A Security Sandbox for Software Components

Ibrahim Bolarinwa

School of Computing Science

Sir Alwyn Williams Building

University of Glasgow

G12 8RZ

A dissertation presented in part fulfillment of the requirements of the Degree of Master of Science at the University of Glasgow

1st of September, 2015

Abstract

Component Based Systems consist of multiple autonomous “software chunks” called components that interact with each other. The aim of this technique of software delivery is to reduce system complexity through modularity. These systems support openness, as long as components comply with specified standards and interfaces, and this causes issues of trust between components which might be malicious and created to compromise private data shared with them, by users and other components.

The objective of this dissertation is to develop a sandbox for a Component Based System that would reduce the effect of this risk through the use of privacy policies. Privacy policies are typically used in web or enterprise environments to regulate access to data based on conditions. In this project, the policies are used to regulate the transfer of private data between components when they’re making use of other components.

The Sandbox developed generates policies for components and monitors the compliance of these components to the policies when they use other components and private data is shared with these components they’re using. It increments a compliance index when policy is satisfied and decrements it when it is violated. Thresholds of the compliance index can be set where bundles can have their policies “upgraded” based on compliance. Users can also decide whether to uninstall bundles after another threshold of policy violation. The sandbox makes use of logs generated during component use to perform this monitoring.

The Component Based System used is called MCom. MCom was developed in the department by Dr. Omoronyia and is a Java based, lightweight and network agnostic Component Based System.

The sandbox was evaluated using by evaluating its performance and in most cases the sandbox did not heavily impact the function of the Component Based System used. A case was also made for the compliance index helping to dynamically ensure safety of private data as it flows through components, both locally and remotely through remote use of components, which is something MCom facilitates. Finally, suggestions were made about how the work can be improved upon and used in the real world.

Education Use Consent

I hereby give my permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic form.

Name: Ibrahim Bolarinwa   Signature: [illegible]

Acknowledgements

I thank God for giving me the ability and the strength to follow this dissertation through till it had to be handed in.

I'm profoundly grateful to my supervisor, Dr. Inah Omoronyia, for his invaluable support and advice at all stages of writing this dissertation. Thank you for devoting your time to guide and direct me through the execution of this work, till the end. Thank you also for being a kind and amiable teacher to me. I would as well like to thank Professor Johnson for his advice towards the end of the project.

I am eternally indebted to my family for supporting me and making it possible for me to complete this Master's degree. I thank my father, my mother and my siblings for all their encouragement.

Contents

Chapter 1 Introduction

Here the tone of the project is set, exploring succinctly its motivation and the approach for solving the problem it raises.

1.1 Problem Statement

Applying a privacy policy based mechanism in developing a sandbox for component based systems (CBSs) will make such component based systems more secure, specifically reducing sensitive data compromise. Such sandboxes can also manage the policies dynamically in the sense that they can be upgraded if a trend of compliance is noticed.

A security sandbox in any software context serves the purpose of protecting the platform from the potential adverse effects of untrusted software units deployed in said platform [13]. In the context of a component based system, the focus might be on individual untrusted components. Several people have designed security sandboxes for both desktop and mobile based software systems, some of these being component based. A lot of the work so far focuses on the application of techniques like access control restrictions [8], monitoring and logging (of system calls for example) [11, 13, 10], isolation and indirect communication between untrusted and trusted units [4], the use of policies to detect unusual behaviour [9], or a combination of these techniques when designing and implementing their sandboxes. The work on using policies usually tends towards specifying policies for access control and general behavioural pattern.

Observing the sandboxing techniques so far, a few areas have been identified where improvements can be made, particularly in component based systems. In such systems components might provide or require services from one another, and even when techniques described in the previous paragraph are used to prevent malicious behaviour by untrusted components, data privacy between components is still an issue that requires some focus. This is because all techniques do not entirely prevent some sharing of data between components. Therefore, a higher level of detail into what a component can or cannot share with another component is necessary; it will complement direct compromise of data to malicious components.

Another area worth considering is the dynamism of security sandbox systems. Common practice in sandboxes is to statically define what a sandbox can or cannot do. There is also some level of dynamism in constant monitoring and triggering some protective measure (like stopping the running untrusted component) should a component behave maliciously. However, dynamism in sandboxes can also be extended to levels of strictness and trust being determined by the behaviour of untrusted components over time. For example, if an untrusted component is continually given access to some private data over a period, and an observation of its use of this data does not suggest any malicious intent, a sandbox system can decide to permit access automatically in the future, reducing any overhead from explicitly granting this access each time.

To achieve these two aims, a privacy policy management system can be built and incorporated into a component based platform. The idea is to create a system that:

a) Defines the permissions of sandboxed components to private data
b) Manages these permissions, dynamically adjusting them based on the behaviour and choices of the trusted components when interacting with untrusted components, and untrusted components when they gain access to data.

To do this most effectively, the system must sit in between communications between components, particularly ones that involve service requests where some private data might be shared between components. It could also work in a distributed environment whereby components in one deployment invoke services of components in the sandbox of another deployment in a different network location. Such an environment is established in the Component Based System used for this thesis. There are however potential performance costs for this, which are duly discussed.

1.2 Approach

The approach taken to potentially solve this issue is to use a simple privacy policy based system that specifies conditions that components in a component based system must satisfy when they transmit user data to other components. The policies are tied to bundles; therefore they can be modified for individual bundles. The conditions used in this work are relatively simple to specify. There are two conditions, which are consent and notice, which would be explained in more detail further down this dissertation.

The sandbox itself was developed for a Component Based System called MCom, which was developed in the Department of Computing Science by Dr. Omoronyia, who also acted as supervisor for this project. The sandbox was first modified such that components can require and use other components when carrying out activities, creating a case for the sharing of data within components. Logging by the platform was used to record this interaction and a Monitor was used, both instantly and periodically depending on the type of log, to go through the logs to see if satisfaction or violation of policy occurred. It increments or decrements a "satisfaction index" based on this.

The sandbox was evaluated based on two aspects, performance relative to a previous version of MCom, and based on wider discussions on the potential use of the satisfaction index, the effects of longer chains of data sharing between components and network issues.

1.3 Dissertation Overview

This dissertation document is structured rather conventionally. The literature survey, in Chapter 2, explores Component Based Systems, Sandboxes, and privacy policy specification. It makes a case for the approach described in the previous section based on this exploration. Chapter 3 discusses in some detail the state of MCom before this work, requirements and how they were gathered. Chapter 4 provides an explanation of the sandbox's design and implementation.

Chapter 5 discusses evaluation, and Chapter 6 builds upon the matters discussed in the evaluation and suggests methods of improvement and possible adaptations of the project in more real life situations.

1.4 Conclusion

This chapter has provided a quick overview on the objective of this dissertation and how achieving this objective was set about. It is now important to discuss how this objective was reached.

Chapter 2 Survey

In this chapter, an examination of Component Based Systems and current trends in developing security systems and sandboxes will be made, in order to understand in more detail various approaches. First, Component Based Systems will be discussed. This will be followed by a discussion about sandboxes and the concepts behind their design. Since the platform and sandbox that will be developed is to be implemented in Java, sandbox implementations discussed would lean towards Java systems. A discussion about privacy policy mechanisms will follow, and finally a conclusion will be made about how they can be applied to improve upon the various software sandboxing techniques so far.

2.1 Component Based Systems

Component Based Systems (CBSs) are systems that are composed primarily of independent, reusable "chunks" of software known as components [1]. The fundamental gain of CBSs is to avoid monolithic software systems that become difficult to maintain [1].

The foundation of any CBS is a Component Model (CM). The CM defines how a CBS will function. This includes rules for how components can be developed, how they will interact and communicate with other components and the outside, how components can specify services they provide and/or require, how they can be deployed and other environmental constraints that they must satisfy [1]. Examples of CMs include Fractal, OpenCom, Enterprise Java Beans (EJB), Open Service Gateway Initiative (OSGi), the CORBA Component Model (CCM), etc. [2, 1, 3, 4].

Component Frameworks are implementations of CMs [2]. They are commonly referred to as middleware, as they sit in between the underlying system environment (e.g. an Operating System) and installed components. Examples of component frameworks include the Julia implementation of Fractal [2], Felix, Equinox and Kloplerfish implementations of OSGi, etc.

There are different definitions for components. The most common definition, by Szyperski et al [5], defines a software component as "a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties". Another definition, by Heineman, [6] adds to Syzperski's definition the fact that components comply with a CM.

2.1.1 Data Security Needs of Software Platforms

Security for Software Platforms primarily involves the protection of resources the platform has some control over from malicious and exploitative agents. More important amongst these resources is the data that flows through the platform. A platform might suffer from a direct attack through a network, e.g. exploiting a potential network loophole, but it might also be attacked more subtly through malicious modules or applications, which might appear legitimate but covertly steal data.

To prevent malicious incidents of the latter type, software platforms need to verify modules as they are deployed, as well as manage and regulate how data flows through a system and how it is accessed by modules (dynamic analyses). Dynamic analysis is important in order to curb compromises that might not be detected during the initial verification process. Component Based Systems particularly need mechanisms for this dynamic analysis since a lot of the focus on developing CBSs that are open and adaptable.

2.2 Example Component Model: Fractal

Fractal Component Model (FCM) is a CM created by Bruneton et al. [2] FCM is designed to be highly stratified and reflective, providing composite components which consist of multiple sub-components and reflective capabilities that provide opportunities for the modification of the internal structure of components. The reflective capabilities are referred to as "control". There is also provision for components to be included in multiple other components as shared components. With these properties, components become more configurable, improving their adaptability. Fractal also provides "plain black-box objects", which have no control, for the purpose of legacy support.

Julia is a Java implementation of FCM, and it serves as its reference implementation. There are other implementations, including AOKell (Java), FracTalk (SmallTalk), FractNet (.Net), Plasma (C++), etc.

2.2.1 Fractal Components

Fractal Components are distinctly identifiable and enclosed units that are accessible via interfaces. [2] The two major varieties of interfaces are the client interfaces, for issuing commands, and server interfaces, for accept incoming commands. Fractal components consist of a limited amount of sub-components in a "content", and a "membrane" that houses internal and external interfaces. The membrane is the primary element that brings about reflective capabilities in a component, as the interfaces it houses enable observation and modification of a component's internals.

The internal interfaces of the membrane can only be accessed by the content's sub-components, while the external can be accessed from outside the component [2]. The sub-components can be collectively represented by the membrane. The membrane, through "Controllers", can also control the behaviour of the sub-components. For example, it can stop and resume their activities.

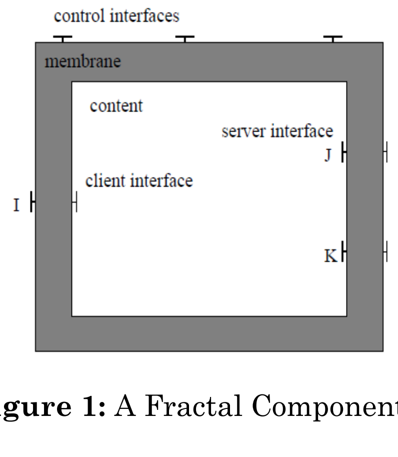


Figure 1: A Fractal Component [2]

2.2.2 Fractal Bindings

Bindings in Fractal are the model's method of component communication. They occur between the interfaces of components [2]. Bindings are of two kinds: primitive and composite. The primitive bindings are direct bindings between a single client and a single server interface. Composite bindings occur within any number of interfaces.

2.3 Example Component Model: OpenCom

OpenCom is a generic, lightweight and language independent systems software targeted CM [3]. OpenCom was designed to address a need for CBSs for systems software, as distinct from application software CBSs. It was designed to be usable in any kind of systems software environment, from embedded systems to PC operating systems. To achieve this, the creators tried to capture the most essential needs of these systems, avoiding any non-essential domain and environment specific needs. By being customisable and expandable, OpenCom can be made to fit these specific needs. It was also built to have minimal memory and processor overhead in line with its versatility.

From a bird's eye view, the model consists of a basic component runtime kernel for starting up and connecting components, and a Component Frameworks layer, which is slightly different from the notion of CFs introduced earlier [3]. This layer can loosely be defined as independent collection of components that tackle a particular concern. Extensions can be in housed in this layer. Extensions are components that specifically add to the basic capabilities of the runtime kernel, facilitating customisation and expandability of OpenCom environments. According to [3], there are currently two major types of these extensions: platform extensions, which provide environment specific customisations, and reflective extensions, which provide reflective and adaptive customisations.

2.3.1 OpenCom Components

OpenCom components are quite similar to Fractal components in terms of definition. They also communicate via interaction points (similar to interfaces in Fractal) that are connected through bindings, though there are slight syntactic and semantic differences in its application of the concepts. Components are deployed and started in "Capsules", which are holders that grant components access to the runtime kernel API [3].

Interaction points are either "interfaces" or "receptacles". Interfaces are similar to server interfaces in Fractal. They are points for accepting invocation of services. Receptacles are conversely somewhat similar to client interfaces in Fractal. However, the [3] mention that it can also be for third-party composition, where components specify other components whose services they require [3]. This is because receptacles are specifications for services that components need from other components. Components can have zero or more interfaces or receptacles.

A binding is one to one connection between an interface and a receptacle [3]. An interface can be bound to more than one receptacle at a time, but a receptacle can only be bound to one interface. A component can create a binding between two components. In fact, bindings are represented as components, except that a termination of such a component also terminates the binding.

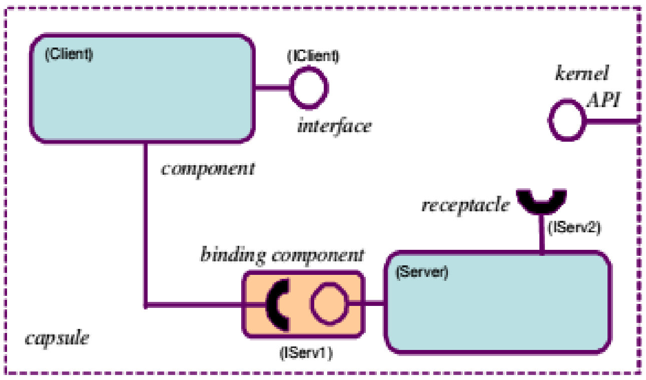


Figure 2.2: OpenCom Model [3]

2.4 Example Component Model: OSGi

Open Services Gateway Initiative (OSGi) is a Java CM developed and maintained by the OSGi Alliance. [7] It was initially created as a lightweight, dynamic specification for the development of reusable components deployable in household appliances, so as to achieve home automation. However, it has witnessed widespread use in application development for desktop, mobile as well as other platforms.

According to the OSGi Alliance, [7] the major aims of OSGi are to provide a simple but efficient model where the component implementations can be modularly encapsulated, while providing functionality through services that can be advertised to other components. Also, components do not have to be aware of the behaviour or availability of other components, just the services that the component requires. Lastly, dynamism is achieved through the ability to start, stop, add or remove components at system runtime.

OSGi specifications [7] are rules for the functionality an OSGi system should provide to a component. Frameworks are environments that aim to satisfy these rules. However, frameworks in OSGi are separable from implementations of a framework, in the sense that the frameworks are a blueprint the implementations build upon.

Examples of OSGi framework implementations include Apache Equinox, Felix, Kloplerfish, etc.

2.4.1 OSGi Components and Framework

OSGi components are called bundles. Bundles are Java JAR files with metadata included in a "manifest file". The manifest file specifies the services the component needs to function and the services it renders, again similar to the interfaces that exist in Fractal and OpenCom. It also specifies the bundle's identifier and version, as well as other relevant information.

OSGi Frameworks are structurally divided into layers, which are as follows [7]:

Execution Environment: This is usually either a subset of or a version of the Java Runtime Environment. It specifies the classes and methods the other layers and methods have access to.

1. Module Layer: This layer uses the information in the manifest file to determine services offered and dependencies required by bundles.
2. Lifecycle Layer: The lifecycle provides the ability to dynamically manage bundles, i.e. the ability to start, stop, add or remove bundles mentioned earlier.
3. Services Layer: The Service layer provides a mechanism for service advertisement and discovery. The services are represented as plain Java objects.
4. Security Layer: This is an optional layer that takes advantage of the Java 2 code security setup. It can make use of policy files to specify fine grained access control for code to various resources.

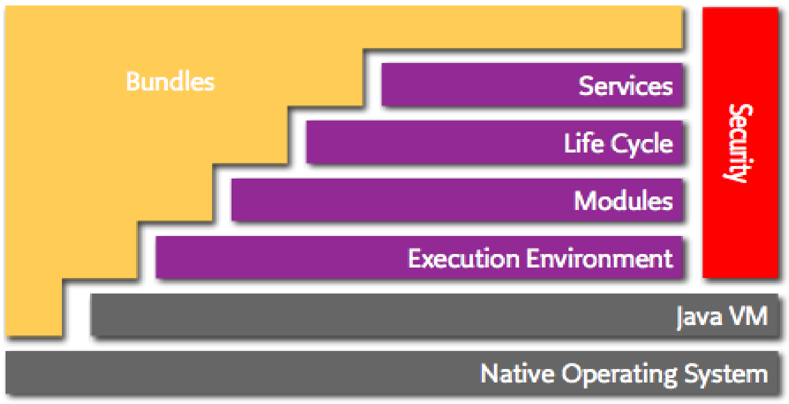


Figure 2.3: OSGi Framework Layers [7]

2.5 Why CBSs Need Sandboxes

The major lessons that can be learnt from the CBSs discussed are that these systems are designed to be adaptable (through reflective functionality), modular and standardised such that components can easily be deployed and managed as long as they conform to the standards (openness). All this is aimed towards smooth coexistence of modules. Most of the CBSs discussed do not address security as a fundamental concern in their specifications, and the ones that do do not specify rule based regulation of data sharing between components. This means untrusted components are a security issue that aren't a primary design element in CBSs. Fractal, for example does not specify any particular security specification built in, as the focus is on providing an easy to extend and adaptable model [2]. The focus when developing OpenCom was it being lightweight enough to capture the universal requirements for systems software, and security requirements wasn't one of these requirements [3]. OSGi does have a security layer in its specification, which depends on the Java security architecture [7] and offers "fine - grained access control". It is however quite general, and there's no specific way to dictate for example what must be done before a bundle A sends data to a bundle B that provides a service that it requires.

Sandboxes can solve this kind of problem. Of course, sandboxing for CBSs isn't a new concept, at least relatively. A short discussion of sandboxing systems already implemented, mostly for CBSs follows, to see how existing sandboxes cater for the issues just highlighted. The argument is that while a lot has been done, very few sandboxes address this issue specifically.

2.6 Malicious Behaviour

The discussion about adverse behaviour in software systems can take different directions. Viruses are popular and encompass a wide range of adverse behaviour, from slowing down a platform to stealing sensitive information. StrangeBrew and BeanHive have been cited as viruses in Java based systems, [8, 9]. PJApps and HongTouTou [10] are trojan viruses that affect Android based mobile systems.

There are also discussions about adverse behaviour outside viruses. Huang et. al. mention several, [9] including Denial of Service attacks that might be triggered by malicious while loops, shared object attacks (illegal access to an object A through another object that has access to A), etc. Enck et al [11] discuss about how mobile applications can compromise sensitive information collected from mobile devices like location data, SIM card details, etc, sending them to advertising sites or dangerous cloud services. This can be perpetrated by viruses inserted into unofficial versions of legitimate applications, and applications created specifically for malicious ulterior motives. They can also gain access through third party code in official applications [12].

2.7 Sandbox Review

Inoue and Forrest [8] implemented dynamic sandboxing of java programs using a combination of java security policy files and The Principle of Least Privilege. Dynamic sandboxing in the context of their work refers to the generation of java policy files detailing the minimum permissions a java program needs to properly execute. The generation of a policy happens during a training period where a "clean" version of an application is run with a security manager that generates the policy. Their system is effective and dynamic in a sense, as in theory malicious versions that try to violate the minimum permissions could be stopped from doing this. It doesn't however consider the wider concern of an exploit free application sharing data with another application that has malicious intent.

Huang et al [9] in their work describe an OSGi security layer that fits between the OSGi framework and the Java Virtual Machine. The security layer monitors events and logs events that have a security implication. A module (Detection Engine) matches security or stability breaches against a pre-processor that contains rule-based policies and a pre-defined response is triggered accordingly. There wasn't a lot of discussion as to how these policies would be specified or updated. Or how the policies might treat potential data exploits.

Gama et al [4] also implemented a system specific to OSGi. Their work is rather interesting, as it employs Aspect Oriented Programming (AOP) in improving dependability, security and resilience in OSGi platforms. They implement a sandbox, and in their sandbox the 2 main techniques applied are indirect communication between main and sandboxed components through a proxy, and an isolation policy for sandboxed components. However, there isn't much discussion about what the isolation policy entails. They also implemented monitoring, which was external to the sandbox. The main purpose of their monitoring is "self-healing" of the sandbox; restarting components or the entire sandbox if this is perceived as required. The proxy system is good, but there also isn't much discussion as to how the proxy can prevent the compromise of private data.

Bläsing [13] focuses on sandboxing in the Android Mobile Operating System. Their system does static and dynamic analysis of android application installation files. The static analysis examines the android installation file (APK file) and looks for patterns that might have security implications like reflection or security permissions. The dynamic analysis (the sandbox) sits at the kernel level of the android operating system and examines and logs calls made by the application when installed. Their implementation works thus: they install the application in an emulator environment. Once installed, they use an application called Android Monkey to trigger events on the emulator that the application running on the emulator might respond to, making system calls. The calls are logged by the Sandbox for later analysis. There's no explicit discussion of the implications of their log data or what they've learnt or deduced from it, or how it should be interpreted. Also, the sandbox takes no proactive measures to prevent malicious usage. It appears their logs might rather be used manually. Finally, the system does not appear to be usable in an actual android environment, on a mobile phone.

Another work that tends towards the Android OS is [10], where they develop a Behaviour Based Malware Detection for Android Applications. They collect application and system call data made by android applications on users' phones (this is classed as behavioural data). This data is stored in a remote server and analysed for finding malware. To accomplish this, they developed an android application called "Crowdroid", which monitored Linux kernel system calls and sent them to the centralised server. Once the data is collected, they use a clustering algorithm to find patterns such that unaffected installation instances of an application will exhibit similar behaviour while an infected instance will exhibit behaviour that deviates from aforementioned similar behaviour. This therefore would enable the identification of these infected instances. Their system does provide a way for use in an actual android environment, but suffers from the problem of not being able to detect malware without having significant training from uninfected versions of an application.

One work that was quite data-centric was Enck et. Al.. They implemented "TaintDroid", [11] a tool for tracking the flow of sensitive privacy related data within applications for the android mobile operating system platform. This is done in order to detect applications that compromise this data by sending it to advertising sites or other malicious locations. Examples of sensitive data in their paper include location based data, device ID, phone number, SIM card serial number, etc. TaintDroid works by identifying sensitive data and attaching a taint marking to it. The data is then tracked as it interacts with other data, noting how this might result in a data leak. This is called Dynamic taint analysis. The data monitored is tracked until it departs from the system, usually through the mobile's network module. One good thing about their work is how the system brings a potential solution to the problem of data being leaked between applications or components in a system. They also strive to do this in as efficient a way as possible, taking into account the constraints of the Android OS. The entire process can however have a significant performance drain, particularly in more complex systems.

In general, most of the techniques for security in Java systems surveyed implement mechanisms for preventing general malicious behaviour. TaintDroid is an example of work that focuses on the data aspect of malicious behaviour, and further work on this area is worthy of more effort. The aim of this work, as stated earlier, is to explore a potential way of ensuring data privacy isn't compromised in sandboxes for Java component based systems: privacy policies.

2.8 Privacy Policies

Privacy policy is concerned with the representation of rules that specify what users can share with a system. In an enterprise environment, it might also refer to conditions that stipulate access to specific system resources [15].

2.8.1 Policy Languages

Policies in the software context are essentially rules for specifying acceptable pattern of behaviour. Privacy policies specify how accessible resources, though more commonly data, that users and organisations have control over should be [14]. Privacy policy languages can help both parties regulate their privacy rules [lobo]. It is quite common place for these languages to be in XML format. Common examples of privacy policy languages include Platform for Privacy Languages (P3P), eXtensible Access Control Markup Language (XACML), A P3P Preference Exchange Language (APPEL), Enterprise Privacy Authorization Language (EPAL), etc [14].

Policy languages are designed for different contexts [14]. For example, P3P, one of the first language specifications, was designed for use on the web, EPAL was designed for use in Enterprise systems, and GeoPriv for location based access. XACML is classed as a Sophisticated Access Control Language that can also be used in Enterprise settings.

EPAL and XACML meet most of the requirements for policy definition and application [15]. They are both platform-independent and provide a representation method for applicable policies [15]. While they have slightly different syntax and semantics, they follow the same standard for policy enforcement model [15]. This shared model is discussed below.

2.8.2 Policy Enforcement Model

The standard common to XACML and EPAL separates the concerns for policy enforcement primarily between a Policy Enforcement Point (PEP) and a Policy Decision Point (PDP) [15]. The PEP receives a request from an entity seeking access to a resource, and formulates an authorisation request using an agreed upon format to the PDP. The requests are formulated using "attributes", name-value pairs with associated type that might describe the requester, the resource, or the action that is to be performed on the resource.

The PDP evaluates an authorization request based on policies specified in the system. A policy can contain a single or multiple "rules". The rules specify applicability, which refers to the kinds of requests it can be evaluated against, conditions that must be met for the rule to be used, and an effect, which is usually permission or denial of access. The PDP sends an authorisation decision to the PEP which then grants or refuses access based on the decision.

The policy languages are mainly standards and implementing them is a responsibility for enterprises. The full set of features provided by the languages are too complex and beyond the scope of this project. XACML and EPAL are meant to cater for a wide range of enterprise functions, involving different roles. XML as a choice for language representation will however be adopted, because of its universal nature and easy parsability.

2.8.3 Dynamic Policy Management

Dynamic Policy Management in this paper involves a Policy system being able to learn policy requirements from user behaviour and interactions with untrusted software. Adding this to policy management is efficient because users might not be able to predict all the scenarios a particular policy will be used and the preference for that scenario [16]. Bandara [16] uses Inductive Logic Programming to automate the learning of policy requirements, but the results of the work weren't accessible at the time of writing this dissertation.

2.9 Conclusion

In this chapter, Component Based Systems, Sandboxes and Privacy Policies have been discussed in order to justify a need to apply privacy policy mechanisms to sandboxes for component based systems. CBSs, being open, dynamic, and adaptive, have a need to protect private data from being exploited. What follows next is a discussion about the design and implementation of a CBS and sandbox to test this hypothesis.

Chapter 3 MCom CBS and Requirements

Here the MCom Component Based System, upon which the Sandbox was developed, is introduced. MCom has some similarities to OSGi, as it is also java based and components are packaged as Java JAR files. Requirements are also discussed, with a view to summarising what the system is meant to do, albeit in more detail than before.

3.1 MCom1.0.0

Implementing the ideas discussed in the previous chapters requires a Component Based System to build the sandbox upon. The MCom CBS was chosen due to familiarity with the platform, as well as time constraints that made it difficult to adopt a more popular CBS like OSGi. MCom was initially developed by Dr. Inah Omoronyia (project supervisor), although its basic contract invocation feature (to be explained later) was developed as part of an assignment. It is a network agnostic CBS; it communicates over a local network via the underlying Operating System's network interface, using Java's Socket API. This is what gives it its lightweight property; it can work properly in UNIX based Operating Systems as well as Windows without modifying the system, as long as the appropriate version of Java Runtime Environment (JRE) is installed (Java 8).

Components in MCom are referred to as bundles. Like OSGi, bundles in MCom are also Java Archive (JAR) files, but relevant metadata is added through the use of Java annotations (to be discussed later). Bundles are modular and provide functionality in the form of contracts. The contracts are actually annotated Java methods. Bundle contracts can be advertised to and be found by other mCom instances through special mCom instances called registrars.

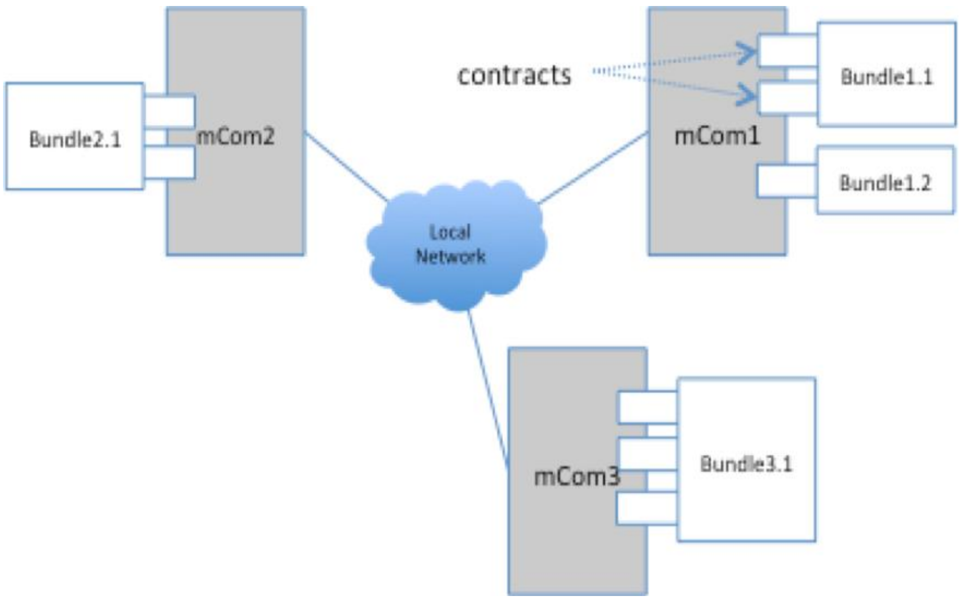
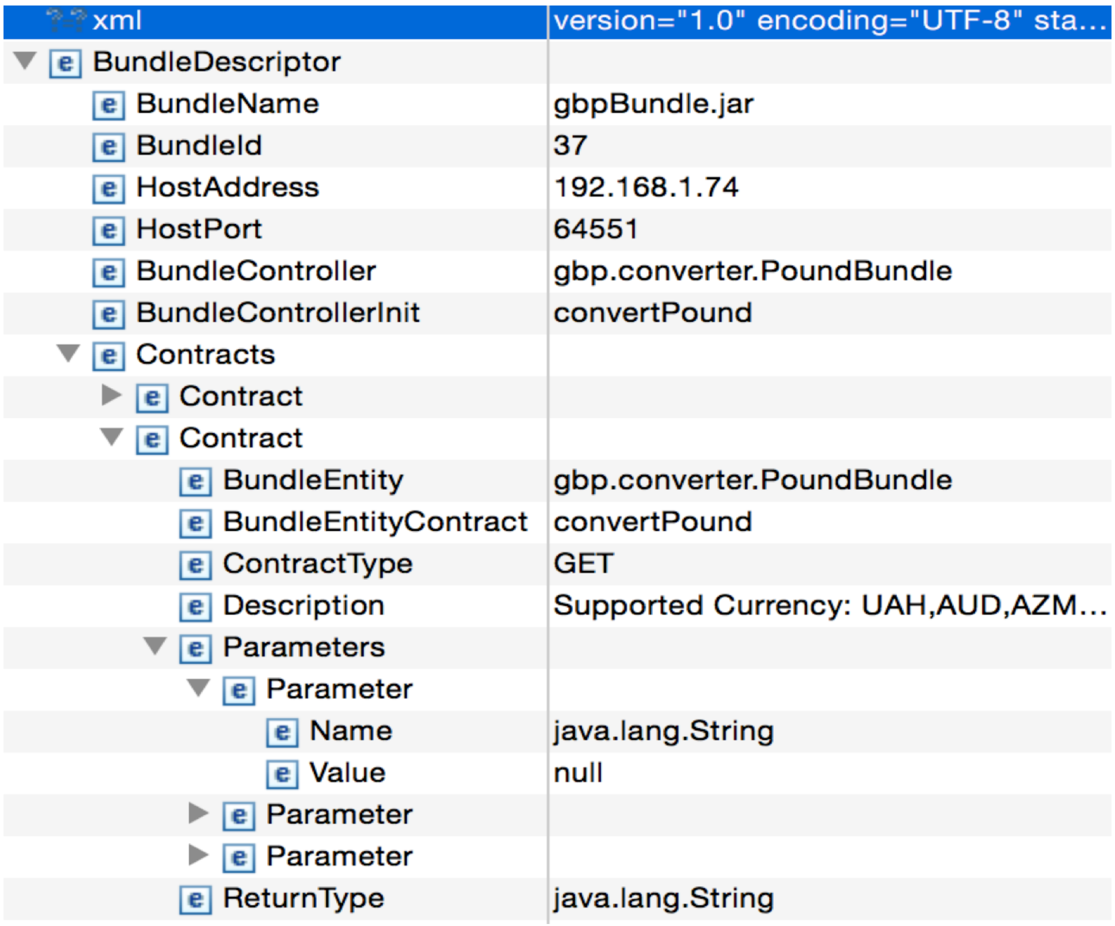


Figure 3.1: Structural overview of MCom Component Platform

MCom instances can be instantiated and interacted with through command interfaces. There are a number of commands that enable users deploy, advertise, and find bundles, invoke specific contracts in bundles, etc. Bundles to be deployed are placed in a file directory, and when deployed, a bundle descriptor is generated for each bundle. The bundle descriptor is also stored in an XML file format, and used to retrieve information about individual bundles while bundles are active. A sample xml tree diagram of a bundle descriptor is shown below:



xml	version="1.0" encoding="UTF-8" sta...
▼ BundleDescriptor	
BundleName	gbpBundle.jar
BundleId	37
HostAddress	192.168.1.74
HostPort	64551
BundleController	gbp.converter.PoundBundle
BundleControllerInit	convertPound
▼ Contracts	
▶ Contract	
▼ Contract	
BundleEntity	gbp.converter.PoundBundle
BundleEntityContract	convertPound
ContractType	GET
Description	Supported Currency: UAH,AUD,AZM...
▼ Parameters	
▼ Parameter	
Name	java.lang.String
Value	null
▶ Parameter	
▶ Parameter	
ReturnType	java.lang.String

Figure 3.2: XML Tree Diagram of a Bundle Descriptor

The BundleController is a class that controls the bundle. The BundleControllerInit is a method that intialises parameters and calls methods required for a bundle to run appropriately during invocation. This method has to be in the BundleController. A Contract's BundleEntity is simply its container class.

Screenshots of MCom instances in use are in the appendices.

3.1.1 MCom Commands

To give an overview of the actions that can be carried out on MCom, here is a list of the MCom commands:

Command	**Action**

isReg%<bool>	Enables the instance to switch on/off registry service.
drs	The instance executes Dynamic Registrar Discovery. This is the automatic discovery of instances of MCom on the local network that assume the role of a Registrar.
reg	This command displays a list of available Registrars that an instance can advertise or look for bundles.
deploy	deploys all bundles in deployment directory. Only deployed bundles can be advertised, looked up and invoked.
llookup	Command prints the BundleDescriptor of all locally deployed bundles
rlookup	Command executes a remote lookup for all advertised bundles with known Registrars. Can only do this if drs has been executed.
adv%<bundleId>	This command advertises bundle to all known registrars in the form of the bundle descriptor XML.
invoke	A remote invocation of a specified bundle contract.This function requires bundleId, contract name and contract parameters as input.

Table 3.1: List of MCom commands

3.1.2 Contract Invocation Mechanism

MCom encodes messages sent to other instances primarily in XML format. The typical set of steps involved in sending a contract invocation over the network includes invoking a command to dynamically locate registrar MCom instances, then invoking a remote lookup command that will return a list of bundle advertisements in form of the host and bundle information, very similar to the tree diagram above. A user can then invoke a contract by supplying name of contract and arguments which are encoded and sent to recipient.

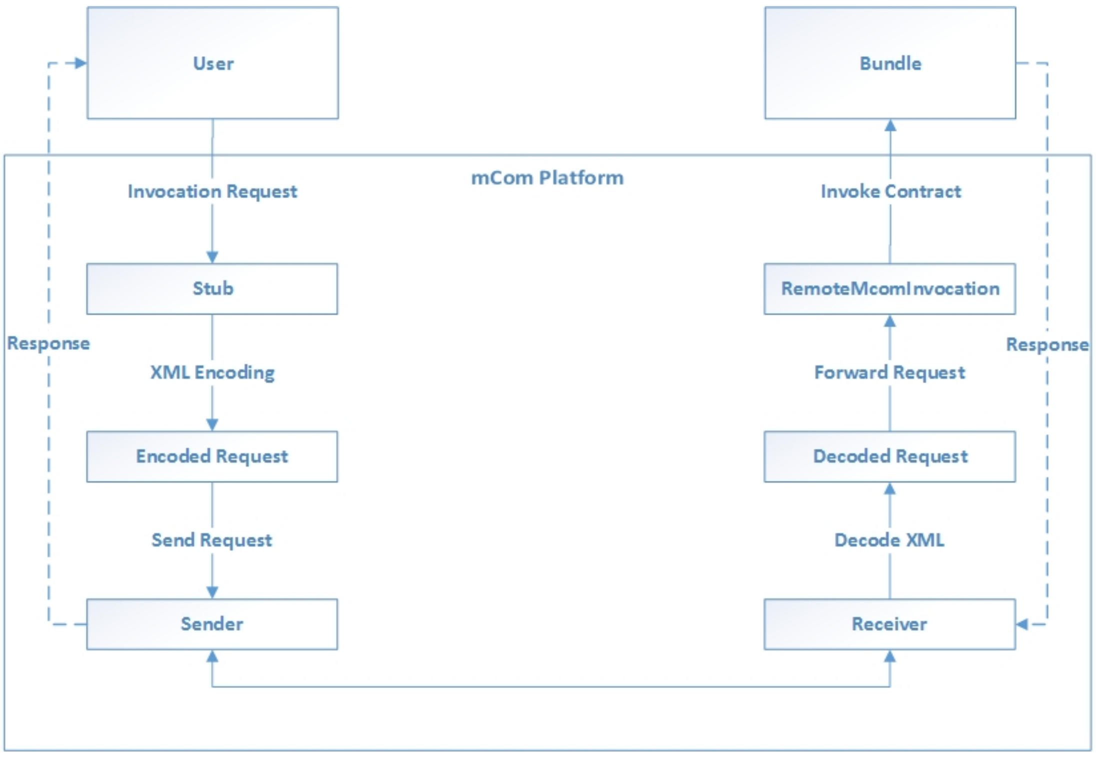


Figure 3.3: Contract Invocation Mechanism

3.1.3 Libraries Used

XML building and parsing by the platform was done using W3C's Java DOM parser library. This library was adopted as it provides a convenient and efficient API for parsing XML documents.

There is also an m-annotations library that hosts all the annotations used in MCom bundles.

3.1.4 Communication

As mentioned earlier, MCom communicates mainly by using Java's API for Socket - based network programming. Messages sent between MCom instances are encoded with headers describing the type and content and the sender, and a body which is typically structured in XML form, at least for the more lengthy messages.

An MCom instance has Sender and Receiver classes that implement means to send and interpret all the kinds of messages the instance might want to send and might receive from another instance, either on the same machine or in the local network. The receiver in particular implements a thread that defines a ServerSocket, which listens for messages that might be sent to it. When instances advertise bundles or invoke other bundles, the address and port of this ServerSocket is sent as part of the message. The Receiver thread is initialised with the MCom instance.

3.2 Requirements

As previously elucidated, the Sandbox to be built is concerned primarily with data privacy and prevention of private data leak between components in a CBS. The formation of requirements for the sandbox therefore focused on the functionality that the MCom CBS can use to fulfill this purpose.

Requirements were gathered through the survey of Component Based Systems, Sandboxes and Privacy Policy Systems discussed in the second chapter, mainly through observing what has been done and discussions with the project supervisor on what can be improved about CBS Sandboxes. There were also discussions about use cases involving MCom contract invocation where steps taken by a sandbox might lead to better fulfillment of privacy requirements. The requirements mainly discussed were functional.

3.2.1 General Improvements

These requirements were not available in MCom before the sandbox was implemented. Providing this functionality would enable better evaluation of the sandbox. These requirements are listed as follows:

- A mechanism for the determination of whether a bundle should be trusted or not should exist and be invoked during deployment.
- A method should exist for contracts to be able to require and invoke functionality from other contracts in other bundles (Inter-bundle Contract Invocation). This functionality would enable data sharing between bundles, which is a major function for which the sandbox is being built .It would also make MCom potentially a distributed CBS if contracts can invoke other contracts in bundles located in other computers in a local network.
- Contracts in bundles should be able to specify other contracts that they require.
- Contracts should be able to lookup bundles that provide contracts they require.

3.2.2 Sandbox

The main requirements for the MCom Sandbox were considered and categorised in terms of policy generation and management, policy enforcement, logging and the provision of functionality that would enable bundles use functions provided by the sandbox. They are listed as follows:

- The MCom sandbox should be able to classify bundles as trusted or untrusted based on the evaluation mentioned in the first bullet point in General Improvements.
- The sandbox should be able to generate initial policies for bundles based on this classification and store these policies.
- The sandbox should provide a means for the MCom platform to observe bundle policies and individual bundles to retrieve their policy in order to know conditions to be satisfied for data sharing between bundles.
- The sandbox should be able to log contract invocation
- The sandbox should perform dynamic management of individual bundle policies. This should involve ensuring that bundles comply with policies.

- The sandbox should have a method of indicating the levels of policy compliance in bundles.

The approach of generating individual policies for bundles, rather than applying one or more general policies to all bundles, was taken in order to apply fine-grained policy management.

3.3 Conclusion

The MCom CBS, while not mainstream, does possess the general characteristics of CBSes, including modularity and the provision of services that can be invoked. The CBS misses a way for components to interact with each other, which is integral to the sandbox's function. Remedying this is discussed, along with how the sandbox was designed, in the next chapter.

Chapter 4 Design and Implementation

This chapter features discussion on the design and implementation methodology of the MCom sandbox. The design and implementation of each feature is explained hand in hand. The discussion covers the modifications made to MCom, the privacy policy system employed, and how the sandbox as a whole functions.

4.1 MCom Modifications

These correspond with the general improvement requirements mentioned in the previous chapter. The modified MCom version will be referred to as MCom1.0.1.

4.1.1 Initial Trust Mechanism

An initial mechanism for determining whether a bundle should be classified as trusted or untrusted was added. Since the bundles are packaged as JAR files, a simple mechanism for doing this is to check if the files are signed during the deployment process. This is the approach adopted by MCom 1.0.1. The choice of this mechanism was made as it was relatively simple and realistic to implement in the given timeframe of the project. Signed JAR files are recognised as trusted and not sandboxed, while unsigned JARs are sandboxed. The private key and certificate used in MCom 1.0.1 was generated locally with Java's keytool. Within code, the check is easily done by running the java command for checking if a JAR file is signed.

During invocations, a user is warned if selected bundle is sandboxed and asked if they want to proceed.

4.1.2 Inter Contract Invocation

To facilitate invocation between contracts, bundles specify the contracts a particular contract requires, through annotations. The bundles can search for other bundles that have the contracts required by an individual bundle, as well as invoke these contracts through a library called mcom-core, which was written as part of the project implementation. The library is external to the MCom platform, and performs these functions through Java Reflection. During deployment and bundle descriptor generation, the required contracts information is encoded in the description of individual contracts. A check whether the contracts required by a contract exist in the remote lookup results of an MCom instance is done when an attempt is made to invoke the contract needing them.

Bundles can search for bundles offering a contract through mcom-core. It returns information about bundles as XML bundle descriptors. The API might return more than one bundle for a particular contract that is required. The task of selecting choosing one (or more) is left to the bundle requiring the contract. The mcom-core library performs other functions that will be discussed further. Further documentation for the library is available in the Appendices.

The figure below can fit into the first side of figure 3.3, with the bundle replacing user and mcom-core being in between the bundle and the Stub. Reflection is required because mcom-core isn’t a part of the MCom-platform code. In this case, the library communicates reflectively with the Stub.

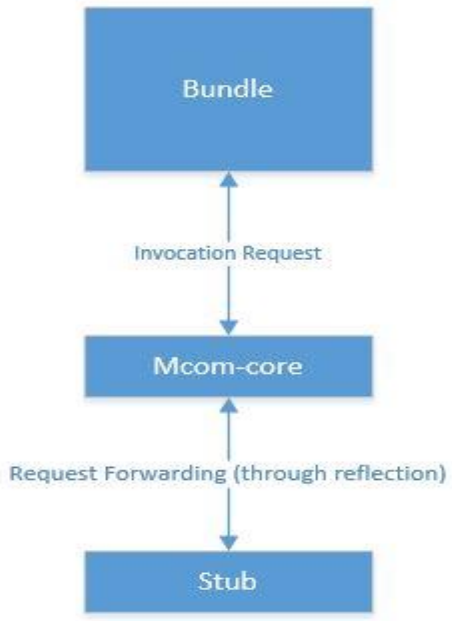


Figure 4.1: Inter Contract Invocation

4.1.3 Sensitivity Levels

Sensitivity Levels are used to determine which data supplied by a user is private. The Sensitivity levels used are private and public. When invoking contracts, users can annotate the arguments as private. Bundles that receive such arguments are expected to comply with policy specified for them i.e. seek consent from and notice to the user invoking them if consent and notice are required.

Contracts in bundles are also able and expected to set a sensitivity level for their return values. Other contracts in bundles that invoke contracts with private sensitivity level are expected to comply with the policy specified for them.

Mcom-core encodes the results from a contract invocation in the form of an InvocationResult, which includes the result itself as well as the sensitivity level sent by invoked bundle.

4.2 Policies

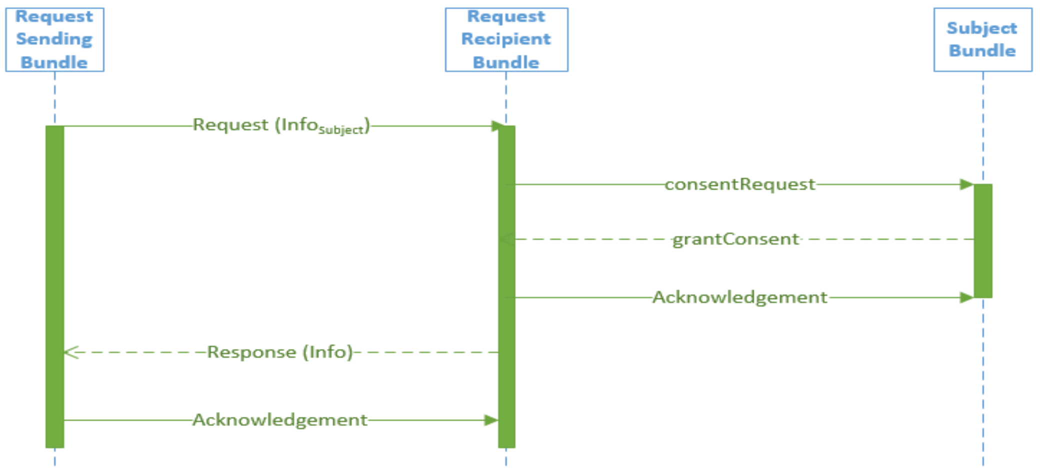
The privacy policy is centered on two conditions. The conditions are notice and consent. As the names imply, consent involves seeking permission and notice involves sending a notification. The purpose of consent is to give data owners control over the flow of data through components, while the purpose of notice is to provide a data owner assurance. The policies are used for the management of inter contract invocations.

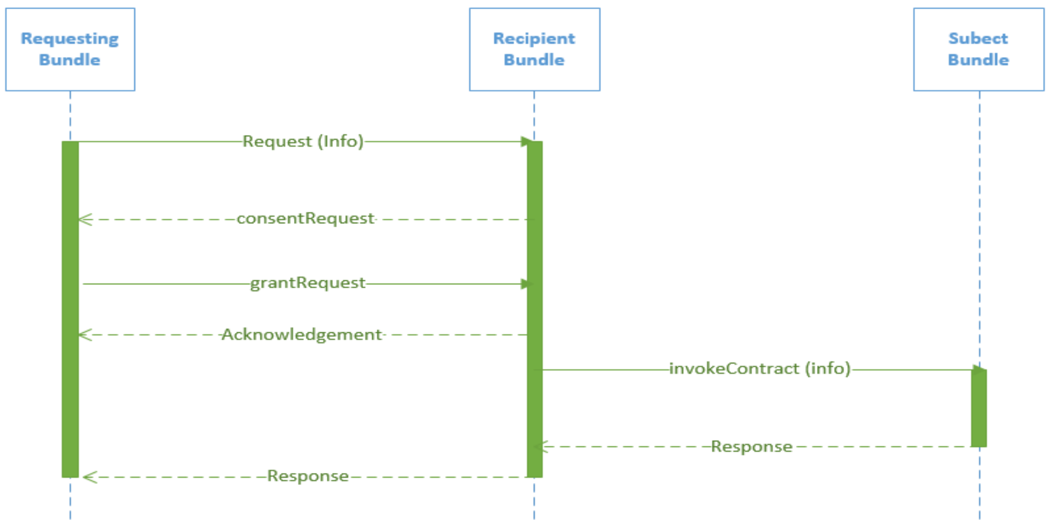
The policies are modeled based on communication transmission principles that involve a sender, a receiver, and a subject. The subject constitutes the information or service being requested for by the sender. The subject might or might not be within the same machine. In the context of this project, a subject is information returned by a contract in a bundle being invoked remotely.

4.2.1 Consent

For a bundle to fulfill the consent condition, it is required to seek permission from suppliers of private data before disclosing it to another party. On one hand, an entity (a user or another bundle) invoking a contract that sends private data to that contract must be sought permission from before that data is passed on to another bundle by the invoked contract. On the other hand, a contract being invoked that sends a private result needs to be sought permission from before the result is passed on to a requester. Both scenarios are illustrated in figures 4.2 and fig. 4.3. In fig. 4.3, the response sent from the recipient bundle to the requesting bundle can be preceded by a consent request from the subject bundle, as depicted in fig. 4.2.

Consent also involves an acknowledgement to the entity giving permission that the response to the consent request has been received.





Figures 4.2 & 4.3: UML sequence for Consent policy compliance

4.2.2 Notice

Notice would typically occur after a consent sequence, though in the sandbox it can occur without if a bundle is upgraded to not requiring consent. Notice involves both the subject and the requester; the subject must be notified, while the requester should be notified that the subject has been notified.

MCom-core also facilitates the sending of notice messages, although the second part of the notification involving the requester wasn't implemented in the final sandbox.

4.2.3 Policy Representation

The policies are represented as XML files that correspond to individual bundles. The XML is generated for each bundle as the bundle is being deployed, and stored in a directory for policy files, so that policies can be adjusted individually for each bundle. Mcom-core also provides a mechanism for bundles to retrieve their policies as objects. The policy encoding itself is not complex, it simply specifies if a bundle has to do consent and or notice in true or false. There is also a Policy object that is used to represent a bundle policy within code.

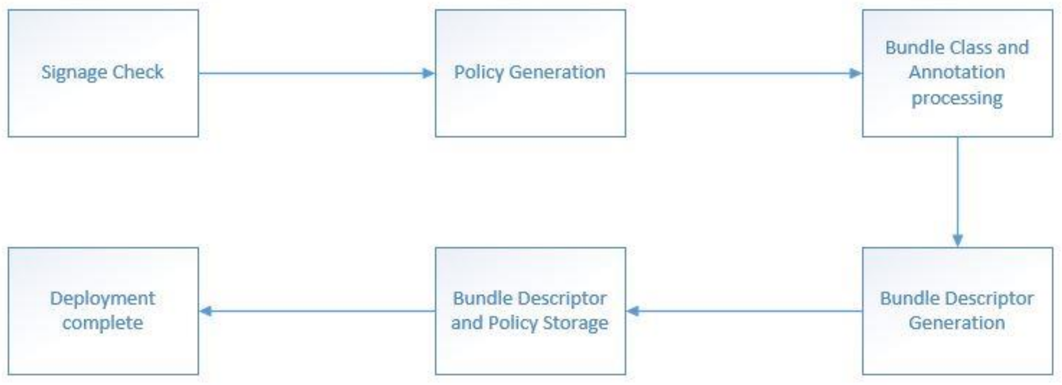

Signage Check
Policy Generation
Bundle Class and Annotation processing
Bundle Descriptor Generation
Bundle Descriptor and Policy Storage
Deployment complete

Figure 4.4: Modified Deployment Process

4.2.4 Communication

The Mcom-core library provides bundles with the ability to seek consent and send notice. Both the sending of a consent request and a notice notification will be referred to collectively as notifications further down this document.

As mentioned earlier, MCom instances implement a Receiver class which starts a ServerSocket that receives messages from other MCom instances, including invocation messages. A separate socket (with same address as initial socket but different port) is defined for receiving consent and notice notifications. The Mcom-core library encodes the consent and notice messages and sends them to this socket, and also returns a reply (like if consent is granted or not) to the sending bundle.

At runtime, the bundles require a means to figure out what address and port to send the request to, as this cannot be determined before an MCom instance is initialised. As a result, all bundles are required to be subclasses of an MComEntity class, which is included in mcom-core. Through reflection, the address and other relevant details about its current execution environment are passed to bundle entities by setting the properties of the bundle entity class inherited from MComEntity. This is done during contract invocation.

4.3 Sandbox

The major tasks of the sandbox are to verify that policy conditions are met by bundles and that policies are adjusted based on the satisfaction of policy behaviour by bundles. The first thing it has to handle is policy generation. The default policy requires both consent and notice to be fulfilled for all private data. MCom1.0.1 provides a mechanism for manually updating the policy of individual bundles.

The sequence that each bundle contract is expected to execute when it receives private data from an invoker entity A and performs an inter contract invocation to C where it sends C the aforementioned private data is as follows:

1. Retrieve bundle policy.
2. If consent is required, seek consent from A.
3. If consent is granted, perform the inter contract invocation.
4. If notice is required, send notice to A.
5. Verify that invocation result from C isn't private. If it isn't, forward this result to A and end. Else continue this sequence.
6. If consent is required, seek consent from C.
7. If consent is granted, send result to A
8. If notice is required, send notice to C.
9. End.

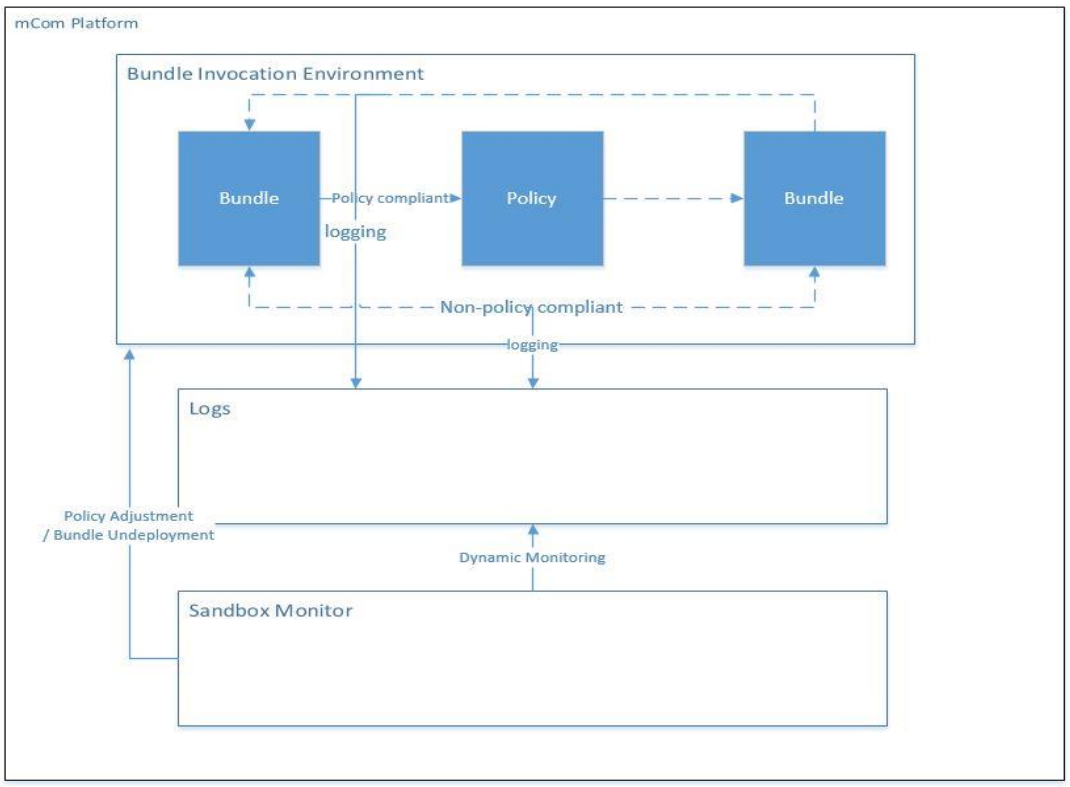


Figure 4.5: Sandbox Overview

4.3.1 Logs

Logging is essential for the sandbox to be able to monitor bundle activity. Logs are the primary method that the Sandbox Monitor uses. Logging is done by the mcom-core library when notifications are sent, and in the MCom1.0.1 stub and Receiver implementations.

Plain Old Java Objects (Pojos) are used to encapsulate log information. The objects used for logging include:

1. ContractInvocationLog: This object stores the basic details for a contract invocation. This includes an array of the private arguments sent in this invocation, the name and ID of the bundle being invoked and the address of the entity that invoked the contract.
2. ContractInfo: This pojo encapsulates information about the contract requiring and the contract required, like bundle name and address.
3. InvocationResult: An invocation result is encapsulated as a result string, as well as a sensitivity enumeration.
4. InterContractInvocationLog, which stores information about the invoker and invoked contracts, the arguments sent to the invoked contract and an InvocationResult object.
5. ContractNotificationLog, stores information about a consent request or notice message, the invocation ID it corresponds to and whether consent was granted, if notification type is consent. This log is generated by mcom-core, using MCom1.0.1 platform methods.

The log and notification objects are stored in Java maps that use the invocation ID as key. The consent and notice logs are stored in different maps so that they can be retrieved more efficiently, depending on which condition needs to be satisfied. The logs were initially stored in serialisable format in log files, but this was later removed because the files didn't have any particular use when an initial version of the sandbox was completed.

4.3.2 Satisfaction Index

Each bundle has a satisfaction index which is updated according to its satisfaction of policy conditions during invocations (see next section). The Satisfaction Index is a way of learning about bundle behaviour. Action can also be taken on bundles based on their reaching index threshold. Policies can receive an "upgrade", so that a frequently consented bundle does not need to seek consent anymore.

The satisfaction index is stored in a Java HashMap with the key being the bundle name.

4.4 Sandbox Monitoring

The sandbox monitors invocations broadly by going through the two kinds of invocation log objects (ContractInvocationLog and InterContractInvocationLog), checking that the required notifications exist in data structures containing notification logs.

4.4.1 ContractInvocationLog Monitor

The ContractInvocationLog is generated after the invocation has been completed, and it is forwarded to the monitor for checking. InterContractInvocationLogs will be generated if the contract invoked made invocations to other contracts. Also, ContractNotificationLogs, as mentioned earlier, would be generated by mcom-core if notifications were sent. Therefore, the Monitor's key task is to make sure that if InterContractInvocationLogs ICs exist for a ContractInvocationLog C, and if these ICs contain one of or the same arguments as in C, the bundle invoked satisfies its policy by making sure appropriate ContractNotificationLogs exist. Its sequence of execution can be described stepwise as follows:

1. Retrieve InterContractInvocationLogs ICs for ContractInvocationLog
2. Retrieve policy for bundle invoked
3. For each InterContractInvocationLog in ICs
    a. Check if private arguments were sent. If so, retrieve notification logs
    b. if consent is required, retrieve consent logs
    c. for log in consent logs
        i. if intended consent recipient address is address of the invoker in ContractInvocationLog and consent granted set to true, set consent satisfied to true.
    d. if notice is required, retrieve notice logs
    e. for log in notice logs
        i. if intended notice recipient addrss is address of the invoker in ContractInvocationLog, set notice satisfied to true.
    f. if all conditions required are satisfied, set policy satisfied to true for bundle
    g. else if both consent and notice are required, and only consent is satisfied, set partial satisfaction to true
    h. if policy satisfied, increment bundle satisfaction index by 1
    i. else if partial satisfaction true, increment bundle satisfaction index by 0.5
    j. else, decrement bundle satisfaction index by 1
4. end

4.4.2 InterContractInvocationLog Monitor

InterContractInvocationLogs are generated after an Inter contract invocation is completed and the result is returned to the invoker contract. The main function of this Monitor is to ensure that contracts that perform inter contract invocations and receive sensitive results satisfy their privacy policy relative to the contract invoked that returned said sensitive result. Its sequence of execution can be described stepwise as follows:

1. Retrieve invocation result from InterContractInvocationLog CI
2. Retrieve invoked contract address ic_address from CI,
3. if result is sensitive, retrieve bundle_results sent by that bundle
4. for bundle_result in bundle_results
    a. if bundle_result == result, set sensitive_information_shared flag to true
5. if sensitive_information_shared is true, retrieve notification logs
6. if consent is required, retrieve consent logs

```
7. for log in consent logs
    a. retrieve notification receiver address r_address from log
    b. if ic_address == r_address, and consent set to true, set consent satisfied to true.
8. If notice is required, retrieve notice logs
9. For log in notice logs
    a. Retrieve notification receiver address r_address from log
    b. If ic_address == r_address, set notice satisfied to true.
10. If all conditions satisfied, set policy satisfied to true
11. Else if both consent and notice are required, but only consent is satisfied, set partial satisfaction to true
12. If policy satisfied is true, increment bundle satisfaction index by 1
13. Else if partial satisfaction is true, increment bundle satisfaction index by 0.5
14. Else decrement bundle satisfaction index by 1
15. End
```

4.4.3 Policy Action Monitor

The trust index for each bundle is stored in a data structure and this thread uses this data to take action for the satisfaction or violation of a bundle's policy. Different thresholds can be set for violation and satisfaction. When a bundle reaches the violation threshold, the MCom1.0.1 instance is notified. A user of an MCom instance is given the option to undeploy the errant bundle during this notification.

When a bundle is undeployed, its bundle descriptor is deleted, and a message is sent to all registrars it was advertised to remove its bundle descriptor from their list of advertised bundles.

A policy upgrade involves setting consent for a bundle to false, if it is currently set to true.

4.5 Testing

Bundles created and used to test MCom initially were modified and adopted to test the sandbox. More details of this are in the next chapter.

4.6 Conclusion

The Sandbox was designed and implemented on MCom to the completion of requirements drawn to test the hypotheses developed in the first two chapters. The next chapter discusses how the sandbox was evaluated.

Chapter 5 Testing and Evaluation

The sandbox was evaluated in two ways. The first evaluation is concerned with performance of the sandbox for various actions. These include deployment and invocation completion time. The second features wider discussions about the sandbox in general, including problems with longer sequence of invocations and the use of the satisfaction index to dynamically manage data flow. As mentioned earlier, the satisfaction index of a bundle is incremented or decremented each time a bundle satisfies its policy or violates its policy.

Case studies are needed to extract relevant data. Four bundles were adopted, for testing that the sandbox works, and modified to use for both evaluations. Details of the bundles are as follows:

1. gbpBundle, which has a convertPound contract. The contract finds the Great Britain Pounds equivalent of a supplied currency argument. The bundle uses an external library (tunyk converver) to perform this function. The speed of the API was not calculated as part of the evaluation.
2. wBundle, has a getWeather contract that returns weather details for a Location name given as an argument. This bundle also uses a library that uses the OpenWeatherMap API.
3. tBundle, with a storeToList contract that does nothing but store its argument in a list.
4. gbpBundlebad, a lite version of gbpBundle that invokes gbpBundle. It was made to test policy violation and monitor response.

GbpBundle was modified to invoke wBundle and also return weather details, in order to test inter contract invocation. For some invocation runs, wBundle's weather contract was modified to send its results as private. With all four bundles, tests were done to confirm that the bundles are able to invoke other contracts, retrieve their policies and send notifications. Policy satisfaction, violation and upgrades were also tested.

Two test environments were used in the evaluation. The first is a Windows personal laptop with an Intel Core i3 processor and a processor speed of 2.53GHz. The other consists of two Linux workstations with core i5 processors and processor speed of 3.2GHz.

5.1 Performance

Here the performance of MCom1.0.1 with sandbox is compared to MCom1.0.0, which is without the sandbox.

5.1.1 Deployment

Since the deployment process was changed slightly, times taken to deploy a bundle in each version were recorded to calculate the significance of performance reduction. One bundle was deployed 19 times on both versions of MCom, and the completion times were recorded. On average, it took 2.6 seconds to complete deployment on the old version, while it took 3.2 seconds on 1.0.1. This represents an approximately 23% increase in deployment time.

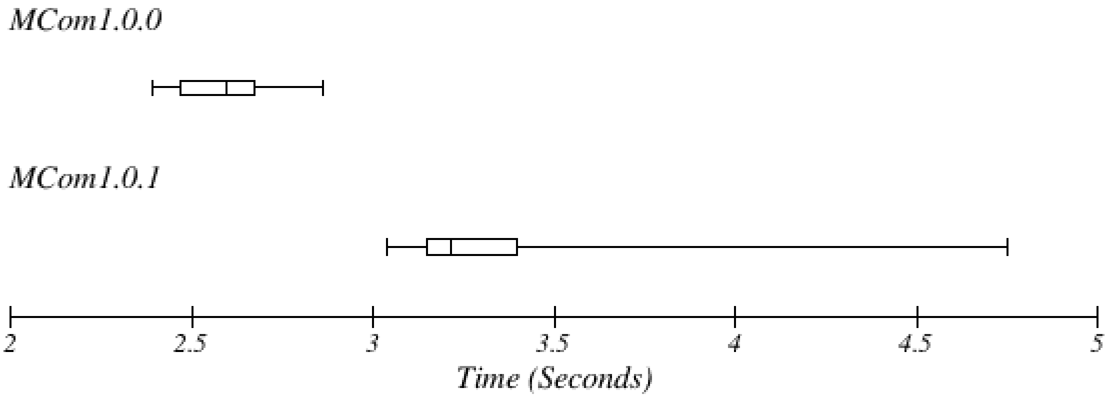


Figure 5.1: Deployment times boxplot

As depicted in the box and whisker plot above, the minimum of the deployment times for MCom1.0.1 was 3.04 seconds, while the maximum was 4.75 seconds.

The significant increase is as a result of the initial trust mechanism. When this was disabled, the deployment times were about the same. The use of the initial trust mechanism can be an option during deployment, instead of compulsory. However this change was not effected in the final sandbox submitted.

5.1.2 Normal Invocation

The level of impact the sandbox has on a normal invocation was also measured. Normal invocation refers to invocations involving no sharing of private data and no inter-contract invocations. In a similar fashion to the deployment test, 20 normal invocations were carried out on MCom1.0.0 and MCom1.0.1. On average, invocation on MCom1.0.0 took approximately 0.52 seconds. On MCom1.0.1, it took 0.58 seconds. This represents an approximately 11.5% increase in the time, which is relatively acceptable. The minimum time it took in MCom1.0.1 was 0.52 seconds, and at the maximum it took 0.72 seconds.

A box plot of the invocation times is shown below:

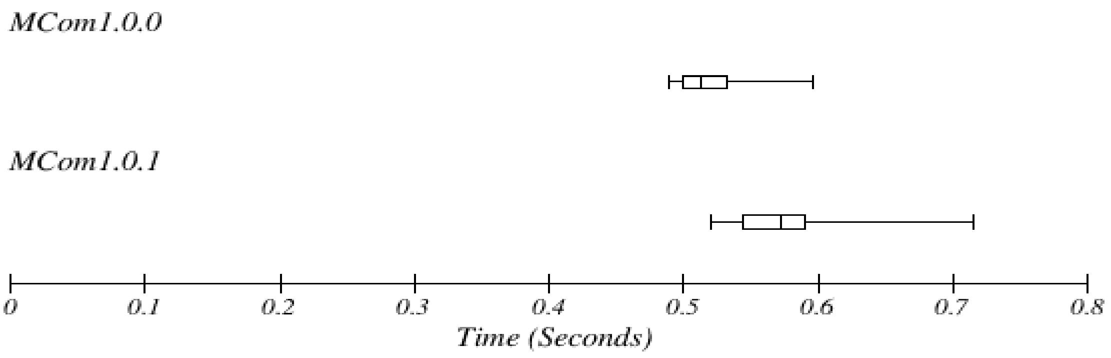


Figure 5.2: Normal invocation times boxplot

The reason that it took longer on 1.0.1 is that the invocation process was slightly altered. Firstly, it might prompt a user if they want to continue using an untrusted bundle, and wait for a response. Secondly, the bundle entity object for the requested contract that is used to execute the invocation request has relevant invocation meta variables that it inherits from MComEntity set as well. This is done reflectively. However, as mentioned earlier, the performance reduction is not perceived as significant.

5.1.3 Effects of Consent and Notice

By default, all bundles have both consent and notice set to true. Performing both however invariably leads to invocations taking more time; especially when one bundle performs multiple inter contract invocations with private arguments. The sandbox adapts to this through policy upgrades. An attempt was made to measure and estimate the effects of the combination of consent with notice.

The experiments were run on the Linux workstations. In this setting, the gbpbundle was made to invoke the wBundle and location was given as private. 20 invocations were performed, with the sandbox prompting for permission for gbpbundle to share the location with wbundle, and permission being granted. On average, the invocations where consent and notice had to be fulfilled took 1.1 seconds. The time taken to respond to the consent request contributes significantly to this average. When notice alone is required, the average time for invocation completion was 0.05 second. This represents a 95.4% increase in invocation completion.

The same experiment was also replicated, but for the consent and notice combination only, for different nodes on a local network. Surprisingly, invocations took less time remotely (0.9 seconds) than invocations on the same machine. Network transmission did not take a significant toll on invocation completion time. The fact that input was done on one machine, the one the invocation originated from, and actual execution done on the remote bundle host is a plausible explanation for this. Therefore, invocation can actually be faster remotely if the network link is fast. It is expected though that remote invocations of this sort would take significantly longer in less efficient networks than the lab environment. In this regard, the performance evaluation was not extensive.

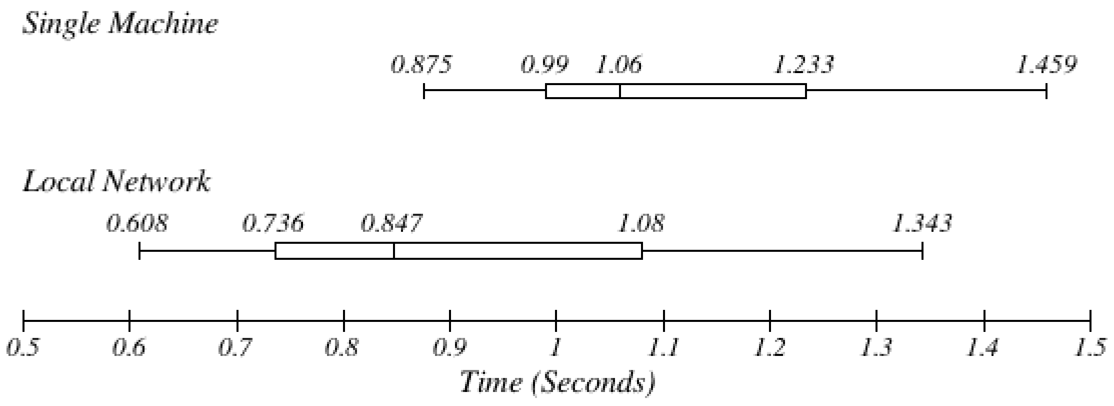


Figure 5.3: Box plots of invocation times with consent and notice

To test a scenario where both consent is needed to send a private argument, as well as return a private result from a remote bundle, another performance test was conducted on the personal Windows laptop where wbundle's contract returned its weather details as private. This in turn requires two consent requests granted. When consent and notice were required, it took on average 4.5 seconds to complete the invocations. The maximum was 7.6 seconds, although it can take longer depending on the time taken for the second consent request to be responded to. Notice alone took on average 0.13 seconds, an approximately 97% increase.

The use of consent is more in line with invokers that are not willing to take risks, even if that means an invocation is not completed, or takes longer to complete. A frequently consenting data owner might be a risk taker that's more interested in seeing invocations completed. On the other hand, a risk averter might deny consent on numerous occasions.

5.2 Longer Chains of Invocation

It is possible that in one invocation, private data is shared across up to four remotely located bundles, or even more. When an argument is given as private, and received by a bundle, the bundle is obligated to comply with policy as it is aware of the nature of the argument. But down a chain of invocations, it becomes more difficult to keep track of the private data flow and policy compliance, as bundles would not encode with the argument its private status or where the argument came from. This becomes a problem as other MCom instances down the chain would be unaware if bundles being invoked are complying with privacy policy.

A potential solution to this problem would be for policy requirements and data origin to be encoded with the arguments from a user in an invocation so that the data is heavier. MCom instances can keep track of these data flows and ensure that the argument metadata remains when a private argument leaves the bundle to another remote bundle. However this can become cumbersome as more data is being managed.

Another potential solution would be to prevent long chains of invocation and restrict contracts in bundles, or an entire bundle from being used if it initiates a long invocation chain, operating under the assumption that such invocations are malicious. This can be detected by initially tracing the line of invocation with the help of remote MCom instances receiving requests in a first invocation. But this might also cause unnecessary overhead and is perceived as rash.

While there's no perfect solution, the first potential solution seemed more reasonable. Therefore, the sandbox was modified such that private data is encoded when a bundle that receives it invokes another bundle, and its policy and the address of the data originator is also sent in a remote invocation message. In this sense the initial bundle's policy sticks to the data as it navigates through the invocation. The changes were tested to work in a chain of invocation involving three remotely located bundles involved in a chain of invocation, such that the data originator received a second consent request before invocations were completed (in this case gbpBundle -> wBundle -> tBundle). The changes were made such that extended monitoring is done when the invocation is between remote and collocated bundles. This certainly involved some changes to how the sandbox monitors worked, but the method fundamentally was still the same as described in the previous chapter.

Performance shifts weren't measured after the changes. However, the size of the XML encoded invocation message sent between remotely located bundles increased by about 141 bytes.

5.3 Satisfaction Index Evaluation

The satisfaction index of each bundle provides knowledge of the previous behaviour of the bundle relative to policy compliance. Therefore, it might be possible to model the risks of data compromise as user data flows between bundles in an inter contract invocation.

The use of the Satisfaction index can provide an even more fine-grained method to evaluate how trustable bundles are, and are a good alternative to the initial trust mechanism of bundle signage in an adaptive system. With the use of thresholds, either determined by an MCom instance based on the behaviour of a user, or directly by the user, an MCom instance might terminate an invocation if data is forwarded to a bundle that is below that threshold during an invocation. Such a threshold would more suitably be a negative index, as it denotes policy violation, although levels of trust might vary for users.

As an illustration, the Satisfaction Indices of the bundles involved with testing longer chains of invocation were read and plotted in the trend graph below. The origin node refers to the original user invocation, and since it isn't a bundle, it is starting from zero. Tbundle is at zero as it hadn't done any inter contract invocations and as such policy compliance checks hadn't been done on it. It can be an effective preventive way for future invocations that have high sensitivity.

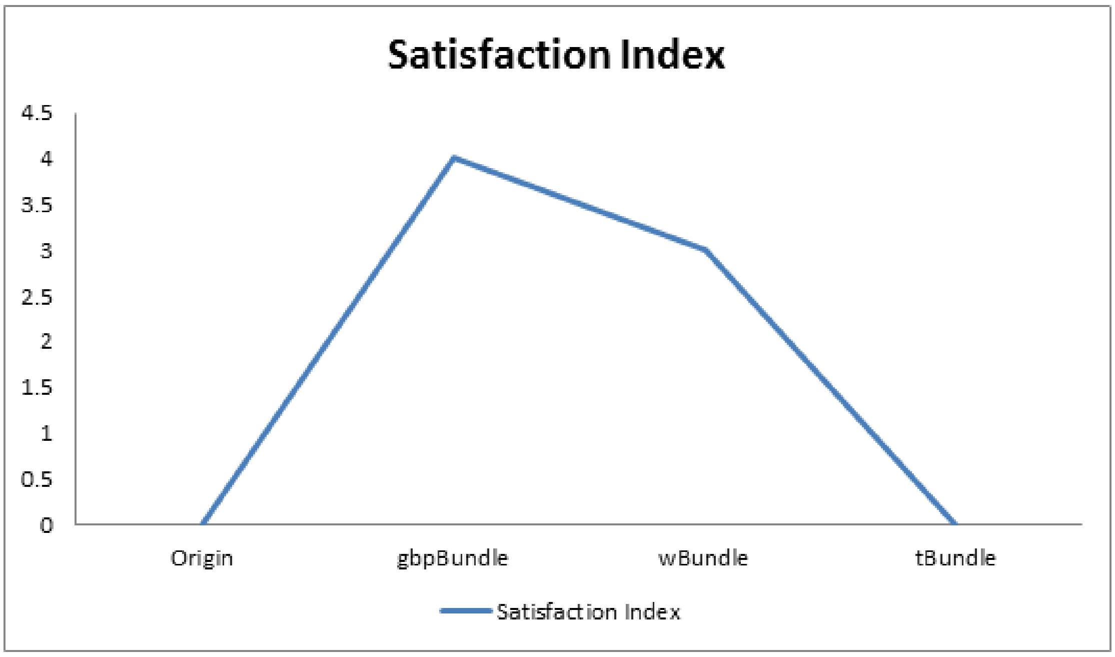


Figure 5.4: Satisfaction Index trend through the sample chain of invocation

5.4 Network Issues

The introduction of inter contract invocations made MCom a distributed CBS, which in turn potentially increased network functions when remote invocations request consent. The networked test environment was efficient, such that the effect of slower networks wasn't observed. Therefore, the sandbox isn't network resilient. The connection might for example time out before a response to a consent request is received, and while a requesting bundle can send more requests after a timeout, the mcom-core library would throw an exception instead.

5.5 Conclusion

The sandbox was implemented in a way that performance isn't greatly affected, in most cases at least. The use of the satisfaction indices to upgrade policies also significant improves performance in medium term. The idea of regulating privacy in Component Based Systems using this framework can lead to less compromise in the real world.

The sandbox is not perfect, but can act as a foundation for future work. One particular limitation of the sandbox is that it is corrective, instead of preventive; it only takes action after invocations occur. The potential use of the satisfaction index is preventive in the sense that it would prevent further compromises but this can happen only after the sandbox "learns from the behaviour of bundles".

Chapter 6 Conclusion and Future Work

The objective of this work from a bird's eye view was to explore the use of privacy policies in a Component Based System (CBS) setting to improve private data security as such data flows through the CBS. With the use of the MCom Component Based System, this was achieved, to an extent.

This has been done by building a Security Sandbox for the MCom CBS. Using privacy policies that were specified in a simple XML format, and based on simple transmission communication principles (consent and notice), the sandbox is able to monitor and regulate data flow within MCom components. Logs were relied upon to do this. The advantages of this method include the following:

- The sandbox is relatively efficient, because the rules it is monitoring compliance of are simple. Monitoring and taking potential action after invocation also helps performance.
- The provider of the private data plays a part in ensuring data security and the system understanding privacy of the data supplied.
- The policies are simple enough to understand and accommodate for potential developers of components to be used in a CBS.

Policy upgrades by the sandbox provide a way for the sandbox to be more efficient when it notices a trend of compliant behaviour by a bundle.

The fact that monitoring and action happen after invocation is also a limitation of the system, as it has the disadvantage of being corrective, rather than preventive. Policy regulation during invocation can be cumbersome. Its effects aren't part of this work however.

6.1 Limitations

While a few limitations were briefly highlighted in the previous chapter, some of the others include:

1. Previous Bugs in MCom: Rectifying issues in MCom1.0.0 while also working on the sandbox was a distraction from making the sandbox better, but was necessary as some were fundamental, e.g. invocation mechanism bugs. An example bug prevented MCom from accepting more than one argument during invocations.
2. Inability to gain feedback from more people. Usage of the system and MCom would require some knowledge about CBSes and how MCom works. This can be time consuming, and in combination with time constraints it was infeasible to hold extensive user evaluations.

6.2 Future Work

More work can be done both on the MCom Sandbox and in a larger scale, real world systems.

6.2.1 MCom

More work on the sandbox could focus on resilience and the specification and implementation of a more complex but robust privacy policy system for the sandbox. Work on resilience can center on the MCom being able to better handle communication failure in various scenarios. For example, failure to execute an invocation can occur if the user consent response granting the requesting bundle permission doesn't reach its destination and the connection times out. As mentioned earlier, MCom can attempt retrying the request.

Regarding the policy specification, more conditions could be built into the current specification, like satisfaction index of the bundle to be invoked being greater than equal to a threshold, or other conditions that a bundle would need to satisfy when using private data. The regulation can also be extended beyond sharing of data between components, but with other sources like remote cloud locations.

6.2.2 Real World Use: OSGi and Android

OSGi is a more popular Component Based System in use. The idea to develop a sandbox like this one for OSGi originated from the similarity it shares with MCom, both being in Java and components referred to as bundles. Bundles can already use each other's services, so the focus would be on creating a mechanism for policy specification and regulation. While there is a Distributed version of OSGi classified as an Apache subproject, it would be better to focus on the other more accepted and used OSGi versions, which are not distributed.

The idea for Android came from the TaintDroid sandbox explored in Chapter 2. TaintDroid already points a direction towards tracking the data, and Android already has its permission system which applications use along with End User Agreements. Like TaintDroid, a privacy policy based sandbox for Android would track sensitive data as it flows through the android system, up to when it is at the network interface, which is where it is potentially sent to malicious remote locations. The key activity would be to detect in User Agreements whether these data locations are stated by the application providers, and how trustworthy the locations are. At the very least, users can be prompted by the sandbox when an application is sending sensitive data on the internet, or this activity can be flagged and reported. The major problem with sandboxes and security systems for Android however is the limited resources on smartphones [11]. But as the hardware capabilities of smartphones continue to improve, this would become less of a problem.

References

[1] Lau, K. K., & Wang, Z. (2007). Software component models. *Software Engineering, IEEE Transactions on*, *33*(10), 709-724.

[2] Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., & Stefani, J. B. (2006). The fractal component model and its support in java. *Software-Practice and Experience*, *36*(11), 1257-1284.

[3] Coulson, G., Blair, G., Grace, P., Taiani, F., Joolia, A., Lee, K., ... & Sivaharan, T. (2008). A generic component model for building systems software. *ACM Transactions on Computer Systems (TOCS)*, *26*(1), 1.

[4] Gama, K., & Donsez, D. (2011, March). Applying dependability aspects on top of aspectized software layers. In *Proceedings of the tenth international conference on Aspect-oriented software development* (pp. 177-190). ACM.

[5] Szyperski, C., Bosch, J., & Weck, W. (1999, January). Component-oriented programming. In *Object-oriented technology ecoop'99 workshop reader* (pp. 184-192). Springer Berlin Heidelberg.

[6] Heineman, G. T., & Councill, W. T. (2001). Component-based software engineering: Putting the Pieces Together*, Addison-Westley.*

[7] OSGi Alliance. (2008). OSGi alliance—Technology. *Website http://www. osgi. org/Technology/WhatIsOSGi.*

[8] Inoue, H., & Forrest, S. (2005, June). Inferring Java Security Policies Through Dynamic Sandboxing. in *PLC* (pp. 151-157).

[9] Huang, C. C., Wang, P. C., & Hou, T. W. (2007, May). Advanced OSGi security layer. In *Advanced Information Networking and Applications Workshops, 2007, AINAW'07. 21st International Conference on* (Vol. 2, pp. 518-523). IEEE.

[10] Burguera, I., Zurutuza, U., & Nadjm-Tehrani, S. (2011, October). Crowdroid: behavior-based malware detection system for android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*(pp. 15-26). ACM.

[11] Enck, W., Gilbert, P., Han, S., Tendulkar, V., Chun, B. G., Cox, L. P., ... & Sheth, A. N. (2014). TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, *32*(2), 5.

[12] MWR InfoSecurity MWR InfoSecurity warns on hidden dangers of third party code in free apps. *Website https://www.mwrinfosecurity.com/media/press-releases/mwr-infosecurity-warns-on-hidden-dangers-of-third-party-code-in-free-apps/*

[13] Bläsing, T., Batyuk, L., Schmidt, A. D., Camtepe, S. A., & Albayrak, S. (2010, October). An android application sandbox system for suspicious software

detection. in *Malicious and unwanted software (MALWARE), 2010 5th international conference on* (pp. 55-62). IEEE.

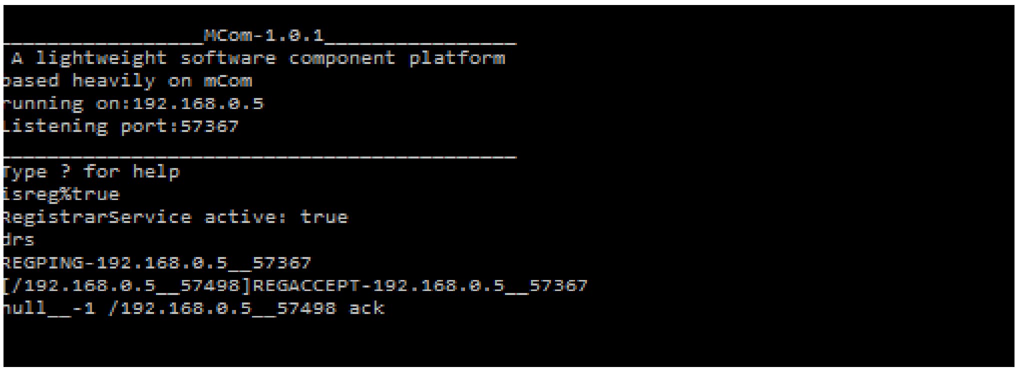
[14] Kumaraguru, P., Cranor, L., Lobo, J., & Calo, S. (2007, July). A survey of privacy policy languages. In *SOUPS'07: Proceedings of the 3rd Symposium on Usable Privacy and Security*.

[15] Anderson, A. (2005). A comparison of two privacy policy languages: EPAL and XACML.

[16] Bandara, A. K., Russo, A., & Lupu, E. C. (2007, June). Towards Learning Privacy Policies. In *Policies for Distributed Systems and Networks, 2007. POLICY'07. Eighth IEEE International Workshop on* (pp. 274-274). IEEE.

Appendix A MCom Screenshots

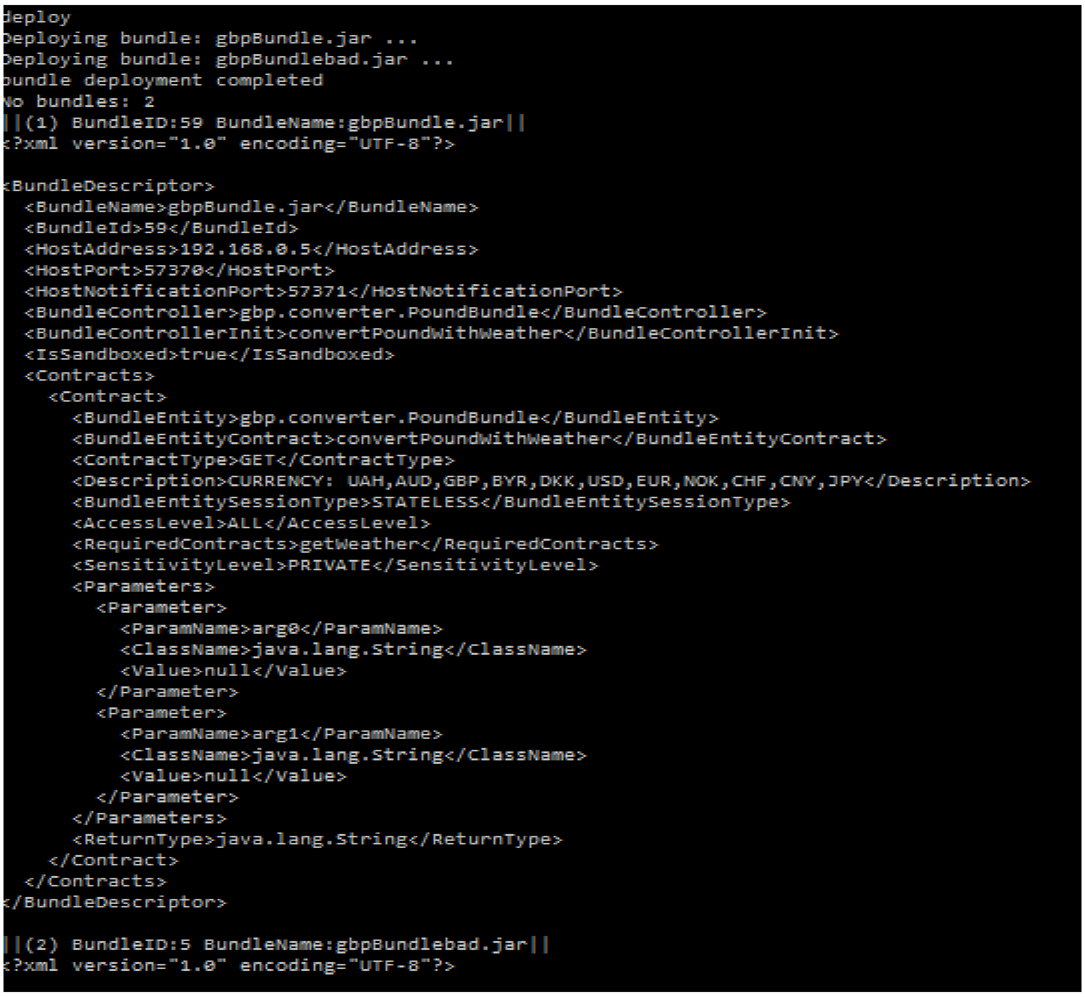
MCom is command line based and an instance can be started by typing the command “java –jar mcom.jar”.



```
________________MCom-1.0.1________________
 A lightweight software component platform
based heavily on mCom
running on:192.168.0.5
Listening port:57367
__________________________________________
Type ? for help
isreg%true
RegistrarService active: true
drs
REGPING-192.168.0.5__57367
[/192.168.0.5__57498]REGACCEPT-192.168.0.5__57367
null__-1 /192.168.0.5__57498 ack
```

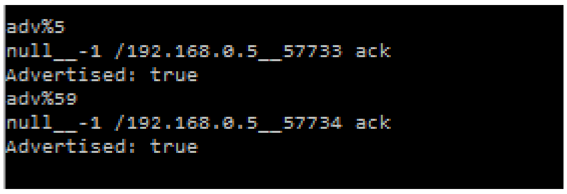
Figure A: MCom Start Screen.

The registry service command “isreg%true” makes the MCom instance a registrar, and can therefore receive adverts for bundles locally and remotely in the local network. The “drs” command discovers registry services located on the local network.



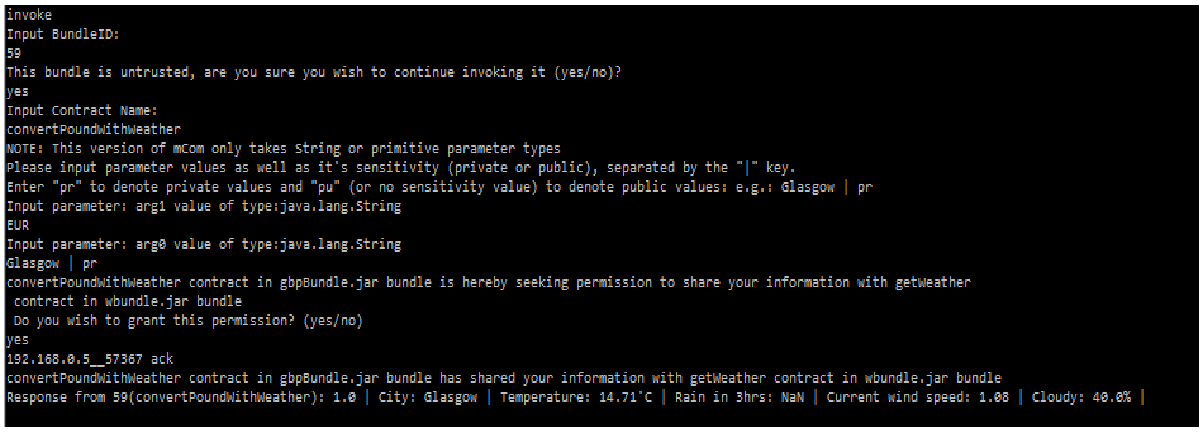
```
deploy
Deploying bundle: gbpBundle.jar ...
Deploying bundle: gbpBundlebad.jar ...
bundle deployment completed
No bundles: 2
||(1) BundleID:59 BundleName:gbpBundle.jar||
<?xml version="1.0" encoding="UTF-8"?>

<BundleDescriptor>
  <BundleName>gbpBundle.jar</BundleName>
  <BundleId>59</BundleId>
  <HostAddress>192.168.0.5</HostAddress>
  <HostPort>57370</HostPort>
  <HostNotificationPort>57371</HostNotificationPort>
  <BundleController>gbp.converter.PoundBundle</BundleController>
  <BundleControllerInit>convertPoundWithWeather</BundleControllerInit>
  <IsSandboxed>true</IsSandboxed>
  <Contracts>
    <Contract>
      <BundleEntity>gbp.converter.PoundBundle</BundleEntity>
      <BundleEntityContract>convertPoundWithWeather</BundleEntityContract>
      <ContractType>GET</ContractType>
      <Description>CURRENCY: UAH,AUD,GBP,BYR,DKK,USD,EUR,NOK,CHF,CNY,JPY</Description>
      <BundleEntitySessionType>STATELESS</BundleEntitySessionType>
      <AccessLevel>ALL</AccessLevel>
      <RequiredContracts>getWeather</RequiredContracts>
      <SensitivityLevel>PRIVATE</SensitivityLevel>
      <Parameters>
        <Parameter>
          <ParamName>arg0</ParamName>
          <ClassName>java.lang.String</ClassName>
          <Value>null</Value>
        </Parameter>
        <Parameter>
          <ParamName>arg1</ParamName>
          <ClassName>java.lang.String</ClassName>
          <Value>null</Value>
        </Parameter>
      </Parameters>
      <ReturnType>java.lang.String</ReturnType>
    </Contract>
  </Contracts>
</BundleDescriptor>

||(2) BundleID:5 BundleName:gbpBundlebad.jar||
<?xml version="1.0" encoding="UTF-8"?>
```

Figure B: Deploy Command with Resultant Bundle Descriptor



```
adv%5
null__-1 /192.168.0.5__57733 ack
Advertised: true
adv%59
null__-1 /192.168.0.5__57734 ack
Advertised: true
```

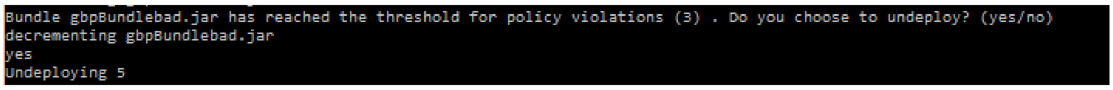
Figure C: Advertising Bundles



```
invoke
Input BundleID:
59
This bundle is untrusted, are you sure you wish to continue invoking it (yes/no)?
yes
Input Contract Name:
convertPoundWithWeather
NOTE: This version of mCom only takes String or primitive parameter types
Please input parameter values as well as it's sensitivity (private or public), separated by the "|" key.
Enter "pr" to denote private values and "pu" (or no sensitivity value) to denote public values: e.g.: Glasgow | pr
Input parameter: arg1 value of type:java.lang.String
EUR
Input parameter: arg0 value of type:java.lang.String
Glasgow | pr
convertPoundWithWeather contract in gbpBundle.jar bundle is hereby seeking permission to share your information with getWeather
 contract in wbundle.jar bundle
 Do you wish to grant this permission? (yes/no)
yes
192.168.0.5__57367 ack
convertPoundWithWeather contract in gbpBundle.jar bundle has shared your information with getWeather contract in wbundle.jar bundle
Response from 59(convertPoundWithWeather): 1.0 | City: Glasgow | Temperature: 14.71°C | Rain in 3hrs: NaN | Current wind speed: 1.08 | Cloudy: 40.0% |
```

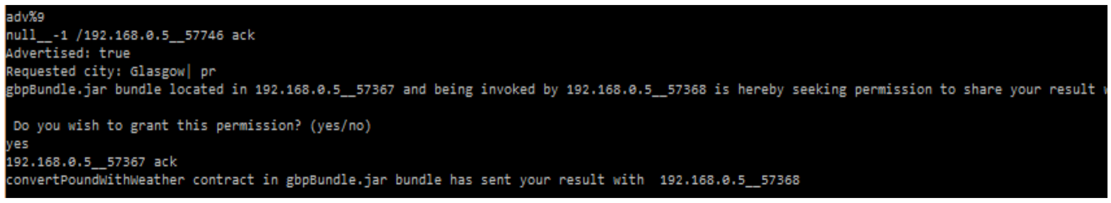
Figure D: Invocation with Policy Compliance

The rlookup command should be run before invoke.



```
Bundle gbpBundlebad.jar has reached the threshold for policy violations (3) . Do you choose to undeploy? (yes/no)
decrementing gbpBundlebad.jar
yes
Undeploying 5
```

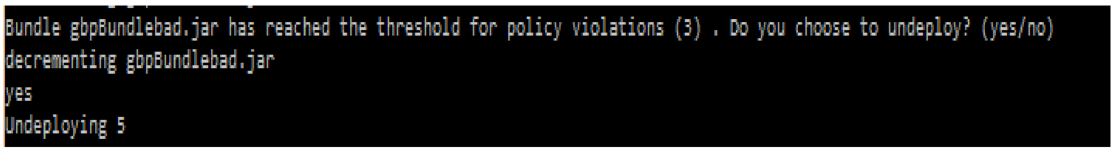
Here consent and notice are being satisfied as the location argument is tagged as private.



```
adv%9
null__-1 /192.168.0.5__57746 ack
Advertised: true
Requested city: Glasgow| pr
gbpBundle.jar bundle located in 192.168.0.5__57367 and being invoked by 192.168.0.5__57368 is hereby seeking permission to share your result

 Do you wish to grant this permission? (yes/no)
yes
192.168.0.5__57367 ack
convertPoundWithWeather contract in gbpBundle.jar bundle has sent your result with  192.168.0.5__57368
```

Figure E: Invocation with Policy Compliance

The bundle invoked is seeking consent the other bundle it invoked to disclose the sensitive result.



```
Bundle gbpBundlebad.jar has reached the threshold for policy violations (3) . Do you choose to undeploy? (yes/no)
decrementing gbpBundlebad.jar
yes
Undeploying 5
```

Figure F: Undeploy prompt, using 3 as a violation threshold

Appendix B MCom Annotations and MCom - Core Functionality Methods

B.1 MCom Annotations

@mController: This is an ElementType.TYPE annotation that is used to identify the class that controls the bundle. A bundle has can have at most one @mController annotation.

2. @mControllerInit: This is an ElementType.METHOD annotation used to identify a method that initializes parameters and calls methods required for the bundle to run appropriately during invocation. A @mControllerInit can only exist within an @mController class.

3. @mEntity: This is an ElementType.TYPE annotation that is used to identify the class that contains bundle contracts.

4. @mEntityContract: This is an ElementType.METHOD annotation that is used to identify contracts specified in a @mEntity. This annotation takes the description of the contract, the required contracts, the sensitivity level of its result and contractType as parameters.

B.2 MCom-Core Methods

Its core functionality methods are in an MComUtils class and they are as follows:

- getAvailableBundlesForContract(String contractName): This method returns an ArrayList of Bundle Descriptors in string format to the requesting bundle.
- getPolicy(String bundleName): This method returns the policy for a bundle as a policy object.
- invokeContract(…): This method invokes a contract in a bundle by reflectively calling the method in charge of inter contract invocations in the MCom platform's stub. The contract to invoke, bundle descriptor and the arguments in an array of strings are passed to it by the bundle.
- seekConsent(…): This method sends a consent message to the intended recipient, which can be an immediate or original invoker of a contract, depending on whether the bundle seeking consent is the initially invoked contract or whether it is further along in a chain of invocations. It returns a Boolean signifying whether consent was granted or not.
- seekInvokeeConsent(..): This is like seekConsent, but the consent message is sent to the host of a bundle that is being invoked by the requesting bundle.
- sendNotice(): This sends a notice message to intended recipient.
- sendInvokeeNotice(): This is like sendNotice, but the notice message is sent to the host of a bundle that is being invoked by the sending bundle.