UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTACÃO

RAFAEL HANSEN DA SILVA

Implantation des services d'analyze statique sur des bundles OSGi

Monografia apresentada para obtenção do Grau
de Bacharel em Ciência da Computação pela
Universidade Federal do Rio Grande do Sul

Prof. Dr. Andrzej Duda
Orientador

Porto Alegre, julho de 2012

Resumo expandido

Orange (nome da marca da empresa France Télécom) é uma das principais operadoras mundiais em telecomunicação. Com quase 210 milhões de clientes, a operadora Orange fornece serviços de internet, redes móveis e televisão. Para oferecer esses serviços, a companhia provê equipamentos específicos para os seus clientes: o *Livebox* oferece serviços de roteador ADSL sem fio e telefonia sobre IP; o *Set Top Box* fornece serviços relacionados à televisão como os canais TNT em alta definição e a possibilidade de gravar os programas de televisão.

Com o objetivo de melhorar a qualidade desses serviços, Orange estuda como aplicar nestes equipamentos o sistema operacional Linux com a OSGi Service Platform. Neste sentido, a plataforma permite o gerenciamento multiaplicação, entre os módulos e a instalação transparente dos componentes. Essas características são consideradas essenciais para a companhia abrir a plataforma Orange para os desenvolvedores de software terceirizados.

Contudo, os aspectos relativos à segurança computacional devem ser considerados antes de abrir o desenvolvimento de aplicativos da plataforma. Estudos recentes [23,24] demonstraram que a plataforma OSGi possui uma série de vulnerabilidades, ignoradas até pouco tempo pelas empresas. Estas falhas, quando considerado o desenvolvimento de artefatos de software por empresas de terceiros, permitem aos provedores de software não fiscalizados ou desenvolvedores maliciosos explorar fraquezas como:

- Indisponibilidade do sistema ou da aplicação (ataque de negação de serviço ou DoS);
- Interferir nas saídas de outros componentes (ataque de saída errônea ou cavalo de tróia);
- Expor o sistema a atacantes maliciosos ou que desejam ganhar o controle do sistema para obter as informações do usuário sem a permissão do proprietário (ataque de exposição ou ataque a integridade dos dados).

Para prevenir um comportamento malicioso, uma ferramenta de validação está sendo desenvolvida na Orange Labs, com o objetivo de analisar todas as aplicações e bibliotecas feitas para os equipamentos da Orange para assegurar algumas propriedades de segurança de maneira estática, sem afetar o desempenho dos equipamentos. Por exemplo, a plataforma de validação deve assegurar que um *bundle* não irá exceder um tamanho especificado, evitando que um componente de tamanho grande afete o comportamento da OSGi Platform ou um ciclo entre serviços OSGi cause uma indisponibilidade dos serviços. Para atingir esse objetivo, alguns ataques direcionados para o OSGi serão estudados, assim como possíveis técnicas para evitá-los.

No quesito de segurança, OSGi Platform é executada sobre a plataforma Java herdando todos os conceitos de segurança utilizados nessa plataforma. A segurança computacional da plataforma aumenta, pois a JVM (tanto na linguagem de programação como no Java bytecode) não permite a execução de algumas operações perigosas ou não confiáveis, como modificar a pilha de execução ao prover a alocação de memória de maneira automática.

Neste contexto a plataforma possui, desde a sua versão inicial, uma forte preocupação com a segurança. Para alcançar esses objetivos, a plataforma Java provê um conjunto de API, controle de acesso, controle de código e verificação do bytecode Java. Exceto pelo conjunto de API, os outros recursos citados são uma importante garantia para segurança da plataforma OSGi [6]. Para alcançar o objetivo de executar códigos não confiáveis com segurança, toda ação executada por um aplicativo Java solicita a autorização do componente de controle de decisão. O controle de decisão de acesso é mediado pelo gerenciador de segurança, se o mediador estiver instalado. Para verificar se uma aplicação possui as permissões necessárias para executar seus comandos, o controle de acesso consulta as permissões Java. Além disso, o gerenciador de segurança usa a técnica conhecida como *Stack Inspection* [25] para analisar se algum componente não confiável está tentando executar uma operação a qual não possui privilégios através de serviços confiáveis. Brevemente, todas as classes serão avaliadas como componentes "*system*", confiáveis, ou "*untrusted*", não confiáveis, e uma marca é usada para indicar esta condição. Quando uma operação potencialmente perigosa é executada, a JVM verifica toda a sequência da pilha. Se existirem componentes "*untrusted*" na pilha de invocadores, o acesso será negado e uma exceção será disparada, senão a máquina virtual executará normalmente o método. Outra característica é a validação do código que verifica se o código ou bytecode Java respeita os padrões semânticos evitando determinadas operações perigosas. Além disso, a JVM provê gerenciamento de memória automático, *garbage collection* e verificação de limites de um array [6]. Também, a linguagem disponibiliza 4 tipos de níveis de acesso a classes, variáveis e métodos, sendo elas "*public*", "*protected*", "*package*" e "*private*".

Todas essas propriedades são checadas tanto no nível do código fonte quanto no nível de bytecode Java, cruciais para a plataforma OSGi que provê a possibilidade de instalar e executar serviços feitos por terceiros.

A plataforma OSGI é uma camada de componentização para a JVM. No sentido do desenvolvimento modular, as características mais importantes do OSGI são:

- Fornecer um ambiente multi-aplicação;
- Fornecer uma plataforma orientada a serviços;
- Permitir a instalação transparente, sem a necessidade de reinicializar a plataforma.

A característica mais interessante desta plataforma para a empresa Orange é: "A plataforma OSGi suporta em tempo de execução os aplicativos feitos em Java através de uma abordagagem modular: as aplicações são particionadas em *bundles*, que podem ser carregadas, instaladas e gerenciadas de maneira independente uma das outras" como citado em [24].

A plataforma OSGi criada pela OSGi Alliance estabelece uma especificação a ser seguida pelas implementações OSGi que fornecem serviços OSGi. Os frameworks OSGi mais conhecidos são: Apache Felix, Concierge, Eclipse Equinox, Knopflerfish.

Entretanto, as características fornecidas pela plataforma OSGi não são suficientes para a Orange Service Plataform. A plataforma também deve assegurar que os módulos instalados e a própria plataforma não sofrerão ataques. Neste sentido, a segurança do OSGi é baseada na herança das características de segurança da plataforma Java. Além disso, o OSGi beneficiado pelos estudos feitos para melhorar a segurança da plataforma Java, intro-

duz o conceito de isolação de "*namespace*" entre *bundles*, e teoricamente, permite somente o acesso dos pacotes declarados como pacotes "*export*" no arquivo *manifest*. Contudo, a plataforma herda também as falhas de segurança existentes na plataforma Java, além de amplificar ou introduzir novas falhas, por exemplo, o método "*System.out.exit*" que para a execução da plataforma OSGi.

Para suprimir essas falhas de segurança, a plataforma de validação proposta por esse trabalho tem como objetivo a verificação automática dos *bundles* OSGi e assegurar que os *bundles* não irão executar operações maliciosas nos equipamentos. Assim, a plataforma de validação aplicará um conjunto de testes para garantir que uma dada aplicação não tenha um comportamento malicioso. Estudos [23] mostram que o fato da plataforma OSGi ser baseada na plataforma Java representa uma importante garantia em termos de segurança. Entretanto, a plataforma Java não está livre de certas falhas na segurança.

Para solucionar este problema, este trabalho propõe o desenvolvimento de uma plataforma, de validação que realizará verificações automáticas sobre os *bundles* construídos por terceiros. Neste contexto, a plataforma detectará os acessos não autorizados que podem ser checados antes do tempo de execução.

Para construir uma ferramenta de validação para as aplicações feitas para a plataforma, é necessário estudar os ataques já descobertos em pesquisas anteriores. Neste sentido, este trabalho usa o catálogo de ataques [24] como referência para decidir quais vulnerabilidades estudar. O artigo estudado demonstra a existência de 32 vulnerabilidades sobre as plataformas OSGi. No artigo citado foi proposta a utilização das permissões Java para solucionar 13 vulnerabilidades. Entretanto, essa solução adiciona uma sobrecarga na execução, não atendendo os objetivos desse trabalho. Neste trabalho, 6 vulnerabilidades serão plenamente abordadas sendo elas: *Big Component Installer*, *Excessive Size of Manifest*, *Decompression Bomb*, *Duplicate Package Import*, *Cycle Between Services* e *Big File Creator*.

A falha *Big Component Installer* consiste em instalar um componente de tamanho considerável na plataforma. O principal efeito dessa operação maliciosa é ocupar um espaço excessivo na memória RAM e no disco rígido causando um DoS com a queda de desempenho da plataforma.

Para detectar um *bundle* com esta intenção, a plataforma analisa se o componente tem um tamanho maior que um valor específico (este valor depende das características dos equipamentos e pode ser mudado através de um arquivo de configuração).

A vulnerabilidade conhecida como *Excessive Size of Manifest* é causada por um *bundle* com um número grande de importações (mais que 1 MB) como definido em [24]. Esta falha explora as implementações Felix e Knopflerfish que trabalha sobre uma thread única. Adicionalmente, o *launcher* de processos precisa de vários minutos para instalar o *bundle*, por causa da necessidade de analisar sintaticamente um arquivo *manifest* de um tamanho grande.

O principal problema neste caso é a indisponibilidade da plataforma durante a instalação do componente malicioso. As implementações das plataformas OSGi citadas precisam executar o processo de início do *bundle* na mesma *thread* responsável por todos os componentes da plataforma. Por causa disso, o ataque é classificado como um ataque de negação de serviços com consequência de indisponibilidade.

Para detectar este ataque, a plataforma de validação consulta o tamanho do arquivo *manifest*, utilizando um serviço de descompressão de arquivos, no *bundle*. Se este componente possuir mais do que o limite especificado será considerado como um *bundle* malicioso, nos testes foi definido mais do que 1 MB.

O ataque conhecido como *Decompression Bomb* é uma variação do ataque clássico usado em outras plataformas. A ofensiva consiste em realizar uma compressão nos arquivos JAR de um arquivo com um grande número de dados repetidos ou correlacionados. Por exemplo, uma imagem BMP preenchida apenas de vermelho com 1920 x 1080 pixels de dimensão, 24 bits por pixel tem um tamanho de 6075,05 KB, em torno de 1000 vezes maior que o seu tamanho comprimido.

O efeito desse serviço malicioso é ocupar um espaço considerável na memória RAM. Além de usar muito dos recursos da CPU durante o processo de descompressão. Depois da descompressão, o *bundle* pode consumir uma parte importante da memória no disco rígido, com a possibilidade de gerar o mesmo efeito, causando um DoS com o efeito de queda de desempenho.

Para detectar a exploração dessa falha, primeiramente foi obtida a taxa de descompressão padrão dos arquivos JAR na JVM SE 1.6 e nos bundles OSGi da companhia Orange, onde se constatou a taxa máxima de 10 vezes o tamanho do arquivo comprimido. Neste sentindo, foi construído um verificador que obtém a taxa de descompressão do arquivo e um bundle de descompressão que permite descomprimir todos arquivos e bibliotecas JAR existentes de maneira recursiva. Caso seja detectado um bundle com a taxa de descompressão acima 15 vezes, este será rejeitado pela plataforma.

O *Duplicate Package Import* consiste em adicionar um pacote de importação duas vezes, ou mais. Como consequência, o *bundle* com um *Duplicate Package Import* não pode ser instalado, bloqueando a execução de serviços. Por causa disso, este ataque é classificado como uma negação de serviço pelas pesquisas em que se baseia esse trabalho. Para detectar esta violação, a plataforma de validação analisa sintaticamente o arquivo *manifest* para obter todos os campos de "*Import-Package*", utilizando o método *getEntries()* da classe *java.util.jar.Manifest*. Depois dessa verificação, a plataforma rejeita o *bundle* no caso onde os pacotes são declarados mais que uma vez.

O *Cicle Between Service* é baseado em dois ou mais serviços que pertencem a diferentes pacotes e consiste em oferecer um serviço falso B a um serviço legítimo A. Quando um serviço A requisita o serviço B para ser executado, B requisitará a execução do serviço A. O serviço falso chama o serviço A, formando um ciclo de execução infinita entre os serviços A e B. Este evento, dada determinadas condições, pode causar a indisponibilidade de um evento legítimo ou pior uma indisponibilidade em cascata de vários componentes relacionados.

Embora seja possível evitar a instalação de componentes que explorem essa vulnerabilidade, não se pode fazer total distinção entre componentes maliciosos e componentes que não seguem as recomendações feitas pela OSGi Alliance. Pois, certos tipos de ciclos serão revelados apenas em tempo de execução. Entretanto, é possível afirmar se um *bundle* é seguro ou se ele é potencialmente perigoso.

Com o objetivo de detectar a exploração dessa falha, a plataforma de validação checa

primeiramente se há um ciclo entre as importações dos componentes e depois se há um ciclo entre serviços. Na primeira fase, na qual se detecta se há ciclos entre as importações, são verificados os arquivos *manifest*, utilizando a classe *java.util.jar.Manifest*, e as classes de todos os *bundles*, através da biblioteca ASM [22], instalados na plataforma e do componente que se deseja testar, para coletar as importações. Baseado nessa informação um grafo direcionado será gerado. Caso haja mais ciclos de importação que os nativos da plataforma, o algoritmo passará à segunda fase, senão o algoritmo termina a sua execução.

Na segunda fase, a plataforma verifica os ciclos de importação detectados e os analisa para checar se existem ciclos entre os serviços em duas subfases, com o auxílio da biblioteca ASM. Na primeira etapa somente informações como: nome da classe, que métodos uma classe especifica possui e a que *bundle* a classe está associada. Na segunda etapa, as mesmas classes serão analisadas novamente para obter as relações entre classes (interfaces, subclasses e superclasses) e métodos, para descobrir quais métodos são necessários para executar determinado método.

Baseado nas informações coletadas, um grafo será produzido para checar todos os traços de invocação de métodos em potencial. Novamente será aplicado o algoritmo para checar se há um ciclo no grafo. Caso haja, será checado se há algum ciclo malicioso no conjunto de *bundles* testados. Se não houver ciclos maliciosos, o *bundle* é considerado seguro, caso contrário a plataforma considera como um *bundle* potencialmente perigoso sendo descartado pela plataforma de validação.

A vulnerabilidade *Big File Creator* consiste em um *bundle* malicioso que cria um arquivo de tamanho grande (relativo aos recursos disponíveis) para consumir espaço no disco rígido. O dano causado pelo *bundle* bloqueia a instalação de novos *bundles* na plataforma. Então, qualquer aplicação que precisa usar a memória no disco não funcionará. O efeito causado é uma negação de serviço com o efeito de queda de desempenho.

Os dois métodos de detecção são limitados. A primeira abordagem é analisar o Java bytecode para encontrar os comandos de escrita que utilizam a classe *FileOutputSteam* do Java SE e analisar o tamanho de todos os arquivos que serão criados. O problema neste processo é que o *bundle* responsável pela criação pode ocultar o tamanho das entradas em outro método e realizar muitas operações para dificultar a análise. O segundo problema é que o tamanho do arquivo pode ser baseado em um valor, variável, evitando a possibilidade de detectar este ataque de uma maneira estática.

O segundo método simula a execução do *bundle*. Entretanto, este procedimento não pode detectar o caso onde existe um comando *if* que será somente ativado em caso de um evento específico, como uma data, ou a situação onde o tamanho pode ser mudado. Contudo, esta falha pode ser prevenida usando as permissões Java, um espaço de disco específico para usuário e/ou *bundle* ou limitar o acesso ao sistema de arquivos.

A arquitetura da plataforma de validação de bundles foi baseada na arquitetura da OSGi Platform. Isto permitiu o desenvolvimento de um software baseado nos princípios da Arquitetura Orientada a Serviços, também conhecida como SOA. Dessa maneira, os diferentes verificadores responsáveis por detectar a exploração de falhas puderam ser implementados como um bundle OSGi indepedente. Este bundle, apenas chama os serviços oferecidos pelos outros bundles da plataforma, como mostrado pela figura 1. Permitindo

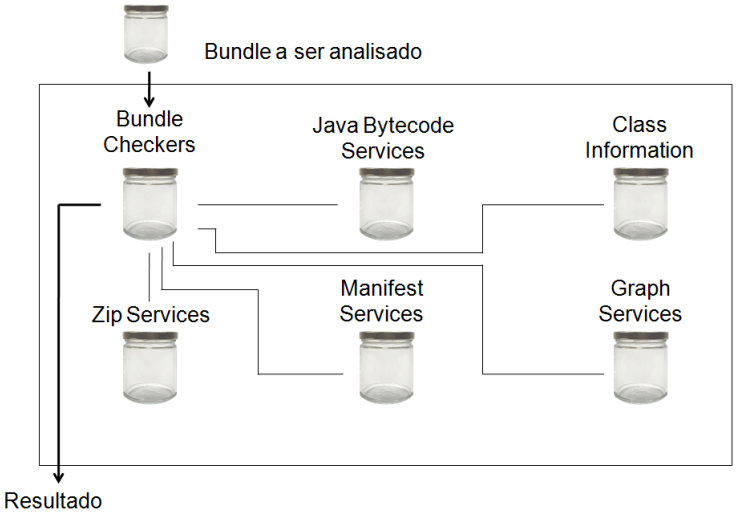

Figura 1: Arquitetura da plataforma de validação.

que todos os outros bundles implementados sejam utilizado em outro contexto. Por exemplo, o bundle Zip Services pode ser chamado por um aplicativo para realizar um serviço de descompressão que não esteja relacionado com a plataforma de validação. Neste sentido, a plataforma foi dividida nos seguintes bundles (apenas os bundles relacionados a esse trabalho foram citados):

- Bundle Checkers: possui o conhecimento necessário para realizar a detecção de 5 vulnerabilidades. Para isso, utiliza os serviços dos bundles mostrados na figura 1;

- Java bytecode Services: implementa os serviços de verificação e obtenção de informação no Java bytecode;

- Class Information: serviços com o propósito de organizar as informações coletadas referentes as classes (herança e interfaces) e métodos analisados (traço de execução do método);

- Graph Services: provê classes para manipular e extrair informações de um grafo. Por exemplo, quantos ciclos existem em um determinado grafo, quais nodos e arestas formam estes ciclos;

- Manifest Services: obtém as informações relacionadas ao arquivo manifest de um bundle;

- Zip Services: extrai arquivos relacionados a um bundle. Possibitando a extração recursiva de todos os arquivos;

- Test Checkers: este bundle é responsável por realizar os testes de integração da plataforma de maneira automática. Todos os outros bundles possuem classes que realizam testes automáticos referentes aos de unidade.

Relativo aos testes de unidades, foi utilizado a metodologia de desenvolvimento dirigido a testes. Com o objetivo de direcionar o desenvolvimento dos componentes. Além de fornecer uma garantia para a companhia de que ao implementar uma nova versão do módulo, este seguirá as especificações inicialmente propostas. Durante a execução dos testes de

unidade, encontraram-se alguns erros nos componentes.

Um deles, diz respeito ao módulo de descompressão que retornava o tamanho total do arquivo descomprimido diferente do esperado. Esse problema foi ocasionado, pois como o código responsável pelo teste executava um software de descompressão no sistema operacional Linux. Como o sistema de arquivo ext4 considera uma pasta como se fosse um arquivo, cada pasta descomprimida, adicionava 4 KB no total esperado do tamanho da descompressão.

Outro exemplo de teste que ajudou a descobrir um erro nos verificadores, foi um bundle que explorava a falha *Cycle Between Services*. Na qual, a classe utilizava o conceito de herença para ocultar a invocação de um traço de execução que gerasse um ciclo. Isso ocorria, pois uma variável do tipo A, recebia um objeto do tipo B, sendo B uma classe filho de A. Nessa situação, o verificador de ciclos analisava os métodos da classe A ao invés de analisar os métodos da classe B.

Relativo as ferramentas, as utilizadas foram:

- A biblioteca ASM do Java que permitiu a analise do bytecode;
- O ambiente de desenvolvimento Eclipse IDE que permitiu a integração das ferramentas utilizadas;
- O software Maven que permitiu o gerenciamento e construção de projetos automatizados;
- O software SVN de controle de versão e de revisão de projeto;
- O Framework de teste JUnit que permitiu o desenvolvimento dos teste unitários e a sua verificação automática.

O método de escolha das vulnerabilidades a serem estudadas, se baseou nos métodos ágeis, onde cada membro escolhe a atividade que tem condições de solucionar. Na companhia existe uma página intranet para cada projeto, onde se pode verificar as atividades pendentes. No meu caso, escolhi os métodos citados previamente nesse trabalho, com a autorização do meu responsável técnico. Sendo as falhas de baixa dificuldade estudadas no início do estágio e no seu decorrer as consideradas mais complexas. Permitindo o estudo do funcionamento da plataforma OSGi, em paralelo ao desenvolvimento da implementação da plataforma de verificação.

Após realizado o trabalho foi constatado que a plataforma OSGi oferece um ambiente multi-aplicação, isolação entre módulos, extensibilidade dinâmica e instalação transparente de componentes. Estas são todas as características desejadas pela Orange com o objetivo de abrir o desenvolvimento de software por terceiros.

Além disso, a plataforma OSGI fornece muitos atributos de segurança, desde isolação entre *bundles* até a herança de todos os mecanismos de segurança como gerenciamento de memória. Apesar dessas características, a plataforma possui vulnerabilidades que não podem ser ignoradas. Estas vulnerabilidades requerem a modificação e/ou a integração de novas ferramentas de segurança para a OSGi Plataform para incrementar a segurança ou solucionar as vulnerabilidades conhecidas.

Neste sentido, o trabalho desenvolveu uma maneira de aumentar a segurança da plataforma, usando verificadores automáticos. A plataforma de validação consiste na análise do bytecode de maneira estática para detectar possíveis explorações das falhas de segu-

rança. Também foram discutidas as razões que, em alguns casos, limitam a detecção da exploração de falhas de forma estática. Mesmo que a ferramenta de validação não detecte todas as possíveis formas de explorar as falhas, a plataforma de validação é uma importante ferramenta complementar de segurança. A plataforma pode evitar que algumas técnicas de detecção em tempo de execução sejam utilizadas reduzindo a sobrecarga no OSGi. Por exemplo, ao utilizar a plataforma de validação, não é necessário guardar e testar todos os traços da pilha de serviços para detectar a vulnerabilidade do *Cycle Between Service* em tempo de execução. A seguir será apresentado o trabalho completo, escrito em inglês, correspondente ao projeto de fim de estudos pela universidade Grenoble INP - Ensimag. Durante o estágio realizado nos meses de fevereiro à agosto de 2011 na empresa Orange Labs, permanecendo a pesquisa em andamento após o término do meu estágio. Os detalhes da implementação, como o código fonte, não foram inseridos no trabalho. O motivo foi o termo de confidencialidade assinado por mim para desenvolver o projeto proposto pela companhia.

Grenoble INP – ENSIMAG
École Nationale Supérieure d'Informatique et de Mathématiques Appliquées

Rapport de projet de fin d'études

Effectué á Orange Labs

Implantation des services d'analyze statique sur des bundles OSGi

HANSEN DA SILVA Rafael
3e année – Option ISI

21 Février 2011 – 19 Aôut 2011

Orange Labs
28 chemin du Vieux Chêne
BP 98
38243, Meylan

Responsable de stage
KOPETZ Radu
Tuteur de l'école
DUDA Andrzej

Abstract

Orange (the brand name of France Télécom) provides mobile network, television and Internet services. In order to offer these services, Orange has developed equipment such as the Livebox and the Set Top Box. However, to maintain the quality of service in these equipment, the company researches methods that allow the integration of new features in its equipments, as well as of services developed by third parties.

To reach this purpose, Orange studies the possibility of introducing in their equipment an OSGi Service Platform, a dynamic module system for Java™, that supports the dynamic and transparent installation of components. This platform would allow the development of applications in a modular fashion, while offering strong isolation mechanisms. This is necessary to open the Orange Platform for the software developed by third-party providers.

In the meantime, recent studies have demonstrated that the OSGi Service Platform possesses a considerable number of security weaknesses. These vulnerabilities, if ignored, can be exploited by the malicious components to block or interfere with Orange services, or to obtain sensitive information from the Orange customers. Something that cannot be neglected by the enterprise.

To avoid these problems, a validation platform will be developed that will analyze all the applications developed for Orange equipment, in order to ensure security properties, such as a non-malicious behavior. For example, the validation platform needs to detect a situation where the application wants to occupy a considerable part of the computational resources such as the processor, the RAM, the hard drive, among others. If the constructed platform built does not detect any malicious behavior, Orange will give a certification to the third-party application.

Key words: OSGi™Platform, OSGi™Platform vulnerabilities, Java Platform, isolation, security, malicious application, validation platform.

Coordinates

Author	**Internship Supervisor**	**Teacher Supervisor**
HANSEN DA SILVA Rafael Ensimag student Phone: 07 86 42 97 16 Email: Rafael.Hansen-Da-Silva@ensimag.imag.fr	KOPETZ Radu Software architect Phone: 04 76 76 45 09 Email: radu.kopetz@orange-ftgroup.com	DUDA Andrzej Ensimag Professor Phone: 04 76 82 72 87 Email: andrzej.duda@imag.fr

Résumé

Orange (le nom de marque de France Télécom) fournit des services de réseaux mobiles, télévision et Internet. Afin d'offrir ces services, Orange a développé des équipements tels que la Livebox et la Set Top Box. Cependant, pour maintenir la qualité des services de ces équipements, l'entreprise recherche une méthode qui permet l'intégration de nouvelles fonctionnalités, et aussi de services développés par des tiers.

Pour atteindre cet objectif, Orange étudie la possibilité d'introduire dans ses équipements une plate-forme de services OSGi, un système de modules dynamiques pour Java™ qui offre la possibilité de faire des installations dynamiques et transparentes des composants. Cette plate-forme permettra le développement modulaire des applications, qualités nécessaires à des developeurs tiers.

Cependant, des études récentes ont démontré que la plate-forme OSGi possède un nombre considérable de failles de sécurité. Ces vulnérabilités, si elles sont ignorées, peuvent être utiliseés par les composants malveillants pour bloquer ou interférer avec les services d'Orange, ou pour obtenir des informations sensibles des clients Orange.

Pour éviter ces problèmes, une plate-forme de validation sera développée pour analyser toutes les applications développées pour les équipements d'Orange afin d'assurer des propriétés de sécurité, comme un comportement non malveillant. Par exemple, la plate-forme de validation doit détecter une situation où l'application veut occuper une partie considérable des ressources de calcul comme par exemple le processeur, la mémoire vive, le disque dur, etc. Si la plate-forme construite ne trouve pas un comportement malveillant, Orange donnera une certification pour l'application.

Mots clés: OSGi™ Platform, les vulnérabilités dans OSGi™ Platform , Java Platform, isolation, sécurité, les applications malveillantes et plate-forme de validation.

Contents

List of Figures

List of Abbreviations

ACM	Association for Computing Machinery Digital
API	Application programming interfaces
BMP	Bitmap Image File
CENG	*Centre d'études nucléaires de Grenoble* - Center for Nuclear Studies of Grenoble
CEO	Chief executive officer
CPU	Central processing unit
DoS	Denial of service
HD	High definition
IDE	Integrated development environment
INRIA	*Institut national de recherche en informatique et en automatique* - National Institute for Research in Computer Science and Control
JAR	Java archive
Java SE	Java platform, standard edition
JDK	Java development kit
JRE	Java SE runtime environment
JVM	Java virtual machine
KB	Kilobyte
MB	Megabyte
OS	Operating system
OSGi	Open services gateway initiative

POM	Project Object Model
R&D	Research and development
RAM	Random-Access memory
S.A.	*Société anonyme* - Corporation
SDK	Software development kit
SVN	Apache subversion
Télécom	*Télécommunication* - Telecommunication
TDD	Test-driven development
Telecom	Telecommunication
TNT	*Télévision numérique terrestre* - Digital terrestrial television

Chapter 1

Introduction

1.1 Motivation

Orange is one of the world's leading telecommunications operators [15]. With almost 210 million customers, the Orange, the brand of France Télécom, provides Internet, television and mobile services. To offer these services, the company provides specific equipment: the Livebox supplies Internet services like ADSL wireless router and VoIP; the Set Top Box offers the television services as the TNT channels in HD and the possibility to record television programs.

To improve the quality of these services, Orange studies how to apply in these equipments the Linux OS with the OSGi Service Platform. In this sense, the OSGi Service Platform enables multi-application management, between the modules and the transparent installation of components. These characteristics are considered essential by the company to open Orange's platforms to software developed by third-party vendors.

However, the security aspects must be considered before opening the software development. Recent studies [23, 24] have demonstrated that the OSGi Service Platform possesses a series of vulnerabilities, ignored until recently by the enterprises. These flaws, when considering the development of software artifacts by third-party companies, allow the uncontrolled providers or malicious developers to exploit weaknesses such as: the system or application availability (DoS attack); to interfere in the other components outputs (Erroneous Output attack); and to expose the system to malicious attackers that wish to gain the system control or to obtain the user information without permission from the owner (Exposure attack).

To prevent a malicious behaviour a validation tool is being developed at Orange Labs, with the objective to analyze all the applications and libraries done for the Orange equipment to ensure some security properties. For example, the validation platform must ensure that a bundle will not exceed a specified size, avoiding that a bundle with a big size affects the OSGi Platform behavior or that cycle between the OSGi services cause the unavailability of the services.

If the applications have all the properties required by Orange, the validation platform

will provide a certification to the software. To reach this goal, some attacks were studied that target OSGi platform together with possible detection techniques.

The report is structured as follows. Chapter 1 presents the enterprise's interest in constructing the validation platform. Chapter 2 presents France Télécom and Orange Labs. Chapter 3 describes the context of this work and the related works. Chapter 4 contains the internship objectives, specification of the project, the architecture specifications, the methodology and the tools used. Chapter 5 exposes the OSGi and Java vulnerability studies during the internship and discusses the algorithms used to detect them, if it is possible. Chapter 6 shows the resources provided by the company and the work plan represented as a Gantt Chart. Finally, Chapter 7 summarizes the internship work.

Chapter 2

About the enterprise

2.1 France Télécom and Orange

France Télécom S.A. is the main telecommunication companies in France and eighth largest telecom operator in the world. Headed by the Stéphane Richard, see Figure 2.1, the company currently employs around 169 thousand people and has 210 million customers, as of 2011 [15].

France Télécom was founded in the 1988 by the France Government as a division of the Ministry of Post and Telecommunications. In 1990, the company gained autonomy from the state and was transformed in a company to explore the public services. In 1996, France Télécom was transformed into a corporation where the French Government was the unique shareholding, to prepare the company for the opening of the telecommunications market that occurred in January 1, 1998. In 2000, with the Internet growth, France Télécom bought the French and British enterprise Orange, responsible to offer Internet and Mobile services, and has became the main brand of France Télécom. In September 2004, the French State sold part of its stake and was no longer the majority shareholder. In the present, the enterprise is presence 220 countries and territories worldwide, counting on with many research centers around the world.

2.2 Orange Labs

In order to maintain quality of its services and improve the technologies in the equipments of the customers, Orange Labs, which was called France Télécom R&D before January 2007, is the research and development division. This division participates of the Europeans programs in the community[2].

Orange Labs network includes more than 5000 collaborators which 3000 engineers and researchers, from 18 centers, predominantly in France (Issy-les-Moulineaux, Lannion, Grenoble, Caen, Rennes, Sophia Antipolis, La Turbie and Belfort) and abroad: Poland, United Kingdom (London), Spain (Madrid), Egypt (Cairo), Japan (Tokyo), Jordan, Korea, China (Beijing), USA (San Francisco and Boston) [18], see Figure 2.2.

Figure 2.1: Orange organization [2].

Also, Orange Labs has important partners in the telecommunications and information system field. Some important partners are the Object Web, that France Télécom is co-founder with Bull and INRIA, with Microsoft in projects in the multimedia domain, and in other domains, with Ericsson, Motorola, Nokia and Intel.

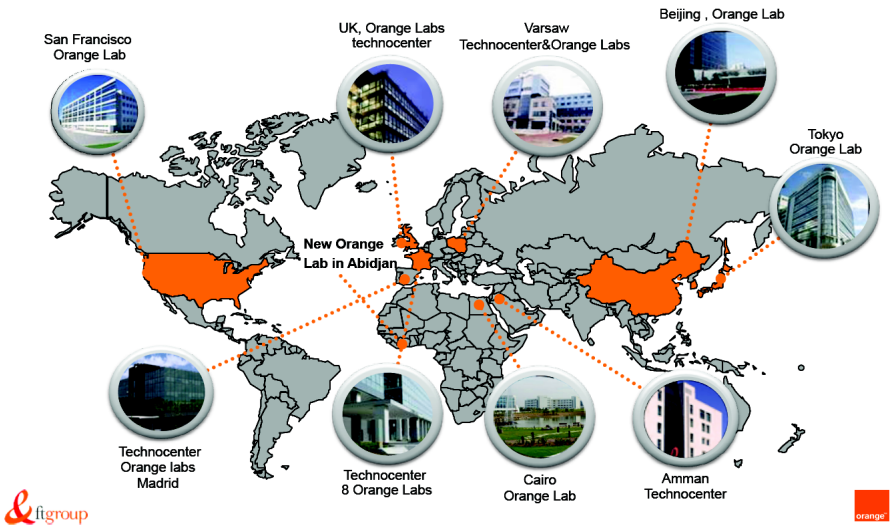

Figure 2.2: Orange Labs around the world [8].

2.3 Grenoble Orange Labs

Located in Meylan city in Rhône-Alpes region, the Grenoble Orange Labs, a member of Orange Labs network, established since 1983 and it is responsible for three main objects of study:

- The communicants object and the terminals including the impact of nano and biotechnology and machine-to-machine applications;

- Health public services, local authorities. Partnerships have been developed with the University Hospital of Grenoble, the CENG, INRIA, with local and regional authorities;

- Software technologies, which are developed architectures and engineering services for Orange.

Chapter 3

Context and related works

3.1 Context

3.1.1 Java Platform

Java Standard Edition is a platform with two main products: Java Development Kit (JDK) and Java SE Runtime Environment (JRE). The JDK contains everything that is in the JRE, plus the tools necessary for developing applets and applications. The Java Runtime Environment (JRE) provides the libraries, the Java Virtual Machine (JVM), the components to run applets and applications written in the Java programming language [7]. This structure can be seen in Figure 3.1.

More specifically, the JVM consists of a virtual machine that runs the Java bytecode (instruction set for the JVM) with libraries set to run the programs. The virtual machine works on the Operating System (OS) layer avoiding that the software has a direct access to the OS. The principal features of the platform are the security and the code portability. The portability aspect is granted by its different version for the most used OS in the present, avoiding the necessity to compile the same software for all target platforms. The platform security is increased because the JVM (both in the programming language and in the Java bytecode level) does not enable to do some dangerous/untrusted operations, e.g. to modify the execution stack.

The platform possesses, since the initial version, a strong preoccupation with the security, as, one of the Java platform's goals is to run untrusted code, such as Java applets downloaded from a public network, without any risk for the users. To reach this objective, Java Platform provides API set, access control, code and Java bytecode validation. Except by the API set, that assists the developer with a set of security tools, the other features cited are important to guarantee the OSGi platform security [6].

In the platform, the access control architecture purposes to protect the resources and the application code, originally with the aim to avoid that untrusted downloaded applets to cause any damage in the user system. To achieve this, all access control decision made by Java are mediated by the security manager, if the manager was installed into the Java runtime. To verify the application grants and policy, the access control consults the Java

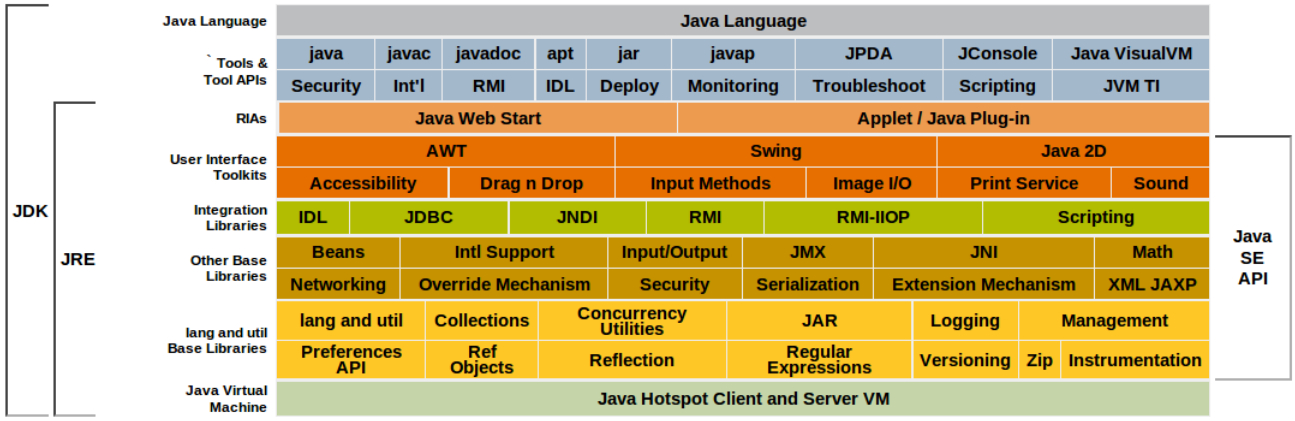

Figure 3.1: Java SE [7].

Permissions, responsible to specify the application access, to verify if the application possesses the authority to realize a determinate action. The technique used by the security manager to perform this control is the Stack Inspection [25]. In short, all the classes are evaluated as a "system" or "untrusted" components, and a flag is used to indicate this condition. When a potentially dangerous operation is executed, the JVM verifies all the stack sequence. If there are untrusted component on the callers stack the access will be denied and an exception will be thrown. Else, the machine will execute the method.

Complementing the access control architecture, the code validation verifies if there is any code or Java bytecode violates the semantic patterns. The most widely known pattern is the type safe feature that helps to prevent frequent faults on the non-type language, improving the safety, but also increasing the platform security. This characteristic prevents common attacks, as buffer overflow are used to obtain protected data or to modify the software execution. To support this feature, the JVM provides automatic memory management, garbage collection, and range-checking on arrays [6]. In addition, the Java language provides four distinct access levels for classes, variables and methods: private, protected, public and package. The most restrictive level is the private access that prevents the access for the class outside and the less restrictive level is the public level that grants the access to anyone.

All these properties will be checked in the code level and in the Java bytecode level and are crucial for the OSGi platform that provides the possibility to install and to execute services made from third-party.

3.1.2 OSGi Service Platform

The OSGi Service Platform is a componentization layer to the JVM [20, 13, 24, 9]. Orange is interested in this platform because: "The OSGi Platform supports the runtime extension of Java-based application through a modular approach: the applications are parted into "bundles", which can be loaded, installed and managed independently from

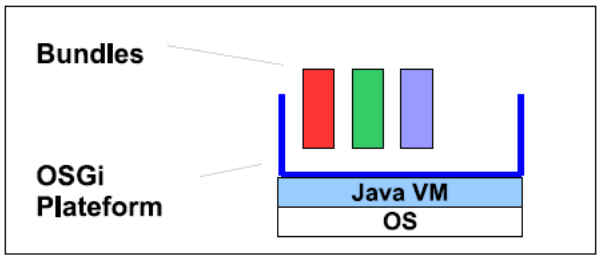

Figure 3.2: OSGi Platform [24].

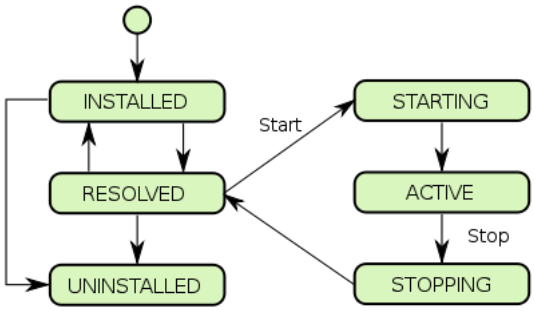

Figure 3.3: OSGi bundle life cycle [3].

each other" as [24].

The OSGi Platform establishes a specification to be followed by the OSGi implementations that supplies the OSGI services. The most known OSGi frameworks are: Apache Felix, Concierge, Eclipse Equinox, Knopflerfish.

"Three main concepts sustain the OSGi platform functionalities: the platform, the bundle, and the interoperability between the bundles" [24]: the platform is responsible to manage the application executions; the bundle is the unit that can represents a software, a library or a set of services, with similar responsibilities that a standard JAR file; and the last main functionality concept, the interoperability between the bundles, is reached at the class level (for example, two classes from different bundles can belong to the same package accessing classes of different bundles) and at the service level (requesting the services registered by other bundles). Furthermore, the bundles can provide services and/or request services declared as public from the others bundles running independently on the OSGi Platform, and on the JVM as a shown in Figure 3.2.

To support the dynamic bundle installation and update, the platform supervises the bundle lifecycle, that can have six possible states: installed, resolved, starting, active, stopping and uninstalled, as a shown in the Figure 3.3. Only in the active state a bundle can provide its services to other bundles on the platform.

In addition, to obtain more information about the bundles running on the platform, the manifest file used by the OSGi platform expands the Java Manifest File concepts. An

```
1       Bundle-Name: Service listener example
2       Bundle-Description: A bundle that displays messages at startup and when
3       service events occur
4       Bundle-Vendor: Apache Felix
5       Bundle-Version: 1.0.0
6       Bundle-Activator: tutorial.example1.Activator
7       Import-Package: org.osgi.framework
```

Figure 3.4: OSGi Manifest File [10].

OSGi manifest file example is Figure 3.4 where the "Bundle-Version", "Bundle-Activator" and "Import-Package" entries will be used to give import informations about the bundle for the platform. The "Bundle-Version" gives information about the bundle version, this data is used in order to allow the execution, at the same time, of two or more bundle version. The "Bundle-Activator" entry informs the name of the class responsible to register the bundle services. The "Import-Package" exposes to the platform all the necessaries packages to execute the bundle. Other two importants optional entries are: the "Export-Package" that communicates all the packages that will be exposed by the bundle; the "Bundle-SymbolicName" that together with the "Bundle-Version" generate a unique identifier for each bundle.

Nevertheless, these features are not sufficient for the Orange Services Platform. The platform must ensure that the modules and itself will not suffer an attack. In this sense, the OSGi security is based on the JVM security aspects, once the platform executes on the JVM and inherits its security features. Thanks to this inheritance of all Java security characteristics a malicious bundle, for example, cannot generate a buffer overflow typically designed to execute malicious code. Also, OSGi platform benefits by all the security studies on security and exhaustive tests done on the JVM level. Furthermore, the platform introduces new security aspects as a proper namespace isolation between bundles and, in theory, only the access to the packages declared as an "export" package in the manifest file.

On the other hand, all the Java flaws are inherited, moreover the platform introduces other vulnerabilities in the system or amplifies the consequences of some vulnerabilities. For example, the *System.out.exit* method stops the OSGi platform execution. Some of these weaknesses will be explored in this work.

3.1.3 Security using Java Permissions and Policy

The Java Permission and Policy allows restricting the untrusted component actions. However, as observed by Frénot and Parrend at [24], only 13 of 32 vulnerabilities found are covered by Java Permissions. Furthermore, Java Permissions can be used at runtime and they introduce an important overhead.

3.1.4 Catalog of vulnerabilities on OSGi platform

This work is based on the OSGi vulnerability catalog done by the INRIA in 2007 [24] including examples of how to explore the OSGi flaws. The article objective is to support and stimulate the security works in development to improve the OSGi platform security. In this paper, INRIA uses a Semi-formal Vulnerability Pattern defined by the authors, which provides information about: the vulnerability reference, the vulnerability description, the protection or possible mechanisms to realize the protection and the reference about a vulnerability exploit.

Unfortunately, not all of the vulnerabilities treated in the publication contain a solution how to solve the vulnerability exploits. For this, the INRIA research was used with starting point to decide which flaws would be chosen to be studied. Moreover, the vulnerability exploits implemented by the French Institute was submitted to the platform tests to certificate that the software detects all the expected flaws.

3.2 Related works

In this section will be exposed two works that apply different approaches to reach the goal to improve the OSGi framework security. I-JVM applies the isolation methodology and AOSL introduces a new security layer at the platform.

The I-JVM work [19] proposes to modify JVM code, to create a total isolation between the components with a little overhead and to ensure the compatibility with the current bundles. Some features were added by the I-JVM work.

The first feature, memory isolation, grants the isolation of *java.lang.Class* object, strings and static variables private.

The resource accounting feature allows obtaining the amount of resources used by a thread. The information can be used to detect DoS attacks.

Finally, the termination of isolates blocks the execution of the set of classes that already finished their execution. The paper obtained success to prevent eight vulnerabilities. Moreover, the benchmark test executed by the researches showed and overhead below 20% for all benchmarks. However, there is the necessity to modify the JVM code.

The Advanced OSGi Security Layer [21] provides an additional security layer in the OSGi framework with the purpose of preventing services from attacking the platform. The proposed prototype focuses on threats such as the denial of service and the shared object attack is located between the OSGi framework and the JVM. The idea is to extract information from JVM and collaborates with OSGi framework rule-based policies through of 6 modules: the Event Extractor module can manipulate the JVM to notify the module when a specific event occurs; the Audit Data Processor and Logging module responsible to filter the irrelevant data; the Detection Engine module possesses the objective inspects the audit data and the log; the response Manager module recognizes illegal and harmful actions and notifies the Advanced Security Manager of these actions; the last module, Advanced Security Manager, is a GUI for the users.

This work solves all the problems proposed in runtime. However, the overhead is greater than 50% and the techniques cannot be applied on a static validation tool.

Chapter 4

Specification of the project

4.1 Objectives

The validation platform's objective is the automatic verification of OSGi bundles in order to ensure that the bundles will not perform malicious operations on Orange equipment. Thus, the verification platform will apply a test set to guarantee that a given application does not have a malicious behavior.

The studies [23] showed that the fact OSGi Platform is based on Java platform, represents an important guarantee of security terms. However, the Java platform is not free of security flaws, this reality is aggravated by the security vulnerabilities introduced by the OSGi platform, e.g. the use of the *System.out.exit* method.

To solve this problem, the internship proposes the development of a validation platform that will realize the automatic verifications on the bundles constructed by third-parties. In this context, the platform would detect attempts of unauthorized accesses that can be checked before runtime.

The tests will cover simple vulnerabilities, such as Duplicate Import in the Manifest and more complex vulnerabilities, such as the Cycle Between the Services. These types of analyzes are important to allow Orange to open the software development to third-parties keeping the security and the quality of service in its equipment (Livebox, Set Top Box, etc). In this section we present the validation platform architecture, development methodology and the used tools.

4.2 Architecture

The OSGi Platform was chosen as the architecture of the validation platform. This choice was motivated by the need of a modular service based architecture. This way different checkers can be implemented as independent OSGI bundles, benefiting from other services on the platform in a low-cohesion, low-coupled manner.

Another advantage is the extensibility, allowing for new checkers to be added without difficulties. The validation architecture is applied to the service, the interface and the reg-

ister concepts. The service concept offers the possibility of an application on the platform to invoke public methods from other modules to request the execution of tasks that are necessary for its execution. For example, in the validation platform, the module responsible to apply the validation software needs, in some tests, to decompress the JAR file to execute the verification. Therefore, the platform invokes the zip module to realize this task and after the platform will run the tests in the decompressed data, avoiding the necessity of other bundles to implement its own decompression tool for the JAR file.

The interface concept offers the possibility to separate the implementation from the service, characteristic that improves the security and decoupling. For example, the validation platform provides an interface for the zip module. With this, the bundles installed in the platform need to call the methods described by the interface zip module to perform the services provided by zip module. Because of this, the zip module can be modified or replaced by another bundle that implements the interface. Once, the service is not strongly linked to the implementation.

The register concept allows the dynamic exposure/withdrawal of the services. More specifically, if a bind method for a specific service is defined, it will be called when the required service reference is available, and the unbind method when the required service is no longer available. It is important, for example, to the other modules discover if a service is available or not. In our example, the dependent bundles will go to wait in the resolved state until the decompression tool obtains the active state. In this sense, the platform was divided into the following bundles (only the bundles related to this work are shown):

- Bundle Archive Checkers: use the services exposed by other bundles in order to see if a bundle has a malicious exploit or no and exposes these verifications as OSGi services;

- Bundle Checkers Interfaces: offers the interfaces necessaries to access the Bundle Archive Checkers;

- Java bytecode Services: offers the necessary interfaces to access the Java bytecode Services Implementation;

- Java bytecode Services Implementation: implements all Java bytecode services necessaries to obtain relevant informations in the Java bytecode;

- Class Information Services: offers the interfaces necessaries to access the Class Information Implementation;

- Class Information Implementation: implements the service with the purpose to organize the informations collected in the classes.

- Graph Services: exposes all the services implemented by the Graph Services Implementation;

- Graph Services Implementation: provides the graph structure and methods to manipulate and extract information on graphs.

- Manifest Services: exposes all the services implemented by the Manifest Services Implementation;

- Manifest Services Implementation: extracts the information related to the bundle manifest and offers a service to recovery these information.

- Zip Services: exposes all the services implemented by the Zip Services Implementation;

- Zip Services Implementation: extracts the information and the files related to a bundle and offers a service to recovery these information.

- Test Checkers: This bundle is responsible to validate the platform and test the integration between the bundles implemented automatically. All the other bundles also have automatic tests, but only in order to test their own execution.

4.3 Development methodology

The OSGi principles, modularity and the service calls, are applied to make the validation platform. For example, the tools separate the module test from all the API necessary to realize the tests. This technique improves the reliability of tests, the easy maintenance and extensibility of the platform.

In order to endorse the reliability of the application development the Test-driven development principle [17] was applied, during the stage. In other words, before doing the task, the developer prepares all the tests to check the new functionality. Only after the new feature is totally implemented and the tests return a positive answer, other features will be implemented as showed in Figure 4.1. The aim of this method is to make the developer to think about the specification and check if the developer understood what needs to be done. In addition, TDD give more guarantees that the code will follow the specification. Once, the developer needs to have understood the specification, before to start to write the tests and consequently the new functionality. Furthermore, the future updates will respect the specification, since the previous tests implemented will be run to check if the modifications changed the feature in a manner undesirable. Moreover, this technique prevents the developer to be influenced by the software development, increasing the probability of developers to implement all the necessary tests.

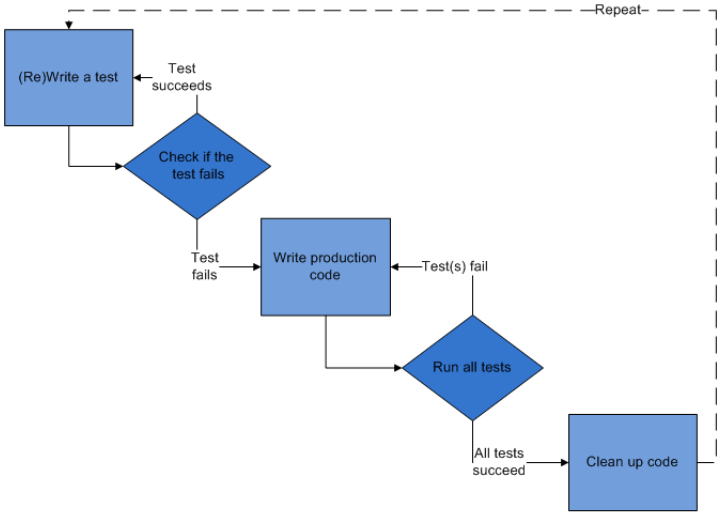

Figure 4.1: Test-driven development [14].

```
1   // File TraceClassVisitor.java
2   public class TraceClassVisitor extends ClassVisitor {
3      public TraceClassVisitor() {}
4
5      public MethodVisitor visitMethod(int access, String name, String desc,
6         String signature, String[] exceptions) {
7         System.out.println(``Method name: ''+ name);
8         MethodVisitor mv = super.visitMethod(access, name, desc, signature,
9         exceptions);
10           return mv;
11     }
12  }
13
14  // File BasicClassInformation.java
15  @Component
16  @Service //Declares that the class will offer an OSGi service
17  public class BasicClassInformation implements ClassInformation{
18      public void informationAboutTheClass(ClassReader classReader){
19       TraceClassVisitor classVisitor = new TraceClassVisitor();
20       classReader.accept((ClassVisitor) classVisitor,
21       ClassReader.SKIP_FRAMES);
22      }
23  }
24
25  // File ClassInformation.java
26  public interface ClassInformation {
27    public void informationAboutTheClass(ClassReader classReader);
28  }
```

Figure 4.2: Using ASM with the OSGi Declarative Services

4.4 Used Tools

ASM: The ASM Java bytecode framework designed by ObjectWeb consortium is a library that offers API to manipulate and analyze Java bytecode. ASM API is based on the Visitor design pattern [22] and runs quickly, adding little overhead when compared to other frameworks that perform transformations on Java bytecode. Figure 4.2, shows an example of the use of ASM API utilization, print the name of all methods in the class. A similar approach was used during the stage to obtain the necessary information such as class dependencies and method dependencies in the Java bytecode.

Eclipse: Eclipse IDE is a multi-language free open source software development environment and an extensible plug-in system [12]. Eclipse started as a Java IDE and is written mostly in Java. In the meantime, "the Eclipse Platform is built on a mechanism for discovering, integrating, and running modules called plug-ins, which are in turn represented as bundles based on the OSGi specification Platform" as cited in [5], see Figure 4.3, the plug-ins opened the possibility to employ in the IDE other programming languages

such as C, C++, Python, etc.

This IDE allows the easy integration with many other services, how to assist the team development using, for example, the Subversion software and easy module integration using the Maven software that will be described below. During this stage, the Eclipse Galileo 3.5 version was used.

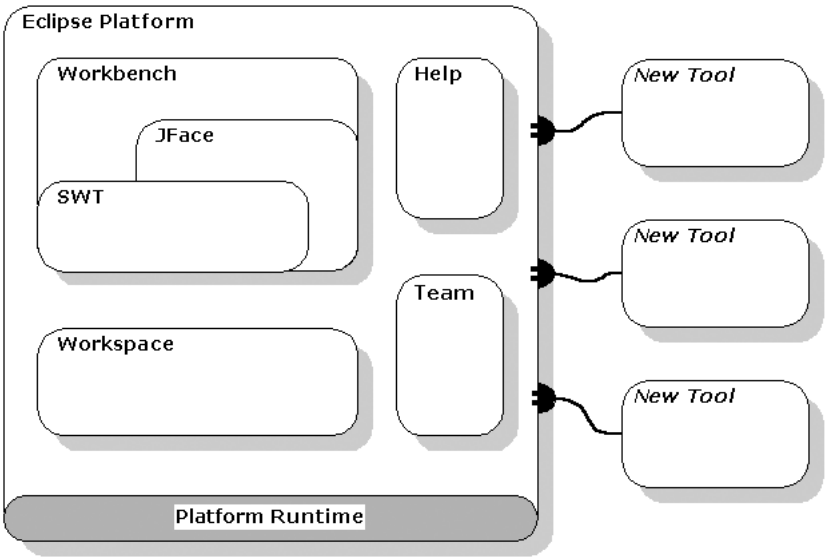

Figure 4.3: Eclipse OSGi Platform [12]

Maven: Maven is a software tool for project management and to build automation [1]. Maven started as a project management for Java programming, nowadays the software can also be applied to build and administrate projects written in other languages such as C#, Ruby. To build the project Maven uses the Project Object Model (POM) responsible to describe how the project needs to be built together with its dependencies.

Subversion: Subversion is a free/open software version and revision control tool [11]. It means that Subversion allows to the manage changes made on the files and directories, over time. The users can recover older versions of their data and obtain the history of how the data changed. For example, a developer can compare a previous stable version to an unstable current version to find out which bugs are in this version.

Another Subversion feature is the possibility of collaborating with people in the construction of the documents. Any developer in a project can commit its changes in the software code. When the other developers execute the code update, they will receive all the changes in the project. If there is any type of conflict, the software will detect during the commit or update phase warning the developer, indicating the location of the conflicts are and stopping the operation. The task only will be completed after the developers solve

all the conflicts in the code.

JUnit: JUnit is a framework for testing source code. The tool helps in the test-driven development [4]. The framework allows that test cases created by the developers will be executed without the necessity to do any configuration and verifies automatically the test in all compilations. In the JUnit 4.x is only necessary to use the JUnit library and the annotation *@Test*, to set any method as a test method. In the internship, the JUnit version 3.x and 4.x were used.

Chapter 5

Vulnerability studies

5.1 Introduction

The first step to construct a validation tool is to study the attacks already discovered in previous researches. This work, uses the [24] attack catalog as a reference for the vulnerabilities studied. The work demonstrated the existence of 32 vulnerabilities on OSGi. They are cited according to the Lindqvist authoritative taxonomy of intrusion results, where the main types are: DoS, Erroneous Output and Exposure (or Trojan Horse).

Denial of Service is considered the consumption of the platform resource, such as the processor time, RAM, disk space, which lead to platform unresponsiveness. This type of vulnerability is the most common in the Java SE, OSGi platforms and all vulnerabilities studied in this work are classified as a DoS: Big Component Installer, Excessive Size of Manifest, Decompression Bomb, Duplicate Package Import, Cycle Between Services and Big File Creator.

The Trojan Horse, in this work, will be considered any application that offers a service for the user, but also has the possibility to open a backdoor to be used by an invader to obtain access to the system control and to sensitive information.

Finally, Erroneous Output will be considered as malicious bundle with the unique aim to modified any bundle correct output into an erroneous output, in other words, the answer will be modify to a wrong value.

5.2 Big Component Installer

The Big Component Installer flaw consists in installing a considerably large component in the platform. The principal effect of this malicious operation is to occupy a considerable space in the RAM and in the hard drive memory causing a DoS with the performance breakdown consequences.

To detect a bundle with this intention, the platform analyzes if the bundle has a size bigger that a specify value (this value depends on the equipment characteristics and can

be changed by modifying the configuration file).

5.3 Excessive Size of Manifest

This vulnerability concerns that a bundle with a very large number of (similar package) import (more than 1 MB) as defined by [24]. It exploits the Felix and Knopflerfish implementations that work on a unique thread. Additionally, the launcher process takes several minutes to install the bundle, because the necessity to parse a manifest file with a big size.

The main problem is the unavailability of the platform during the malicious component installation. The cited implementations of OSGi Platforms need to execute the bundle start process in the same thread responsible for all the platform components. For this, the attack is classified as a Denial of service with the unavailability consequence.

To detect this attack, the validation platform consult the size of the manifest in the bundle. If it has more than a specific limit, more than 1 MB, it will be considered a malicious bundle.

5.4 Decompression Bomb

This attack is a variation of the classical decompression bomb attack used in other platforms. The offensive consists to realize a compression in the JAR file of a file with a large amount of repetitive or correlated data. For example, a BMP image (with 24 bits per pixel) with 1920 x 1080 pixels dimension has a decompressed size of 6075.05 KB around 1000 times larger than the compressed size of 6.11 KB.

The effect of the malicious service is to occupy a consider space in the RAM and use a lot of CPU during the decompression process. After the decompression, the bundle can consume an important hard drive memory, with the possibility to generate the same effect, causing a DoS with the performance breakdown effect.

To detect this exploit, any bundle that maintains file(s) with a considered size made of identical bytes will be labeled as malicious. These bundles usually have a significantly higher rate of decompression. For this reason, the platform will assume that all bundles are 15 times larger than the compressed file (estimation obtained after analyzing all the native libraries from Java SE 6 and some bundles in the OSGi Platform) will be classified as decompression bombs.

The first approach was to uncompress the bundle and compare the uncompressed rate with the rate defined as the decompression limit. If it is larger than the limit rate, it will be rejected by the validation test. However, this methods presents two problems with. First, it may cause important DoS in the server machine responsible for testing the bundle. Second problem is the possibility of introduce compressed files in the JAR files. A malicious library can take advantage of this characteristic, avoiding to be detected. If the validation test checks only the decompression rate in the bundle without checking the file in the bundle,

a malicious library will not be detected.

To avoid the first problem, two methods are applied to analyze the bundle file. In the first is only read the headers in the compact files to estimate the file size. With this estimation, the validation test can detect the most common types of decompression bomb without decompressing the files. Then, the method performs a real decompression in the bundle to obtain the real size in a buffer. This procedure allows to detects decompression bombs that modify the size in the file header. This test is safe, because it will use a buffer with a limited size, preventing the uncompress process to occupy all the RAM or hard disk in the server (the CPU is not a problem, because the machine will only execute the validation tests).

The second problem can be solved only by recursively decompressing all the libraries in the bundle file to obtain the total bundle uncompressed rate. The only weakness of this technique is to detect decompression bomb compressed with a new compression method. In this case, the platform must be updated to support the new compression method.

5.5 Duplicate Package Import

The Duplicate Package Import consists in adding a package import twice (or more times). As a consequence, the bundle with the Duplicate Package Import cannot be installed, blocking the execution of the services. Because of this, the attack is classified as DoS, with the unavailability consequence. To detect this violation the validation platform parses the manifest file to obtain all the fields in the "Import-Package". After the verification, the platform rejects the bundle in the case the packages are declared more than once. The platform could easily correct the Manifest File , the process is only to exclude the duplicate imports in the file. However, this task is out of the scope of the validation platform.

5.6 Cycle Between Services

This attack is based on two or more services that belong to different bundles, and consists in offering a fake service B to a legitimate service A. When the service A requests the service B execution, B will request the service A execution. The fake service calls service A, forming an infinite cycle of execution between services A and B.

After the malicious bundle installation, when a service calls B, the services will be involved in the infinite cycle call. Eventually, the calls will generate a stack overflow, triggering a DoS, which results in unavailability. In the meantime, the effects created by this attack will depend of the dependency injection used by the provider of the bundle: if it was constructed using the default method offered by the OSGi platforms, the bundle that started the Cycle Between the Service would change its state in the platform from ACTIVE to RESOLVED. This situation causes different damages depending on the bundle that started the cycle invocation: if the fake service started the cycle only the dependent

services will be affected. However, if a non malicious bundle started the cycle all of services provided by the bundle will be unavailable. Moreover, all the services that performed an invocation of the affected bundle will be affected. Furthermore, the malicious bundle will continue to offer the fake service, giving the possibility to others bundles to suffer the same problem and cause the unavailability in all the services depends of the affected operations on the platform.

Another situation concerns the legitimate bundles are implemented with the Declarative Services strategy, [16]. In this situation, a stack overflow will be triggered by the platform. This will have a similar an effect similar to the caused by the first possibility in the default strategy. Only the services that require use B will be affected, but in this case, all bundles will maintain their current state, ensuring the execution of services that do not have any dependence referring to the service B.

Before explaining about the detection method, some comments must be made to explaining the method detection: it is impossible to assert whether a bundle is malicious or not before runtime, because certain types of cycles will only be revealed at runtime. However, it is possible to assert whether a bundle has the possibility to execute a malicious cycle between the services before the runtime. Therefore, the validation platform can distinguish the safety bundles of the possible malicious bundles. All bundles detected as potentially dangerous can be classified as: bundle that does not comply with the recommended patterns (for example, the bundle that uses a circular reference between the imports to run in the platform) or a malicious bundle.

In order to detect this flaw, the validation platform checks in a first time the cycle between the imports and after the cycle between the services. In the first phase, the manifest file and the classes (applying the ASM API) from all bundles (installed on the platform and those are being tested) will be analyzed to obtain all imports and exports done by the bundles. Based on the information gathered, a directed graph is generated. The graph nodes represent the bundles and the edges represent a link between an export and an import. The algorithm to obtain all cycles in the graph is applied and the detected cycles are compared with the native cycles in the platform. After this comparison, if there are new cycle(s) in the platform the algorithm will start the second phase and only analyzes the bundles that belong to the new cycle(s). Else, the analyze returns that all bundles tested do not exploit the vulnerability.

In the second phase, the possible bundles belonging to a possible cycle between services have their Java bytecode analyzed in-depth checked in two sub phases, through the use of ASM API: in the first stage, only basic information will be collected such as: class name, what methods a specific class have and the bundle that is associated to the class. Subsequently, all classes will be analyzed in a second time to obtain the relationship between the class (implemented interfaces, subclasses, super classes) and the methods, which methods are necessary to execute a given method.

Based on the information collected, a graph will be produced, where is possible to check all the potential traces of all the method invokes. The algorithm to obtain cycles in graphs is applied again to obtain the potential service cycles.

Next, the platform executes a method to remove the potential cycles limited to a unique

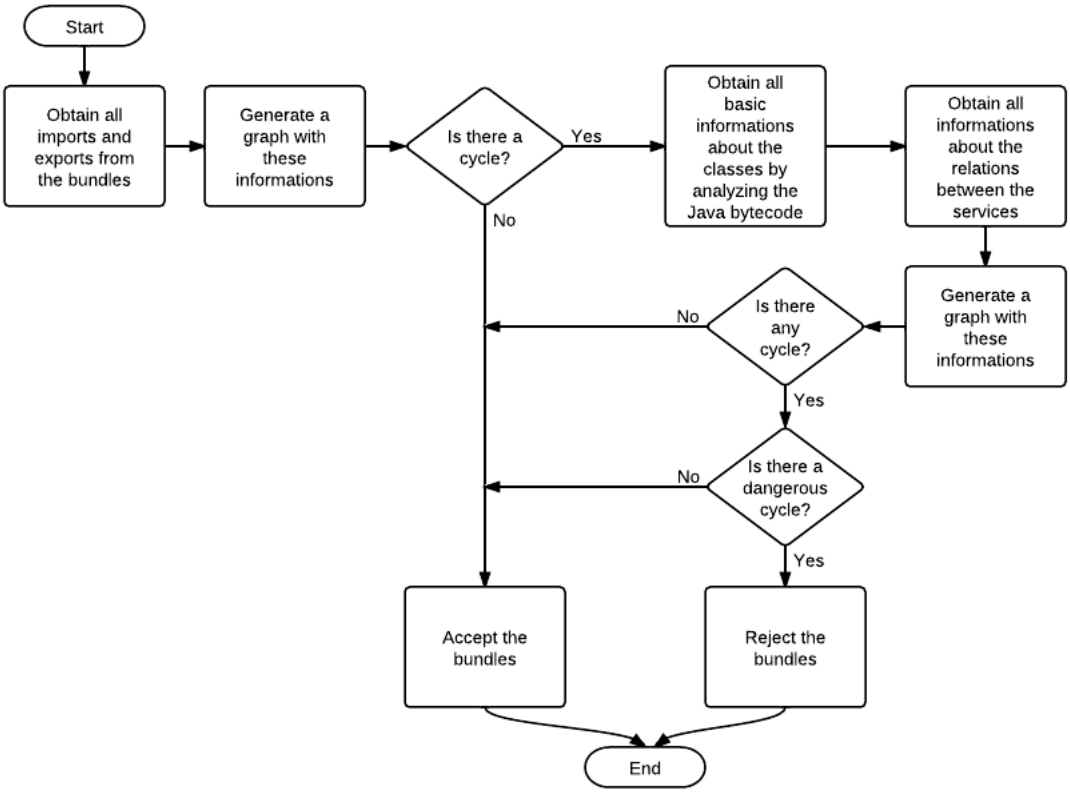

Figure 5.1: Cycle Between Services - Flowchart

bundle (even if this cycle has a cycle between services, this bundle is capable of causing its own unavailability). Finally, checks if there are any malicious cycles in the set of the tested bundles. If it is not contain any cycle the bundles are consider as a safe for this vulnerability, else the bundles are consider as a potentially dangerous and will be rejected by the validation platform.

5.7 Big File Creator

This vulnerability consists in malicious bundles that create a big file (relative to the available resources) to consume disk memory space [24]. The damage caused by the bundle blocks the installation of new bundles in the platform. Then, any application that needs to use the disk memory will not work. The effect causes a DoS with the performance breakdown consequence.

The two detection methods are limited. In the first approach, analyze the Java bytecode to find write commands using the FileOutputSteam class from Java SE and discover the size of all the entries. The problems here is that the bundle creator can hide the entries size, creating the file size in an other method and realizing many operations to difficult the analyze. The second problem is that the size can be based on a variable value, avoiding

the possibility to detect this attack in a static analyze.

The second method simulates the bundle execution. However, this procedure cannot detect the case where there is a “if” command that will only be activated in the case of a specific event, as a determinate date, or the situation where the size can be changed. Therefore, the flaw is better prevented using the Java Permissions, a specific disk space per user or/and bundle or to limit the access to the FileSystem.

Chapter 6

Work plan

6.1 Introduction

In this section, we present the available resources provided during the internship and we illustrate the internship project schedule.

6.2 Available resources

Orange Labs provided the following resources during the internship:
- One computer with Linux and Windows XP OS, using a user or root login, Internet access;
- One terminal machine with Windows 2003 OS, Internet access and the user mode;
- All cited open source software applications;
- Access to the digital libraries as the ACM library for example;
- Access to SVN server;
- Access to the intranet information related to the project and to the internship;
- The possibility to rent books from the France Télécom library.

6.3 Gantt Chart

The Gantt Chart shown in Figure 6.1 represents the time distribution for the tasks done until the moment of this report publication and it hides all the intermediate phases to conclude the tasks as a module development to support a test.

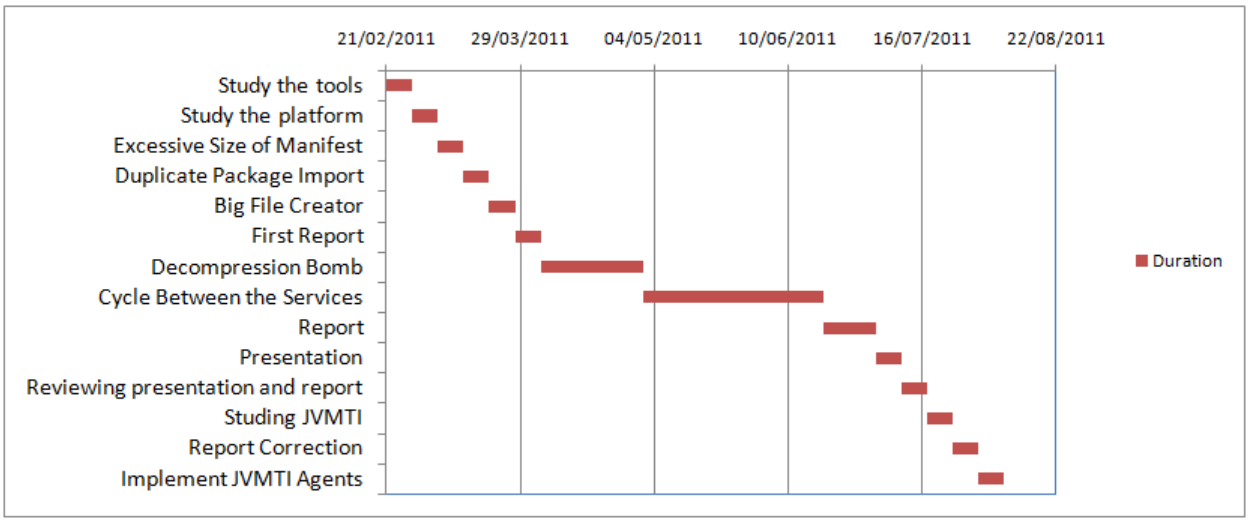

Figure 6.1: Gantt Chart

Chapter 7

Conclusion

The OSGi platform offers a multi-application environment, isolation between the modules, dynamic extensions and transparent installation of components. These are all features desired by Orange in order, to open its platform for software developed by third parties.

Furthermore, the OSGi platform provides many security features such as namespace isolation between the bundles, and inherits all the Java Platform security mechanisms like memory management, avoiding any buffer overflow attacks, and ensuring important security properties. In spite of these characteristics, the platform possesses security flaws that cannot be ignored, such as for example, the cycle between the services attack. These vulnerabilities require the modification and / or the integration of new security tools with OSGi Platform to increase the security or solve the known vulnerabilities.

In this sense, this work presents a way to improve the security of the platform, using automatic checkers. The verification platform consists in static code analysis to detect possible flaws statically. We have also discussed the reasons which, in some cases, limit the detection of all possible exploits statically.

Even when the verification tool cannot detect all the possible exploit in the OSGi platform, the validation platform is an important and complementary security tool. The validation platform can avoid the runtime techniques that employ methods with high overhead. For example, with the validation platform is not necessary to store and test all the services stack traces to detect the Cycle Between Service vulnerability at runtime.

The internship has provided an excellent opportunity to learn more about the new technologies, products in development and the security needs for these products, especially in the telecommunications and computer science domain where you need to satisfy the customer necessities without compromising their security. Moreover, the internship allowed for the first contact with the OSGi platform already used in software renowned such as the Eclipse Platform and most of J2EE application servers. The internship encouraged also to apply the acknowledged software development techniques such as TDD.

Bibliography

[1] Apache maven project. Available at: http://maven.apache.org. Accessed February 28, 2011.

[2] Orange-innovation.tv. Available at: http://www.orange-innovation.tv/fr. Accessed March 28, 2011.

[3] Osgi - wikipedia, the free encyclopedia. Available at: http://en.wikipedia.org/wiki/OSGi. Accessed Febraury 22, 2010.

[4] Welcome to junit.org! — junit.org. Available at: http://www.junit.org/. Accessed March 1, 2011.

[5] Eclipse platform technical overview. Rapport technique, Object Technology International, February 2003. Available at: http://www.eclipse.org/whitepapers/eclipse-overview.pdf. Accessed March 29, 2011.

[6] Java security overview, Apr 2005. Available at: http://java.sun.com/developer/technicalArticles/Security/whitepaper/JS_White_Paper.pdf. Accessed March 30, 2011.

[7] Java SE technical documentation, 2010. Available at: http://download.oracle.com/javase/. Accessed March 30, 2011.

[8] Orange development strategy in Africa, 2010. Available at: http://euroafrica-ict.org/wp-content/plugins/alcyonis-event-agenda//files/Orange_development_strategy_in_Africa.pdf. Accessed March 29, 2011.

[9] *OSGi Service Platform Enterprise Specification*. aQute Publishing, 2010. Available at: http://www.osgi.org/Download/Release4V42. Accessed February 25, 2011.

[10] Apache felix - index, 2011. Available at: http://felix.apache.org/. Accessed February 21, 2011.

[11] Apache subversion, 2011. Available at: http://subversion.apache.org/. Accessed March 29, 2011.

[12] Eclipse - the eclipse foundation open source community website, 2011. Available at: http://www.eclipse.org/. Accessed March 30, 2011.

[13] Osgi alliance — main / osgi alliance, 2011. Available at: http://www.osgi.org. Accessed February 21, 2011.

[14] Test-driven development, 2011. Available at: http://en.wikipedia.org/wiki/Test-driven_development. Accessed June 13, 2011.

[15] welcome to orange.com, 2011. Available at: http://www.orange.com/. Accessed March 28, 2011.

[16] O. Alliance : *Osgi Service Platform Enterprise Specification*. Aqute Publishing, 2010. Available at: http://www.osgi.org/download/r4v42/r4.enterprise.pdf. Accessed March 7, 2011.

[17] Scott Ambler : Introduction to test driven design. Available at: http://www.agiledata.org/essays/tdd.html. Accessed July 15, 2010.

[18] Olivier Corredo : France Telecom R&D devient Orange Labs, Jan 2007. Available at: http://www.lemondeinformatique.fr/actualites/lire-france-telecom-retd-devient-orange-labs-21870.html. Accessed March 30, 2011.

[19] Nicolas Geoffray, Gaël Thomas, Gilles Muller, Pierre Parrend, Stéphane Frénot et Bertil Folliot : I-JVM: a Java Virtual Machine for Component Isolation in OSGi. Research Report RR-6801, INRIA, 2009.

[20] R. Hall, K. Pauls, S. McCulloch et D. Savage : *Osgi in Action: Creating Modular Applications in Java*. Manning Pubs Co Series. Manning Publications, 2010.

[21] Chi-Chih Huang, Pang-Chieh Wang et Ting-Wei Hou : Advanced osgi security layer. *In Proceedings of the 21st International Conference on Advanced Information Networking and Applications Workshops - Volume 02*, AINAW '07, pages 518–523, Washington, DC, USA, 2007. IEEE Computer Society.

[22] Eugene Kuleshov : Using the asm framework to implement common java bytecode transformation patterns. Rapport technique. Available at:http://asm.ow2.org/current/asm-transformations.pdf.

[23] P. Parrend et S. Frenot : Security benchmarks of osgi platforms: toward hardened osgi. *Softw. Pract. Exper.*, 39:471–499, April 2009.

[24] Pierre Parrend et Stéphane Frénot : Java Components Vulnerabilities - An Experimental Classification Targeted at the OSGi Platform. Rapport de recherche RR-6231, INRIA, 2007.

[25] Dan S. Wallach et Edward W. Felten : Understanding java stack inspection. *In In Proceedings of the 1998 IEEE Symposium on Security and Privacy*, pages 52–63, 1998. Available at: http://sip.cs.princeton.edu/pub/oakland98.pdf.