Chapitre 6

Résultats

Nous avons pu tester notre approche sur l'application Ruby on Rails écrite par Kereval. A titre de rappel, cette application est en particulier vulnérable aux injections SQL et à l'altération des données issues d'un formulaire tel que nous l'avons décrit dans la partie 3.3. Les résultats obtenus sont encourageants car nous avons pu détecter toutes les attaques exploitant les vulnérabilités citées précédemment.

En particulier, une injection SQL était possible lors de l'authentification pour accéder à un compte administrateur, telle que décrite dans la partie 3.3.1. Nous avons pu obtenir l'invariant imposant que le mot de passe passé en paramètre de la requête soit égal au mot de passe retourné en session, permettant ainsi de détecter cette attaque (voir listing 5.5).

Une autre attaque permettait de modifier le montant d'un achat lors de l'envoi de la requête pour le paiement. Nous avons ici pu obtenir un invariant imposant que le montant présent dans l'action avant le paiement soit égal au montant présent dans l'action du paiement de la commande, permettant de détecter lorsque le montant est altéré d'une page à l'autre (voir listing 5.6).

Par ailleurs, nous avons observé un très grand nombre de faux positifs, c'est à dire des invariants violés mais qui ne reflètent pas d'attaques réelles. Cela est dû à un mauvais apprentissage de l'application. A l'heure de l'écriture de ce rapport, nous n'avons pas eu le temps de réaliser l'écriture de tous les scénarios pour parcourir l'application complètement et automatiquement. Par conséquent, certains invariants générés sont trop dépendants d'une exécution donnée. Nous constatons ainsi la nécessité d'un bon apprentissage du comportement normal de l'application, pour réduire le grand nombre de faux positifs actuels, comme nous l'avons justifié dans la partie 5.3.

En terme de performance, la fonction *set_trace_func* qui nous permet d'exécuter du code supplémentaire à chaque nouvelle ligne du programme est lente. Plus il y a de variables à analyser dans l'action courante, plus le

traitement sera long. Nous n'avons pas réalisé de comparatifs pour chiffrer les pertes de performance mais la différence est perceptible lors de la navigation. Nous avons des pistes pour améliorer les performances de notre approche, mais elles sortent du cadre de ce stage. Nous en parlons brièvement dans la prochaine partie.

Chapitre 7

Travaux futurs

Nous avons plusieurs pistes et idées pour améliorer le travail réalisé durant ce stage. Tout d'abord, en ce qui concerne les performances de notre approche et le choix de la fonction *set_trace_func*, il nous parait nécessaire de nous intéresser à l'interpréteur Ruby. La fonction *set_trace_func* est à l'origine une fonction de débugguage. Nous aimerions voir si nous pourrions étendre l'interpréteur Ruby pour y ajouter une fonction similaire répondant spécifiquement à nos besoins pour gagner en performance. A ce sujet, Ruby propose un système de plugin écrit en langage C, permettant d'étendre l'interpréteur aisément.

Comme nous l'avons énoncé précédemment, un bon apprentissage de l'application est essentiel pour obtenir des invariants précis et indépendants d'une exécution donnée du programme. Il est nécessaire de finir l'écriture des scénarios pour pouvoir se propager dans toute l'application et obtenir un grand nombre de traces sur des exécutions et des contextes différents (simulation du comportement des utilisateurs normaux, simulation d'un administrateur...).

Toujours dans l'objectif de réduire le nombre de faux positifs, nous pouvons également améliorer la génération d'invariant réalisée par Daikon, en limitant les différents invariants calculés. Par exemple, nous pouvons envisager de considérer les invariants comparants deux variables avec les opérateurs $>$ ou $<$ à des variables entières uniquement. A l'heure actuelle, Daikon utilise ce genre d'invariant même sur des chaînes de caractères, ce qui n'est pas très pertinent dans notre cas.

Enfin, nous envisageons de tracer également le flot normal de l'application. Cela permettrait d'éviter qu'un utilisateur accède directement à une page dont il ne devrait pas avoir accès. Par exemple, c'est le cas où un utilisateur cherche à accéder à une page de l'administration sans passer par la page d'authentification.

Chapitre 8

Conclusion

Durant ce stage, nous avons travaillé sur la détection d'intrusion comportementale. Nous avons continué les travaux initiés par Sarrouy et al. [13] dans l'équipe SSIR à Supélec. En particulier, nous nous sommes placés dans le contexte des applications web et des attaques contre les données. Notre approche a consisté à modéliser le comportement normal par des contraintes - ou invariants - sur les variables du programme. Pour cela, nous avons identifié tout d'abord les variables critiques de l'application. Nous avons tracé ces variables durant une phase d'apprentissage. Puis, nous avons utilisé Daikon pour générer les invariants à partir des traces d'exécution. Enfin, nous avons vérifié les invariants générés à l'exécution de l'application. Si un des invariants est violé, nous considérons que nous sommes en présence d'une attaque.

Nous avons implémenté notre approche avec le langage Ruby et le framework web Ruby on Rails. En partenariat avec l'entreprise Kereval, nous avons pu tester notre approche sur une application vulnérable et obtenir de premiers résultats encourageants. Nous avons en particulier pu détecter les attaques connues contre l'application test : injections SQL et altération des requêtes. De nombreuses améliorations sont envisageables, notamment en terme de performance et pour réduire le nombre de faux positifs que nous obtenons.

Ce stage m'a permis d'avoir une expérience dans le milieu de la recherche que j'ai beaucoup apprécié. La poursuite des travaux déjà initiés dans l'équipe m'a apporté un cadre de travail déjà avancé, qui m'a permis de continuer sur des bases existantes. Les technologies employées ont été très intéressantes et formatrices selon mon point de vue, et le fait d'avoir une application vulnérable pour tester notre approche a également été un gros avantage durant le stage.

Bibliographie

[1] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A Sense of Self for Unix Processes. In *Proceedings of the 1996 IEEE Symposium on Research in Security and Privacy*, pages 120–128. IEEE Computer Society, IEEE Computer Society Press, May 1996.

[2] Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3) :151–180, 1998.

[3] F. Gilham R. Jagannathan P.G. Neumann H.S. Javitz A. Valdes T.D. Garvey T.F. Lunt, A. Tamaru. A real-time intrusion detection expert system (ides). 1992.

[4] A. Valdes D. Anderson, T. Frivold. Next-generation intrusion detection expert system (nides) : A summary. 1995.

[5] Christopher Kruegel, Darren Mutz, Fredrik Valeur, and Giovanni Vigna. On the detection of anomalous system call arguments. In *8th European Symposium on Research in Computer Security (ESORICS 2003)*, pages 326–343, Gjøvik, Norway, October 2003.

[6] Darren Mutz, William Robertson, Giovannni Vigna, and Richard Kemmerer. Exploiting execution context for the detection of anomalous system calls. In *Proceeding of the 10th International Symposium on Recent Advances in Intrusion Detection (RAID'2007)*. Springer, 2007.

[7] Debin Gao, Michael K. Reiter, and Dawn Song. Gray-box extraction of execution graphs for anomaly detection. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 318–329, 2004.

[8] David Wagner and Drew Dean. Intrusion detection via static analysis. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, May 2001.

[9] S. Chen, J. Xu, E. Sezer, P. Gauriar, and R. Iyer. Non-control-data attacks are realistic threats. In *Usenix Security Symposium*, pages 177—-192, 2005.

[10] Lorenzo Cavallaro and R. Sekar. Anomalous taint detection. Technical report, Secure Systems Laboratory, Stony Brook University, 2008.

[11] Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, volume 7, page 11, 2006.

[12] Jonathan Christopher Demay, Eric Totel, and Frederic Tronel. Sidan : a tool dedicated to software instrumentation for detecting attacks on non-control-data. In *4th International Conference on Risks and Security of Internet and Systems (CRISIS'2009)*, Toulouse, October 2009.

[13] Olivier Sarrouy, Eric Totel, and Bernard Jouga. 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.

[14] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69 :35–45, 2007.

[15] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2) :99–123, February 2001.

[16] Giovanni Vigna, William Robertson, Vishal Kher, and Richard A. Kemmerer. A stateful intrusion detection system for world-wide web servers. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC 2003)*, pages 34–43, Las Vegas, NV, December 2003.

[17] William K. Robertson, Giovanni Vigna, Christopher Kruegel, and Richard A. Kemmerer. Using generalization and characterization techniques in the anomaly-based detection of web attacks. In *Proceedings of the Network and Distributed System Security Symposium (NDSS 2006)*, San Diego, CA, February 2006.

[18] M. Cova, D. Balzarotti, V. Felmetsger, and G. Vigna. Swaddler : An Approach for the Anomaly-based Detection of State Violations in Web Applications. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 63–86, Gold Coast, Australia, September 2007.