HAL
archives-ouvertes.fr

Component-based Access Control: Secure Software Composition through Static Analysis

Pierre Parrend, Stéphane Frénot

▶ To cite this version:

Pierre Parrend, Stéphane Frénot. Component-based Access Control: Secure Software Composition through Static Analysis. Software Composition, Mar 2008, Budapest, Hungary. 4954/2008, pp.68-83, 2008, <10.1007/978-3-540-78789-1_5>. <inria-00270942>

HAL Id: inria-00270942
https://hal.inria.fr/inria-00270942

Submitted on 7 Apr 2008

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Component-based Access Control: Secure Software Composition through Static Analysis

Pierre Parrend, Stéphane Frénot
tel. +334 72 43 71 29 - fax. +334 72 43 62 27
{pierre.parrend, stephane.frenot}@insa-lyon.fr

INRIA ARES / CITI, INSA-Lyon, F-69621, France

Abstract. Extensible Component Platforms support the discovery, installation, starting, uninstallation of components at runtime. Since they are often targeted at mobile resource-constraint devices, they have both strong performance and security requirements. The current security model for Java systems, Permissions, are based on call stack analysis. They proves to be very time-consuming, which makes them difficult to use in production environments.
We therefore define the Component-Based Access Control (CBAC) Security Model, which aims at emulating Java Permissions through static analysis at the installation phase of the components. CBAC is based on a fully declarative approach, that makes it possible to tag arbitrary methods as sensitive. A formal model is defined to guarantee that a given component have sufficient access rights, and that dependencies between components are taken into account. A first implementation of the model is provided for the OSGi Platform, using the ASM library for code analysis. Performance tests show that the cost of CBAC at install time is negligible, because it is executed together with digital signature which is much more costly. Moreover, contrary to Java Permissions, the CBAC security model does not imply any runtime overhead.

Introduction

Extensible Component Platforms enable the composition of components that are provided by several issuers, and that can be loaded, installed, uninstalled, at runtime. In the Java world, such platforms can be Java Cards[17], MIDP [12], or the OSGi platform [14]. The functional composition is supported for instance, in the case of the OSGi Platform, through an efficient support of dependency resolution. Composition of non-functional properties, such as security, as so far not supported in such systems.

So as to support the composition of access rights of components, we propose the Component-based (CBAC) Security Model. The objective is to replace Java Permissions through validation of the access rights through static code analysis:

[0] This work is partially funded by MUSE II IST FP6 Project n026442.

Permissions prove to be difficult to use in production environment due to the important overhead they imply at runtime[1], To do so, we take advantage of the installation phase of the bundles to perform suitable checks, so as to relieve the runtime phase from costly checks. The advantages of this approach based on static-analysis are numerous. The main ones are the gain of performance at runtime, the absence of program abortion, and the flexibility of policy expression. This latter is fully declarative, which makes it possible to protect additional JVM methods (Threads, ClassLoader) and component methods.

This paper is organized as follows. Section 1 presents the existing security models for Java Component Platforms. Section 2 presents the CBAC Security Model. Section 3 presents the validation of our approach. Section 4 concludes this work.

1 Security Models for Java Component Platforms

Because it has been designed with security in mind, and because it has been subject to extensible testing by the community after its initial releases, the Java Virtual Machine [13] is usually considered as a very secure execution platform.

Various security mechanisms have been proposed to build secure Java Platforms. In this section, we review them so as to identify whether they provide suitable solutions for Component Platforms. First, the J2SE security mechanisms are presented. Then, optimizations for access control through static analysis, which allows to relieve the component runtime from verification overhead, are discussed. Lastly, security mechanisms for Component Systems are presented.

1.1 Java Security

The security properties of the Java Component Platform are supported by the Virtual Machine itself, which provides a platform that fully isolates - if required - the applications from their environment. These mechanisms build a sound basis to support a specific Access Control mechanism based on method calls stack inspection, *Stack-based Access Control* (SBAC).

The Java Virtual Machine is characterized by following built-in security features, that makes it safe: type safety, automatic memory management, Bytecode validation and secure class loading.

Type Safety ensures that programs can not generate type mismatching, and can not execute elements that are not functions [21]. It aims at preventing the use of pointers to execute arbitrary code and to prevent buffer overflows.

Automatic memory management is performed through Garbage Collection. It prevents memory leak, and relieves the developer from memory management, which is error prone. Automatic memory management ensures that no old functions stay available for undue future use.

[1] Our tests shows that the performance loss amounts between 55% and 144% for specific calls

Bytecode validation consists in checking the compliance to the JVM specification of the executable code before it is loaded for execution. This prevents in particular the delivery of malicious code in an untrusted component, that could not be generated by a compiler but would be forged to abuse the platform.

Secure class-loading provides a sound namespace isolation between different ClassLoaders. The namespace isolation ensures that no conflict occurs when independent components provides classes with the same name - and that no access is possible to objects and classes that are not purposely made publicly available. Some platforms have been developed, that allow to re-establish and control the access between ClassLoaders: the OSGi is an example [14]. Secure class-loading therefore allows the concurrent execution of applications that should not interact with each others.

SBAC: the Java Security Architecture Thanks to these language-level security properties, access control models can be defined in Java-based System.

The initial paradigm is Sandboxing, which prevents an untrusted application (such as an Applet) to access the host system [9]. This restrictive view of security has been extended to allow secure execution of partially trusted code [11]: an application could need to access the file system, but may not be trusted to perform network communications, to avoid backdoors. In this case, granted permissions would allow 'FilePermission', but no 'SocketPermission', for instance.

To support such a security model, Stack-based Access Control (SBAC) is used. Whenever a sensitive call is reached, the whole call stack is re-built, and the different Protection Domains of the code are identified. A Protection Domain matches a code signer (*i.e.* code provider), or a specific code location on the local file system. For each Protection Domain, access is granted to a certain number of actions. If all Protection Domains are associated with sufficient grants to execute the whole call stack, the sensitive call is executed. Otherwise, the call is aborted and an Exception is launched.

SBAC provides a fine-grained security models, that is tightly integrated in the JVM, and in the application. However, it has also several limitations. First, the permissions are defined programmatically, and can not be extended for additional methods. Secondly, the program aborts if no sufficient permissions are available, which can cause significant user trouble. Thirdly, the runtime performance overhead is important, and often causes the security features to be neglected in production systems. Significant optimization efforts have been done [10], but performance overhead in unavoidable when the whole call stack is to be rebuilt at each check.

1.2 Static Analysis for Optimization of Runtime Checks

The performance overhead at runtime that is implied by the SBAC paradigm urge the optimization of the verification process. Two main solutions exist: optimization of runtime analysis, and identification and removal of redundant checks.

The first approach to SBAC optimization is to perform optimization of runtime analysis, through static analysis before the execution. This approach avoids rebuilding the call stack at each security check.

The process is the following: an abstract model of the code is built out of the program code, and exploited at the check point, instead of the actual call stack. One method used to perform validation and to make this abstract model throughout the code is Security Passing Style (SPS) [19, 20]. SPS consists in rewriting method calls to add an additional parameter, which is a pointer to the abstract model. This method can be used to enforce arbitrary security model. Its application to SBAC is presented by [19, 20].

Optimization can be done through two different strategies: eager [4, 5] and lazy [8] computation of the program state. The eager approach avoids numerous re-computation, when the number of security checks is high. The lazy approach provides usually better performances, since security checks are usually scarce in the code. Both strategies are proved to be formally equivalent [3].

The main limitation of static analysis is that the abstract model of the program builds a slightly bigger set than the actual call stack. This causes a (limited) number of false positive.

The second approach to SBAC optimization is the identification and removal of redundant checks. Actually, when a given piece of code has the right to access a given file, this right will not be withdrawn if the same code - with the same call stack - have a renewed access to the file.

Removal of redundant checks and of dead code is defined in an abstract way by [6]. Algorithmic example of such removal is given by [3]. The implementation of these models can be done with call graphs [8].

Removal of redundant checks is performed as a complement to optimization of call stack computation, and uses the same techniques of code rewriting. It implies an additional overhead during code analysis, but provides only performance gain at runtime.

Such optimization does not solve all drawbacks of the SBAC model. The runtime overhead is still non negligible, and the conservation of runtime checks preserves program abortion, which is not compatible with a satisfactory user experience.

1.3 Securing Java Components

The strong security features that are provided by the JVM, and the sound modular support, enables Component Platforms to be defined, so as to enable the realization of complex applications through software composition. Securing these component platforms is done in two steps. First, the deployment must be secured. For more information related to this aspect, see [15] and [16]. It will not be considered further here. Secondly, access control must be adapted to support the dynamic life-cycle of the components.

Dynamic Permissions In single application Java systems, permissions to access sensitive methods are set statically in the `java.policy` file. In extensible component platforms, where components are installed, started, stopped and uninstalled at runtime, these permissions must be set at runtime. Dynamic permissions are therefore defined: for each issuer, a specific permission set is given.

When components from new issuers are installed, a suitable permission set is introduced. Dynamic permissions are defined both in MIDP [1], and in OSGi [14]. However, few implementations seem to be actually available, which let think that the specified solutions do not match widespread use cases with a satisfactory usability level. The main restriction is the following: dynamic permissions are often set by the user when required. This causes an uncomfortable alternative: either the program crashes in the middle of a computation, or the user is disturbed every now and then to add new permissions which consequences she does not understand. Moreover, software composition and the interdependence between components is not taken into account. Dynamic permissions also have the same drawbacks as standard permission: runtime performance overhead, and the impossibility of defining new permissions type, which could be necessary in highly dynamic environments.

Due to a lack of maturity, the current dynamic permissions for software component platforms do not seem to be well suited for their target applications.

Taking Composition into Account for building secure Systems The specific programming paradigm that is implied by software composition provides new anchors to build secure software: the component life-cycle management support, and the interaction between the components. However, none of them are so far exploited in security specifications, which are directly derived from monolithic security mechanisms.

The life-cycle support provides a suitable check point for security control, that enable disturbance free execution: the install phase. Currently, only digital signature validation is performed at this moment. More control, such as static analysis, could be introduced, to relieve the runtime from performance overhead and impaired user experience that is caused by standard execution permissions.

The interaction between the components can be exploited at this install time check point. Actually, dependency resolution is performed during this phase. Extending the performed computation with security checks would prevent a later analysis of the program state and call stack, and make it possible to validate the security properties of the applications before they are launched. Consequently, runtime checks would be greatly limited, which would prevent unpleasant program abortion - or dangerous user intervention.

2 The CBAC Security Model

The CBAC - Component-based Access Control - Security Model aims at taking advantages of the specific properties of software composition, to propose a security model that is both more performant and more suitable to the use cases of Java Component Platforms.

2.1 Principles

The principle of the CBAC Security Model is to perform an install time analysis of the installed components, while taking the emerging compositional properties into account.

The access rights for a component that is to be installed are checked immediately before its installation. If the rights are sufficient, the component is installed. It they are not, the component is rejected. The access rights must be computed in two complementary fashions: first, the component must not perform forbidden actions itself; secondly, the component must not depend on components that perform actions that it is not allowed to perform. This interdiction must consider two specific cases: privileged components, that can perform sensitive actions on behalf of others with less rights (for instance a log service must access the file system, but the logged component need not to have itself the access right to the log file); services, which build runtime dependencies, and are used in Service-Oriented Programming [7] environments.

2.2 Hypotheses

The CBAC Security Model is valid when the two following hypotheses are valid:

- the component platform itself is not modified *i.e.* the process of access right verification can not be tampered with. This can be obtained through the use of a Trusted Computing Base [2].
- each component contains a valid digital signature, which guarantees that no modification has been done to the component archive, and that the component signer name is unique and known by the platform[2].

The CBAC Security Model is defined to fully support secure OSGi applications. Since it considers fundamental properties of software composition, it should be easily adaptable to other specifications of component platforms. However, it may be that specific options or features are not defined.

2.3 Modelization

The definition of CBAC is performed in three steps.

First, CBAC is defined for one component. When a single component is installed, does it own the rights to execute all the method calls it performs ? The dependencies to other bundles are not considered, and sensitive methods from the platform API only are considered.

Secondly, CBAC is defined for N components. When a component is installed that has dependencies to other, does it have sufficient rights to execute the method calls it performs both directly and indirectly via the dependencies ? Suitable grants are to be available for the whole call stack.

Thirdly, CBAC is defined for B components, taking into account the protection of component methods. The methods provided by the platform can be tagged as 'sensitive', as well as the ones provided by bundles. This means that if a given bundle has a method that triggers a malicious action, such as *e.g.* sending private data over the network, its access can be restricted.

[2] In particular, this implies that default Java Archive signature tools should not be used [16]

Static Permissions for one Component **Hypotheses** Following parameters are defined to describe security policy-related entities in a dynamic extensible component platform:

b **:** a given bundle,
pf **:** the Component Platform,
C **:** the set of calls made by a bundle,
S **:** the set of all sensitive method calls in the Platform API,
I **:** the set of all innocuous method calls in the Platform API,
$C_S = C \bigcap S$ **:** the set of all sensitive method calls made by a bundle,
$C_I = C \bigcap I$ **:** the set of all innocuous method calls made by a bundle,
$\{p\}$ **:** the set of bundle providers,
A_p **:** the set of authorized sensitive calls for the provider $p \in \{p\}$, *i.e.* the policy for the provider p.

Theorem CBAC permissions for one component ($CBAC\_1C$) are valid if:

$$p, pf, b \vdash C_S \wedge A_p, C_I \quad (1)$$
$$with\ PSC_{pf,b} = C_S$$

$PSC_{pf,b}$ is the set of Performed Sensitive Calls by the bundle b on the Platform pf. This means that the set of Performed Sensitive Calls by the bundle b on the Platform pf is the set of sensitive calls that are identified in the bundle. These calls are explicitely allowed by the security policy A_p. Otherwise, that is to say if some sensitive calls that b may perform is not allowed by the policy, the bundle is rejected. This mechanism supports robust coarse-grained access control policies.

Proof A bundle b is associated with the calls C it performs on the platform. A platform pf is associated with the set of sensitive calls S, and the set of innocuous calls i it makes available. C is a part of S and I. Thus, the platform pf and the bundle b are associated to C, and consequently to the set of sensitive calls it performs C_S, which is a subset of C.

The bundle provider p is associated with the policy A_p, which is defined at the platform level. If the permission is given, the performed sensitive calls C_S build a subset of the policy for p. Consequently, C_S and A_p build a coherent policy for the bundle b provided by p on the platform pf. b can thus be installed. If the permission is denied, C_S is not a subset of p, and C_S and A_p do not build a coherent policy. b can not be installed.

A demonstration with Sequent Calculus is given in appendix A.

Static Permissions for N Components - with Protection of Platform and Bundle Calls The computation of the component validation status for a new component must consider two types of bundles in the dependency tree:

- Leaves L: the components that have dependencies to platform-provided packages only (simple applications, or libraries),

– Nodes N: the components that have dependencies both to the Platform and to other components (complex applications, GUI ...).

The Figure 1 shows an example of a dependency tree for a default configuration of the Apache Felix platform (version 1.0), which is an implementation of the OSGi R4 Specifications.

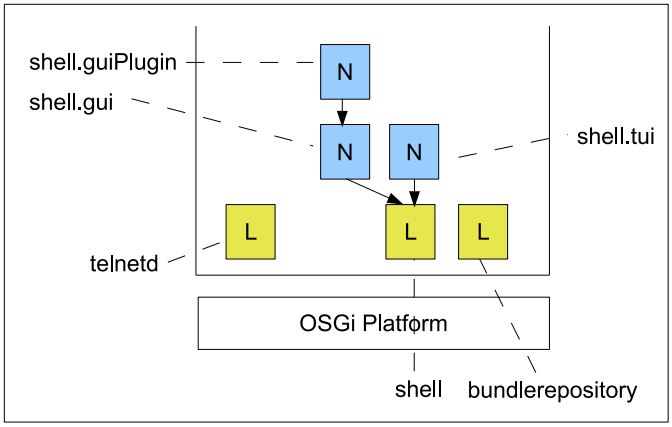

Fig. 1. The Dependency Tree for a default Configuration of the Apache Felix Platform

Hypotheses Following parameters are defined to describe security policy-related entities in a dynamic extensible component platform for N components:

b_i **:** the ID of the considered bundle,
$\{b\}_i$ **:** the list of bundles on which the bundle i depends,
$C_{S_{pf,b_i}}$ **:** the sensitive calls performed by the bundle i directly to the platform, or directly to the bundles on which it depends,
A_{p_i} **:** the set of authorized sensitive calls for the provider p of the bundle i,
PSC_{b_i} **:** the set of Performed Sensitive Calls by the bundle i, directly to the Platform or via method calls to other bundles,
PSC_{pf,b_i} **:** the set of Performed Sensitive Calls by the bundle i that are made directly to the Platform, or directly to the bundles on which it depends,
$PSC_{\{b\}_i}$ **:** the set of Performed Sensitive Calls by the bundles on which the bundle i depends. $PSC_{\{b\}_i} = \sum_{b_j \ in \ \{b\}_i} PSC_{b_j}$.

Theorem The computation of the validation status of a component can be made recursively.

• A leaf component $b_i \in L$ is valid if:

$$b_i, p_i, pf \vdash A_{p_i} \wedge C_{S_{pf,b_i}}, C_{I_{pf,b_i}} \quad with \ PSC_{pf,b_i} = C_{S_{pf,b_i}} \tag{2}$$

Which matches the case for one single component.

• A node component N is valid if:

$$b_i, p_i, pf \vdash A_{p_i} \wedge PSC_{\{b\}_i}, \neg PSC_{\{b\}_i} \tag{3}$$

$$with\ PSC_{b_i} = C_{S_{pf,b_i}} \vee PSC_{\{b\}_j} \tag{4}$$

PSC_{b_i} is the set of Performed Sensitive Calls by the bundle b_i b on the Platform pf or on the bundles on which it depends. This means that a bundle can be installed when the sensitive calls that are made directly to the platform and all sensitive calls that are made via other bundles are allowed by the current policy.

Proof This can be demonstrated with the following argument through recursion.

Suppose that the set of Performed Sensitive Calls for the bundle k, PSC_{b_k}, is available for all bundles k that are already installed on the gateway. The objective is to determine whether the bundle i, which has dependencies to a set of bundle b_j, can be installed without breaking the access control policy. The value of PSC_{b_i}, the set of Performed Sensitive Calls for the bundle i, can then be extracted.

A demonstration with Sequent Calculus is given in appendix B.

The implementation of the CBAC security model for N components requires two complementary mechanisms: checking at install time for the calls that are made directly from the bundle to be installed to the platform, and building of the dependency tree to verify that the required permissions are granted. The bundle b_i can only be installed if a valid dependency tree can be identified.

The mechanism presented so far provides a minimal enforcement process for execution permissions, that can yet be used to protect real applications. As for the CBAC model for one component, the benefits are the absence of runtime overhead, and the declarative - and thus extensible - policy declaration mechanism.

The limitations are the following. First, the validation is performed at a quite coarse-grain level. For instance, the installation of an application can be denied because it relies on a library that contains forbidden sensitive methods, even though these methods are never called. This can occurs for instance in the case of conditional instructions. Secondly, it is not possible to define sensitive method calls that would be provided by the components themselves.

The mechanism of protection of bundle calls shows both the flexibility and limitation of component-grained static access control. The advantage is that permissions can be set in a declarative manner, that is to say that contrary to default Java Permissions, any call can be considered as ‘sensible‘ - for instance for a particular application type, or in case a vulnerability is discovered. The limitation is that component-grained access control does not provide the policy designer with context-dependent behavior: if a given sensitive method is used only in rare cases or is never used, but forbidden by the policy, the bundle can not be installed.

Advanced Features To as to provide a proper protection of OSGi-based applications, several advanced features have to be defined.

The first feature is *Privileged Method Calls*. Privilege calls means that a given component can execute sensitive calls, even though the initial caller of the method does not own sufficient rights. This is for instance the case of logging mechanisms. An component from a less trusted provider can log its action on a file through the platform logger without having access rights to the file system.

The second advanced feature, which is provided by the CBAC model thanks to its declarative nature, is the support of *Positive and Negative Permissions*. Negative permissions are set by identifying a given method call as 'sensitive'. Positive Permissions are set by allowing a given signer to execute some methods. A more flexible expression could be introduced in the future, in particular so as to support negative permissions without impacting the policies for other providers.

The last required feature is to support *Access Control for Service Calls*. The OSGi Platform support Service Oriented Programming (SOP) [7], and thus calls through services that are published inside the platform. Since the services are resolved at runtime, a runtime mechanism is to be defined that enforces the CBAC policy for these service. Otherwise, package level access control can be by-passed through service calls.

3 Validation

The validation of the Component-Based Access Control model is performed through its implementation, performance analysis and identification of the advantages and drawbacks of the current prototype.

The principle of CBAC is the following. When an OSGi bundle is to be installed on an OSGi Platform, its digital signature is checked. If this latter is valid, the CBAC Engine verifies whether the bundle signers have sufficient rights to execute all the code that is contained in the archive. If enforced policies do allows it, the bundle is installed, and then executed without further constraints. Otherwise, it is rejected.

The innovation of the approach is to take advantage of the installation phase of the OSGi bundles to perform access right verification.

3.1 Implementation

The implementation of the CBAC security model is presented in this section. Development choices are justified, and the integration with the *Secure Felix*[3] platform is given. Expression of CBAC policies are then given, along with an example.

The implementation of the CBAC Model is performed on the Felix[4] implementation of the OSGi framework. Static Bytecode analysis is performed with the ASM[5] library, which is much smaller than other libraries for Bytecode ma-

[3] http://sfelix.gforge.inria.fr
[4] http://felix.apache.org/
[5] http://asm.objectweb.org/

nipulation, such as BCEL [6] or SERP[7]. An earlier prototype has also been built using the Findbugs framework, which often implies an overhead of more than 100% in performance.

The CBAC model is part of a research project conducted at the Amazones Team of the INRIA (CITI Laboratory, INSA-Lyon, France), that aims at defining a secure OSGi platform. It is therefore integrated with SFelix, which support a proper verification of the digital signature of OSGi bundles [16]. SFelix is modified so as to make the list of valid signers available for the CBAC Engine. This latter can then check whether the signers have suitable rights to execute all the methods that it contains.

An example of CBAC policy is given in the listing 1: a policy file cbac.policy, defines sensitive methods and required grants for a given signer. The syntax is similar to Java Permissions 'java.policy' files.

```
sensitiveMethods {
 java.io.ObjectInputStream.defaultReadObject;
 java.io.ObjectInputStream.writeInt;
 java.security.*;
 java.security.KeyStore.*;
 java.io.FileOutputStream.<init>;
};

sensitiveManifestAttributes {
  Fragment-Host;
}

grant Signer:bob {
  Fragment-Host;
  java.io.ObjectInputStream.defaultReadObject;
  java.io.ObjectInputStream.writeInt;
  java.io.FileOutputStream.<init>;
  java.security.Security.addProvider;
  java.security.NoSuchAlgorithmException.<init>;
  java.security.KeyStore.getInstance;
  ...
};
```

Table 1. Example of the cbac.policy File

CBAC policy is defined as follows. A list of sensitive methods, and one of sensitive OSGi-related meta-data are defined. If a bundle contains sensitive methods or meta-data, its provider must be granted the right to execute them all. Otherwise, it is rejected at install time. The policies are defined according to a

[6] http://jakarta.apache.org/bcel/
[7] http://serp.sourceforge.net/

declarative approach: it is possible to mark any method call as being sensitive. This makes it possible to protect the framework against vulnerabilities that are discovered after the release of the platform. On the contrary, Java Permissions mechanisms, that are coded in the framework itself, freeze the set of sensitive methods when the platforms (or any application) is released.

3.2 Performances

The main objective of the CBAC model is to relieve the OSGi platforms, which are often executed in resource-constraint environments, from the overhead that is implied by Java Permission at runtime. It is therefore important to control that the performances of the system are not too much impacted by this new mechanism.

Tests have been performed with the implementation of the CBAC security model for one component. Figure 2 shows the duration of the CBAC check only, and Figure 3 shows the performance of Digital Signature validation and CBAC Check, which are to be performed together to ensure the validity of the analysis. It highlights the fact that for a limited number of sensitive methods (which is usually the case), the overhead implied by CBAC is negligible when compared to the duration of digital signature check. Note that the abscissa is not linear, but represents the size of the various bundles that are available in the Felix distribution of the OSGi Platform.

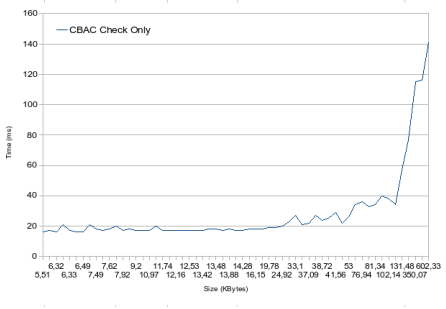

Fig. 2. Performances of OSGi Security: CBAC Check only

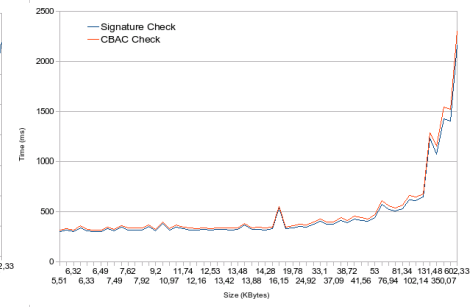

Fig. 3. Performances of OSGi Security: Signature and CBAC Check

The test are conducted for all the bundles that are provided together with the Apache Felix distribution, utility bundles and tests bundles. The performance overhead for small bundles (less that 25 KBytes) is less than 20 ms (between 3,2% and 6,7% of the total validation process). For bigger ones (up to 130 KBytes), less than 40 ms are required (less than 7,7% of the total validation process). Out of the 59 test bundles, only five have a longer verification time. The bigger bundles are 350 KBytes, 356 and 602 KBytes big, and require respectively 115,

116 and 141 ms. Since the digital signature for these bundles is checked in more that one second, the overhead implied by CBAC can be considered as negligible.

3.3 Advantages and Limitations

The innovation of the CBAC approach is to take advantage of the installation phase of the OSGi bundles to perform access right verification. Identified advantages and known drawbacks are now presented.

The CBAC Model, when used together with a Hardened Version of the OSGi Platform [15], support a better protection that other available security mechanism, such as Java Permission. This is shown in Figure 4. The 25% of unprotected vulnerabilities are presented in our Technical Report on OSGi vulnerability [15], and, interestingly enough, are due to the Java Virtual Machine, and not the OSGi Platform. This means that other Java Platforms also present these weaknesses.

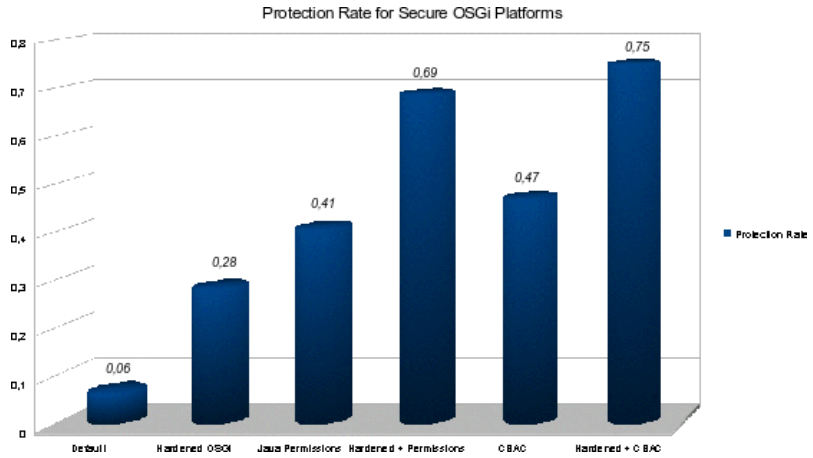

Fig. 4. Protection Rate of existing OSGi Security Mechanisms: CBAC and others

The identified advantages of CBAC are the following:

- no runtime overhead (contrary to Java Permissions),
- no application abortion due to unsufficient execution rights (contrary to Java Permissions),
- no unsecure behavior of the users, which are driven with classical permission into allowing the execution of all untrusted code [18],
- possibility of defining arbitrary sensitive methods and meta-data, which makes it possible to protect the system from vulnerabilities that are discovered after the platform release.

Moreover, the expression of policies is simpler that in the case of Java Permission, since it is declarative rather that programmative. Since the syntax is very similar, they can very easily be learned by Java developers.

The known drawback of the CBAC approach are the following:

- false positives are to be expected when compared to Java Permission. This may not be a problem if the configuration (*e.g.* the trusted signers, the applicative bundles) of the target platforms is known beforehand, as is currently the case in most OSGi-based systems: sufficient permissions can be set at design time. However, it can be a restriction for pervasive systems, which require an important interoperability of mutually unknown code.
- install time overhead (negligible, as shown in Figures 2 and 3),
- disk space consumption (333 KBytes, 319 KBytes of which are built by the - unmodified - ASM library). Our current prototype is much smaller than the original implementation as Findbugs Plugin (2,026 MBytes, mostly because of the Findbugs and BCEL libraries, that amounts to 2,003 MByte), and much more performant (between 40 and 60 % of benefits for the various archives). So as to be used in embedded systems, the unused code of the library is to be pruned.

The formalization of CBAC is given in this study, and a first validation is given based both on qualitative and quantitative criteria, which show that the CBAC model is a performant and useful approach to Access Control for Extensible Component Platforms, such as the OSGi Platform.

4 Conclusions and Perspectives

We propose the Component-Based Access (CBAC) Security Model for enforcing access control in Extensible Component Platforms. The principle is to take advantage of the installation phase of the components to perform static analysis, and to check that the composition of software components does not break the defined policies.

CBAC is in particular defined so as to overcome the known limitation of standards Java Permissions, which are often considered as too heavyweight to be used in production environments. The tests that we perform show that these objective is achieved, even though at the cost of additional false positives. The current limitations of our implementation is that the support of dependencies (CBAC for N components) is currently defined, but not yet implemented. Moreover, the validation in a real system is still to be performed, to as to check whether the defined policy language is flexible enough. So far, there is no reason to think that this is not the case.

Two directions of future works are to be considered. First, the CBAC security model is to be tested for other specifications of component platforms, for instance Java MIDP Profile, or non-Java environments. Our prototype has been tested over the OSGi Platform, but only limited OSGi-specific features have been introduced, and adaptation should not pose major issues.

Secondly, the prototype is to be validated in resource-constraint environments, so as to ensure that the process is lightweight enough to be used in PDA or set-top boxes.

References

1. *Porting Guide - Sun Javatrademark Wireless Client Software 2.0 - Java Platform, Micro Edition*. Sun Microsystem, May 2007.
2. W. Arbaugh, D. Farber, and J. Smith. A secure and reliable bootstrap architecture. In *IEEE Symposium on Security and Privacy*, pages 65–71, 1997.
3. A. Banerjee and D. Naumann. A simple semantics and static analysis for java security. Technical Report 2001-1, Stevens Institute of Technology, 2001.
4. M. Bartoletti, P. Degano, and G. Ferrari. Static analysis for eager stack inspection. In *Workshop on Formal Techniques for Java-like Programs (FTfJP'03)*, 2003.
5. Massimo Bartoletti. *Language-based security: access control and static analysis*. PhD thesis, Universita degli Studi di Pisa, 2005.
6. Massimo Bartoletti, Pierpaolo Degano, and Gian Luigi Ferrari. Stack inspection and secure program transformations. *International Journal of Information Security*, 2:187–217, 2004.
7. Guy Bieber and Jeff Carpenter. Introduction to service-oriented programming (rev 2.1). OpenWings Whitepaper, April 2001.
8. Byeong-Mo Chang. Static check analysis for java stack inspection. *ACM SIGPLAN Notices*, 41(3):40 – 48, March 2006.
9. D. Dean, Felten, and Wallach. Java security: From hotjava to netscape and beyond. In *SP '96: Proceedings of the 1996 IEEE Symposium on Security and Privacy*, page 190, Washington, DC, USA, 1996. IEEE Computer Society.
10. L. Gong and R. Schemers. Implementing protection domains in the java development kit 1.2. In *Network and Distributed System Security Symposium*, 1998.
11. Li Gong, Marianne Mueller, Hemma Prafullchandra, and Roland Schemers. Going beyond the sandbox: An overview of the new security architecture in the java development kit 1.2. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, 1997.
12. JSR 118 Expert Group. Midp 2.0. Sun Specification, November 2002.
13. Tim Lindholm and Frank Yellin. *The Java(TM) Virtual Machine Specification (2nd Edition)*. Prentice Hall PTR, 1999.
14. OSGI Alliance. Osgi service platform, core specification release 4. Draft, 07 2005.
15. Pierre Parrend and Stephane Frenot. Java components vulnerabilities - an experimental classification targeted at the osgi platform. Research Report RR-6231, INRIA, 06 2007.
16. Pierre Parrend and Stephane Frenot. Supporting the secure deployment of osgi bundles. In *First IEEE WoWMoM Workshop on Adaptive and DependAble Mission- and bUsiness-critical mobile Systems, Helsinki, Finland*, June 2007.
17. Sun Inc. Java card platform specification 2.2.2, March 2006.
18. Masaru Takesue. A scheme for protecting the information leakage via portable devices. In *International Conference on Emerging Security Information, Systems and Technologies, IARIA SecurWare*, 2007.
19. Dan S. Wallach. *A New Approach to Mobile Code Security*. PhD thesis, Department of Computer Science, Princeton University, 1999.
20. Dan S. Wallach, Andrew W. Appel, and Edward W. Felten. Safkasi: A security mechanism for language-based systems. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 9(4):341 – 378, October 2000.
21. Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

A Static Permissions for one Component - Demonstration

The demonstration of the Theorem for validating CBAC for one component can be prove through sequent Calculus:

$$CBAC\_1C =$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{b \vdash C \quad pf \vdash S, I}{pf, b \vdash C, S, I \quad S, I \vdash C}}{pf, b, S, I \vdash C, S, IC} \qquad C \vdash C_S, C_I}{pf, b \vdash C_S, C_I} \qquad p \vdash A_p}{p, pf, b \vdash C_S \wedge A_p, C_I}$$

$$with \ \dfrac{C \vdash C_S \wedge C_I}{C \vdash C_S} \quad and \ \dfrac{C \vdash C_S \wedge C_I}{C \vdash C_I}$$

Which describes a Component Platform pf , with a given bundle b to be installed, provider by a bundle provider p. Under this configuration, the conjunction of the called Sensitive methods C_S and the list of Authorized methods by the security policy A_p must be valid.

Which is tantamount to:

If Permission is given:

$$CBAC\_1C =$$

$$\dfrac{\dfrac{p, pf, b \vdash C_S \wedge A_p, C_I \quad \vdash A_p \wedge C_S, \neg C_S}{p, pf, b \vdash C_S \wedge A_p \wedge (A_p \wedge C_S), \neg C_S, C_I} \qquad C_S \vdash A_p \wedge C_S}{p, pf, b \vdash C_S \wedge A_p, C_I}$$

If Permission is refused:

$$CBAC\_1C =$$

$$\dfrac{\dfrac{p, pf, b \vdash C_S \wedge A_p, C_I \quad \vdash \neg A_p \wedge C_S, \neg C_S}{p, pf, b \vdash C_S \wedge A_p \wedge (\neg A_p \wedge C_S), \neg C_S, C_I} \qquad C_S \vdash \neg A_p \wedge C_S}{p, pf, b \vdash C_I}$$

Which means that a bundle can be installed if the validity of all the Sensitive Calls C_S is implied by the list of Authorized Calls A_p. Otherwise, only the non sensitive calls can be executed, which mean that the bundle can not be installed for normal execution.

B Static Permissions for N Component - Demonstration

The demonstration of the Theorem for validating CBAC for one component can be prove through sequent Calculus:

$$CBAC\_NC =$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{b_i \vdash \{b\}_j}}{b_i \vdash b_j \wedge b_k \wedge \ldots \wedge b_q \quad b_j \vdash PSC_{b_j} \quad b_i \vdash C_{S_{pf,b_i}}}}{b_i \vdash b_j \wedge b_k \wedge \ldots \wedge b_q, C_{S_{pf,b_i}}, b_j \Rightarrow PSC_{b_j}}}{b_i \vdash C_{S_{pf,b_i}}, PSC_{b_j} \wedge PSC_{b_k} \wedge \ldots \wedge PSC_{b_q} \qquad p_i, pf \vdash A_{p_i}}}{b_i, p_i, pf \vdash A_{p_i} \wedge C_{S_{pf,b_i}}, A_{p_i} \wedge PSC_{b_j} \wedge PSC_{b_k} \wedge \ldots \wedge PSC_{b_q}}}{b_i, p_i, pf \vdash A_{p_i} \wedge (C_{S_{pf,b_i}} \vee PSC_{\{b\}_j})}}{b_i, p_i, pf \vdash A_{p_i} \wedge PSC_{\{b\}_i}}$$

Which describes a Component Platform pf , with a given bundle b_i to be installed, provider by a bundle provider p_i. Under this configuration, the conjunction of the Sensitive methods called directly or indirectly by the bundle i, noted $PSC_{\{b\}_i}$ and the list of Authorized methods by the security policy A_{p_i} must be valid.

Which is tantamount to:

If Permission is given:

$$CBAC\_NC =$$

$$\dfrac{\dfrac{b_i, p_i, pf \vdash A_{p_i} \wedge PSC_{\{b\}_i} \qquad \dfrac{PSC_{\{b\}_i} \vdash A_p \wedge PSC_{\{b\}_i}}{\vdash A_p \wedge PSC_{\{b\}_i}, \neg PSC_{\{b\}_i}}}{b_i, p_i, pf \vdash A_{p_i} \wedge PSC_{\{b\}_i} \wedge A_p \wedge PSC_{\{b\}_i}, \neg PSC_{\{b\}_i}}}{b_i, p_i, pf \vdash A_{p_i} \wedge PSC_{\{b\}_i}, \neg PSC_{\{b\}_i}}$$

If Permission is refused:

$$CBAC\_NC =$$

$$\dfrac{\dfrac{b_i, p_i, pf \vdash A_{p_i} \wedge PSC_{\{b\}_i} \qquad \dfrac{}{PSC_{\{b\}_i} \vdash \neg A_p \wedge PSC_{\{b\}_i}} \qquad \vdash \neg A_p \wedge PSC_{\{b\}_i}, \neg PSC_{\{b\}_i}}{b_i, p_i, pf \vdash A_{p_i} \wedge PSC_{\{b\}_i} \wedge \neg A_p \wedge PSC_{\{b\}_i}, \neg PSC_{\{b\}_i}}}{b_i, p_i, pf \vdash \neg PSC_{\{b\}_i}}$$

Which means that a bundle can be installed if the validity of all the Sensitive Calls $PSC_{\{b\}_i}$ performed directly or indirectly bu the bundle i is implied by the list of Authorized Calls A_p. Otherwise, only the non sensitive calls can be executed, which mean that the bundle can not be installed for normal execution.