Da Silva *et al. Journal of Internet Services and Applications* (2018) 9:19
https://doi.org/10.1186/s13174-018-0090-7

Journal of Internet Services and Applications

RESEARCH Open Access

Self-adaptive authorisation in OpenStack cloud platform

Carlos Eduardo Da Silva[1*], Thomás Diniz[2], Nelio Cacho[2] and Rogério de Lemos[3]

Abstract

Although major advances have been made in protection of cloud platforms against malicious attacks, little has been done regarding the protection of these platforms against insider threats. This paper looks into this challenge by introducing self-adaptation as a mechanism to handle insider threats in cloud platforms, and this will be demonstrated in the context of OpenStack. OpenStack is a popular cloud platform that relies on Keystone, its identity management component, for controlling access to its resources. The use of self-adaptation for handling insider threats has been motivated by the fact that self-adaptation has been shown to be quite effective in dealing with uncertainty in a wide range of applications. Insider threats have become a major cause for concern since legitimate, though malicious, users might have access, in case of theft, to a large amount of information. The key contribution of this paper is the definition of an architectural solution that incorporates self-adaptation into OpenStack Keystone in order to handle insider threats. For that, we have identified and analysed several insider threats scenarios in the context of the OpenStack cloud platform, and have developed a prototype that was used for experimenting and evaluating the impact of these scenarios upon the self-adaptive authorisation system for the cloud platforms.

Keywords: Access control, Cloud computing, Insider threats, OpenStack, Self-adaptive systems

1 Introduction

The use of cloud services has gained widespread adoption, and can now be found in a wide number of businesses, such as, companies, research centres, universities, etc. Cloud infrastructures can be deployed as a public, private, community or hybrid model [23]. This characteristic defines how data is distributed and the type of user (insider, external or both) that is able to access cloud resources. OpenStack is a software toolbox for building and managing cloud computing infrastructures for the provision of Infrastructure as a Service (IaaS) [23]. The software is open-source and offers services as storage (Swift), networking (Neutron) and processing (Nova). OpenStack also provides an identity management service, called Keystone, which is responsible for providing API client authentication, service discovery, and distributed multi-tenant authorisation by implementing OpenStack's Identity API [1]. In this context, the OpenStack platform has established itself as a widely adopted cloud solution.

Although there has been an increasing adoption of cloud computing systems, some aspects related to security and privacy are still in its infancy, such as, the handling of insider threats. Some efforts have been made for dealing with malicious attacks in cloud [14, 16, 30], but these have not considered insider threats. An insider threat can be understood as a user who has or had authorised access to an organisation's network, system, or data, and exceed or misuse that access in a manner that can negatively affect the confidentiality, integrity, or availability of the organisation's information or information systems [5]. These insider threats are different from those that are restricted to components of the cloud infrastructure, such as, malicious hypervisors and broker [13]. When an insider threat takes place, the damage to the organization can be catastrophic, sometimes resulting in severe financial losses [10]. A famous example of insider threat took place in July 2010, when an intelligence analyst of the US army had access and published more than 250,000 secret documents from the US Department of Defence. Apparently, the analyst had access to the system, since he was an authorised user, however, there were insufficient mechanisms to detect abuse. In this case, the abuse was related

*Correspondence: kaduardo@imd.ufrn.br
[1]Metropole Digital Institute, Federal University of Rio Grande do Norte (UFRN), Av. Senador Salgado Filho, S/N – 59.098-970 Natal, RN, Brazil
Full list of author information is available at the end of the article

Springer Open

© The Author(s). 2018 **Open Access** This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

to the downloading 250,000 documents in a short period of time.

In general, many organizations have several processes that rely on information systems and computer infrastructures. These systems rely on authorisation infrastructures for managing access control decisions, and human administrators for activities related to monitoring and auditing of malicious behaviour. Automated monitoring tools for authorisation systems are not able to detect malicious behaviour, as it looks for violations of the access control policies. On the other hand, a human system administrator could become aware that a high number of related requests in a short period of time might constitute inappropriate behaviour. However, a human system administrator is not able to monitor a large number of requests in the system in search for anomalous behaviour [2].

Self-adaptive systems have shown to be able to provide an appropriate solution to treat these problems due to their efficiency and effectiveness in dealing with uncertainty in a wide range of applications, including some related to user access control [2, 24, 27, 32]. Self-adaptive systems are able to modify their own structure or behaviour during run-time [21]. An example of such system would be the Self-adaptive Authorisation Framework (SAAF) [2]. This framework is capable of adapting at run-time authorisation policies using self-adaptation mechanisms. SAAF's objective is to monitor the usage of authorisation infrastructures, analysing subject interactions and adapt this infrastructure accordingly. SAAF was applied in the context of an authorisation system called PERMIS [7]. Since PERMIS has a particular architecture, SAAF design was specifically tailored to observe and control PERMIS. Thus, applying SAAF to OpenStack would require considerable refactoring because PERMIS and OpenStack are quite distinct in their architectures and how they enforce authorisation.

Based on the above, the contributions of this paper are threefold. First, we discuss the limitations of OpenStack platform for dealing with insider threats. Second, we propose an architectural solution that incorporates self-adaptation into OpenStack, in order to deal with insider threats at the user level. This architectural solution is defined in the context of OpenStack components that are responsible for dealing with authorisation issues based on the Role Based Access Control (RBAC) model [26]. Third, we have developed a prototype for experimenting and evaluating the efficacy and efficiency of self-adaptive authorisation mechanisms in dealing with insider threats in a cloud platform, like OpenStack. For the evaluation, we have identified several potential insider threats scenarios, and for dealing with these threats, we have define potential responses from a self-adaptive authorisation infrastructure that is integrated with OpenStack.

The rest of the paper is organised as follows. In the next section, we present the context of our work, basically, access control models, OpenStack, and insider threats. In Section 3, we describe the motivations to incorporate self-adaptive capabilities to the OpenStack access control mechanisms. Section 4 describes the proposed self-adaptive OpenStack architecture and its implementation. Section 5 describes some of the experiments performed on the prototype that has been developed. Section 6 presents some related work regarding self-adaptation of authorisation infrastructures. Finally, in Section 7, we discuss some of our achievements, and provide an indication of future work.

2 Background

In order to contextualise our proposal, this section presents some background concepts. We start by describing insider threats and access control, followed by a brief introduction on self-adaptive systems and its use for managing access control. We conclude this section by presenting the OpenStack cloud platform, identifying its main characteristics related to access control.

2.1 Insider Threats

An insider threat can be seen as a misuse of the system by authorized users [28]. A key element of this kind of threat is the internal user. CERT[2] [5] defines an internal user as an employee, former employee, contractor or business partner who has access to system data, company information or resources. In this way, to characterize an insider threat, this user needs to have intentions to abuse or take advantage negatively of the company's data, affecting the confidentiality, integrity and availability of their systems [5, 18]. In this paper, we look specifically into abuse, which can lead to theft, one of the three types of insider threat [5]. Alternatively, insider threat can be divided into two main groups [11]: intentional and unintentional. For CERT, only the first group is characterised as an insider threat [18]. However, some insider threats caused by an innocent user may have high potential damage as well, for example: inappropriate Internet use, which opens possibilities to virus and malware infection, exposition of the enterprise influencing in its reputation and future valuation. In this context, our approach considers both intentional and unintentional insiders.

Authentication [9], access control models, and authorisation infrastructures [7] provide critical security measures for enabling confidentiality, integrity, and availability to an organisation's resources. These rely on models, such as Role Based Access Control (RBAC) [26] and Attribute Based Access Control (ABAC) [19], in order to support large scale distributed systems. The RBAC/ABAC models are based

on the assignment of roles/attributes, respectively, to users, and on the definition of rules associating roles/attributes to permissions. For example, in federated identity environments [6], identity providers (IdP) authenticate and assign attributes to theirs users, while service providers employ authorisation infrastructures for checking whether a subject has a particular set of attributes, i.e., privileges, in order to access a resource.

An RBAC/ABAC based system relies on a set of components for protecting access to resources [19], as presented in Fig. 1. When a user requests an operation on a given object, this request is intercepted by the Policy Enforcement Point (PEP). The PEP protects the object, redirecting the request to the Policy Decision Point (PDP), which checks whether to allow or not the subject's access. This is done by evaluating a policy, i.e., the rules that define the permissions associated with a role. Extra information used for decision-making are obtained through the Policy Information Point (PIP). Once a decision is made by the PDP, the PEP enforces it, granting or denying access to the requested object. Policies are maintained by system administrators through the Policy Administration Point (PAP).

However, unless additional measures are put into place, malicious insiders can abuse these security measures. For instance, if a user can be authenticated, and has the required access rights, access to resources should be given, as traditional access control mechanisms in general are not able to monitor users behaviour to identify insider threat.

2.2 Self-adaptive Systems

Self-adaptive software systems are systems that are able to modify their behaviour and/or structure in response to changes that occur to the system itself, its environment, or even its goals [21]. An implementation model of self-adaptive systems is the MAPE-K framework [20], which defines a autonomic element with four stages: **M**onitor, **A**nalyse, **P**lan and **E**xecution. Those elements communicate with one another and exchange information through a **K**nowledge base, and interact with the Target System by means of *Probes* and *Effectors*. Those, presented in Fig. 2, implement a feedback control loop where the *Monitor* is responsible for obtaining, aggregating and filtering the target system status information. *Analysis* stage evaluates the data sent by Monitor in detail in order to detect the need for target system adaptation. Once detected the need to adapt, the *Plan* builds a sequence of steps with the goal of ensuring the adaptation of the target system. *Execute* is responsible for performing the steps defined by the Plan stage, effectively modifying the target system.

There are two approaches for adaptation: parametric or structural [1]. Parametric adaptation is able to change the parameters according to the context. In contrast, the structural adaptation is able to change of system components and their connections, i.e., if a component does not provide a feature, it is possible to replace with one that has. For this work, it is important to highlight the applications of these concepts to security of systems. For instance, self-protection is applicable when the system adapts itself to ensure its security by reconfiguring its architecture or redefining parameters, rules and permissions to avoid

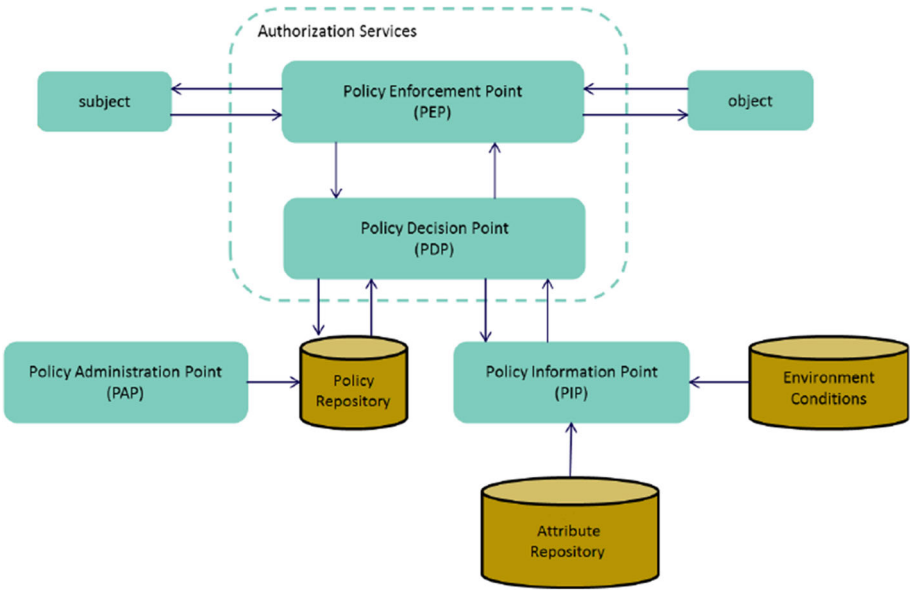

Fig. 1 ABAC Access control mechanisms

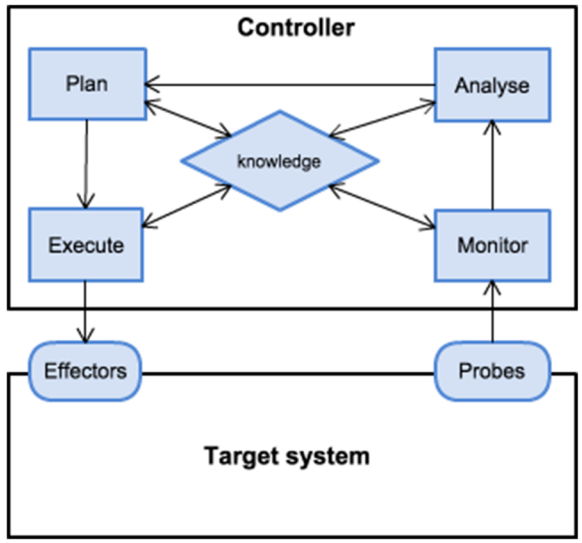

Fig. 2 MAPE-K Feedback loop

invasions or damages to the system caused by possible malicious users. In fact, self-protection has been identified as one of the essential traits of self-adaptation for autonomic computing systems [32].

2.3 Authentication and Authorisation in OpenStack KeyStone

The OpenStack platform is an open source software toolbox for building cloud infrastructures out of conventional hardware[3]. It is widely deployed around the world. One of the main features of OpenStack is its distributed architecture in which several software projects are used to provide different cloud services.

As depicted in Fig. 3, it is possible to have more than one service being provided, for example, a storage service (using Swift) and processing (using Nova), where the components that compose this infrastructure run in a distributed fashion. Neutron provides network services, while Keystone deals with identity management and access control. The general flow involves a user sending their credentials for authentication (step 1), and receiving a token (step 2) which is presented together with a request to the desired service (step 3). The service then checks the token validity with Keystone (step 4), and performs the request indicated by the user, replying with a response (step 5).

OpenStack employs the RBAC model for handling access control. A user in OpenStack is assigned a role associated with a tenant, which represents a cloud resource in one of the services provided by the OpenStack platform. The service can specify policies associating roles with permissions to conduct operations on the service, such as, permission to download a particular object from the storage service. OpenStack uses a token-based authentication mechanism, which requires the user to interact with Keystone for authentication from which it receives an access token. Using this token, the user makes requests to the cloud services, which performs two authorization procedures before allowing the user access. First, the service queries Keystone for checking if the token is valid. Once the token is validated, the user operations in the cloud service are performed according to its permissions.

Figure 4 presents a detailed view of the OpenStack architecture, where the components responsible for access control management are identified, together with the flow of operations performed by the system when a user tries to access a Swift service. Swift offers a cloud storage service so that OpenStack users can store and retrieve data with a simple API.

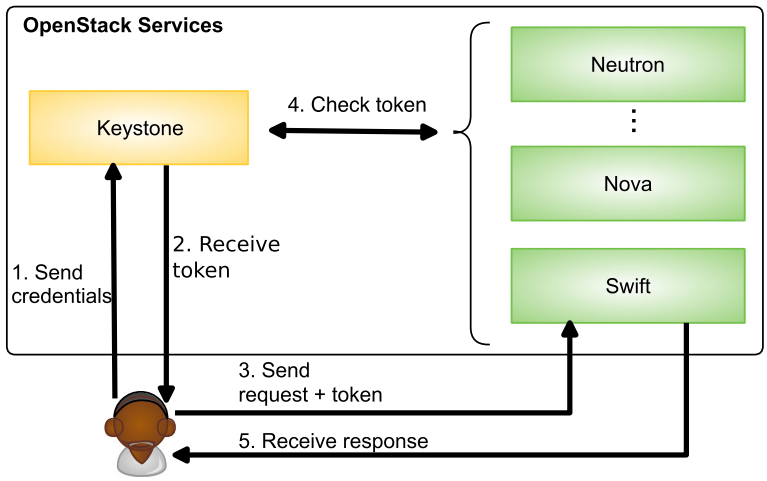

Fig. 3 OpenStack Keystone architecture

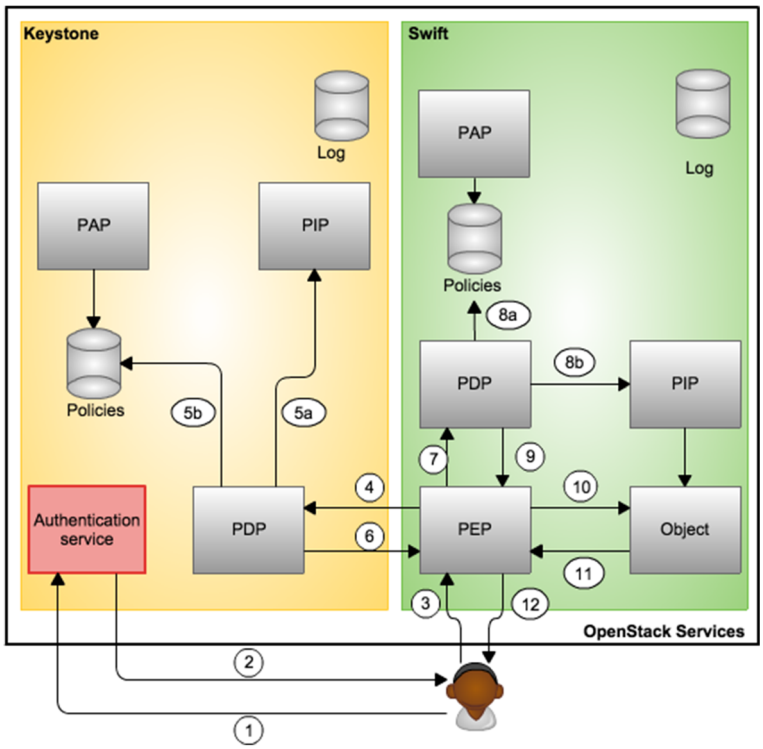

Fig. 4 RBAC components in OpenStack Keystone architecture

The flow begins with a user sending their credentials for performing authentication (1), and then receiving a token as reply of a successful authentication (2). The user then requests an operation from the cloud (3). Swift PEP intercepts this request, protecting the service from a possible unauthorised operation, and requests the PDP in Keystone (4) to validate the token and to check whether the user has access permissions to this service. After validating the token, the Keystone PDP consults the Keystone PIP (5a) and obtains its security policies (5b) for deciding whether the user has access to the Swift service, and returning its decision (6).

At this point, the first part of authorisation has finished, but OpenStack platform has an additional second authorisation step that is performed by the service. After consulting Keystone, Swift needs to evaluate the request against its own policies, for checking whether the user has permission to conduct the requested operation. The Swift PEP then activates Swift PDP to decide (7) whether the user can conduct the requested operation (e.g., upload a file). Swift PDP obtains the access control policies for the service (8a) and uses the Swift PIP (8b) to obtain any information that it needs for evaluating the access control policy. Once an access decision is granted (9), Swift PEP allows the user to perform the requested operation (represented by Step 10 towards the Object) and returns to the user a response to the request (Steps 11 and 12).

3 Problem Description

As previously mentioned, access decisions in OpenStack are computed at two different PDPs when processing a user request. This is the main characteristic of OpenStack authorisation mechanisms, and has prompted us to conduct an analysis of the access control mechanisms in OpenStack in order to identify how self-adaptive capabilities can be incorporated to deal with insider threats. In the following, we present an example of insider threat scenario, followed by a threat model analysis of OpenStack access control mechanisms.

3.1 An Example of Insider Threat

In order to illustrate the problem, we describe a hypothetical insider threat scenario related to theft of information. ACME is a hypothetical Information and Communication Technology company and has a private cloud based on OpenStack that uses the processing (Nova) and storage (Swift) services. Multiple users with different functions have access to the services offered by the cloud. The actions and privileges of each user vary according to the permissions associated with their roles. These roles are associated with users through the OpenStack Keystone, and the permissions set for each service according to their access control rules.

Alice has been working for some time as a consultant on several ACME projects, and she needs full access to

files stored on Swift, as well as multiple folders within it. This is possible because it is associated with a role of consultant. This role has full access to system files and folders. However, the consultancy is completed, but Alice continues with her enabled user in the cloud. Days later she ended up discovering that she still has access to the system and starts abusing the service, downloading indiscriminately current projects of the company. This scenario characterises Alice as a malicious user, as her behaviour of downloading a certain quantity of documents has changed to downloading a massive amount in a short time period, as she does not know for how long this access gap will be opened. As OpenStack is based on RBAC, it has the same limitations of traditional access control mechanisms, i.e., not being able to detect Alice's abuse. Thus, OpenStack requires a solution for mitigating insider threats.

3.2 Threat Model

Each OpenStack service contains a Log component, as shown in Fig. 4. This Log is used to record the different activities related to access control within the system. Among the information logged, we can mention access requests, access control decisions, operations performed in the service and unauthorised attempts.

OpenStack has a distributed and heterogeneous nature in which multiple services are protected by means of a two step token-based authorisation, which relies on the RBAC model. Hence OpenStack users can have access to different services with one or more distinct roles. This has prompted us to perform an analysis on different insider threat scenarios in order to identify possible limitations in applying current solutions [2, 8, 24, 29] to the OpenStack platform. For defining these scenarios, we have considered three variables that are directly related to insider threat: the number of users abusing the system, the number of roles involved in the attack, and the number of services being abused. These variables can assume two values, 1 or many (N). Based on this, we have defined a total of eight insider threat scenarios, which are listed in Table 1, ranging from the case where one user exploits one role for abusing one specific service (SCE#1), one user with several roles abusing one service (SCE#3), several users exploiting one role for abusing one service (SEC#5), to several users exploiting several roles and abusing several services (SEC#8).

Table 1 Summary of insider threat scenarios

Scenarios	Description
SCE#1	One user exploits one role for abusing one specific service
SCE#2	One user exploits one role for abusing several services
SCE#3	One user with several roles abuses one service
SCE#4	One user exploits several roles for abusing several services
SCE#5	Several users exploit one role for abusing one service
SCE#6	Several users exploit one role for abusing several services
SCE#7	Several users exploit one role for abusing one specific service
SCE#8	Several users exploit several roles for abusing several services

An analysis of the proposed scenarios (Table 1) against the OpenStack authorisation components (see Fig. 4) has demonstrated some liabilities of the OpenStack authentication/authorisation mechanism to deal with inside threats when compared with existing approaches.

3.3 OpenStack Distinctiveness

The distributed nature of OpenStack, and its ability to support multiple heterogeneous services, means that OpenStack components are arranged in a different way. For example, there are two Policy Decision Points (PDP), which require two sets of policies in the same system. This has implications on how OpenStack computes access decisions, and because of that self-adaptive solutions for dealing with authorisation, such as Self-Adaptive Authorisation Framework (SAAF) [2], cannot be directly used on OpenStack Keystone. Moreover, SAAF uses PERMIS [7] as authorisation system, which is different from OpenStack Keystone, and one of the differences is latter's reliance on a token-based mechanism. The differences between these two authorisation systems, and the fact that in a system based feedback control loop the target system (in this case the authorisation system) influences the design of the controller [4], implies that a novel architectural solution is required for supporting for self-adaptive authorisation in OpenStack, which will be the topic of the next section.

4 Proposed Approach

Figure 5 depicts our proposed self-adaptive OpenStack architecture (bottom) together with the flow of activities related to self-adaptation (top). The Controller in Fig. 5 represents the MAPE-K feedback loop that monitors the cloud platform, and performs adaptations when a malicious behaviour is detected. The target system is composed by different services provided by the OpenStack platform, including its identity service and access control.

In a self-adaptive system, the controller is usually tightly coupled with the target system since it should be able to reason and make decisions on when, what and how to adapt. For example, the Monitor and Execute stages of the control loop need to be modified according to the actual authorisation system being used. Such characteristic can be observed in other approaches supporting self-adaptation, such as the Rainbow framework [17]. Our proposed solution is tailored to OpenStack, and can be easily extended to consider other OpenStack components. For example, in order to apply our solution to OpenStack Nova, it is necessary to develop: sensors that are

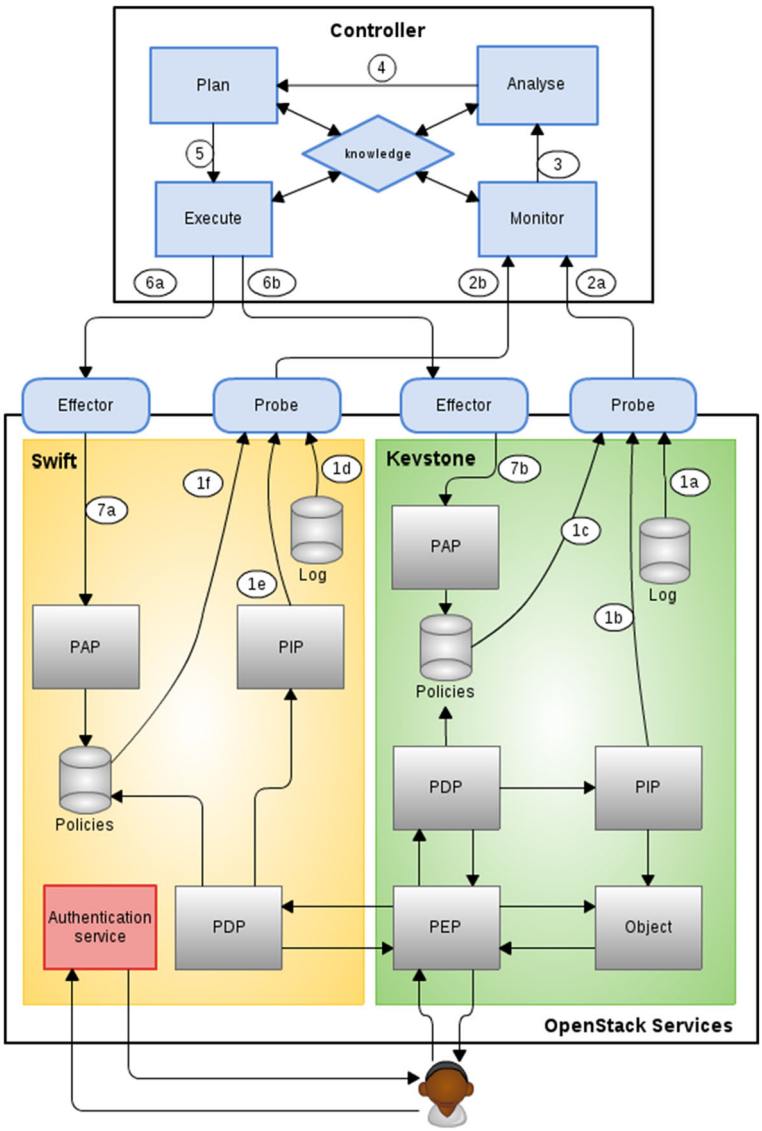

Fig. 5 Architecture of a Self-adaptive authorisation system for OpenStack

able to obtain logging information from Nova (the same technique used for OpenStack Swift can be employed), effectors that are able to interact with Nova API, and the rules and policies that manage the self-adaptation. Beyond that, our generic solution for the provision of self-adaptive authorisation can be tailored according to the application being deployed in an OpenStack platform.

Each OpenStack service has its own set of probes and effectors, which allows the Controller to interact with OpenStack. The information collected by the probes include the different activities related to access control that take place in the OpenStack platform, such as, access requests and authorisation decisions. As aforementioned, each OpenStack service contains a Log component that can be queried for this information (Steps 1a and 1d). There are also probes for obtaining the access control policies currently in place (Steps 1c and 1f), and information about users (Step 1e) and about objects being protected (Step 1b) by means of their respective PIP. This information is fed into the Monitor (Steps 2a and 2b), which is responsible for updating behavioural models in the Knowledge. The Analyse stage (3) is responsible for assessing the collected information in order to detect any malicious behaviours. This stage also identifies a set of possible adaptations for mitigating the perceived malicious behaviour, and prevent future occurrences. The Plan stage (Step 4) is responsible for deciding what to do and how to do it by selecting an appropriate adaptation (based on the set of possible adaptations provided by the previews stage) for dealing with the malicious behaviour, and

producing the respective adaptation plan. Finally, the Execute stage (5) adapts the authorisation infrastructure by means of effectors (Steps 6a and 6b), which alter the access control policies in place through the PAP of each service, i.e., Keystone PAP (Step 7a) and Swift PAP (Step 7b).

4.1 Responses to Insider Threats Scenarios

The applicability of the proposed architecture is exemplified by means of possible responses to the insider threats scenarios described in Section 3. Those responses are captured in Table 2, and incorporated into the MAPE-K controller of the proposed architecture. The responses can be executed either over Keystone, or over the service begin abused. Among the responses, we consider disabling a user (DU) or a role (DR) in Keystone, exchanging a user's role to another (ER), completely removing a role (RUR) or a tenant (DUT) from the user, and shutting down the service (TSO).

These responses may have different level of impact over the user, the role, or the service being accessed. It is possible that some of the responses may disrupt access to legitimate users whilst removing access to insider threats. Based on this, we have summarized in Table 3 these possible impacts.

Table 4 finally combines the information from the previous tables in order to present a complete picture of the identified insider threat scenarios, their possible responses over Keystone or the OpenStack service, and the impact of these responses to users, roles and services. The first column of that table identifies the scenario number. The next three columns describe the scenarios in terms of the number of users, roles and services that are involved in the scenario, which are summarised in Table 1. The following two columns identify the types of responses expected from a controller when handling the insider threat scenario. These responses are either associated with Keystone or the service, and they are summarised in Table 2. Finally, the last column identifies the types of impact that the scenario might have on the users, roles and services, and these are summarised on Table 3.

In order to illustrate our solution, we make use of the hypothetical example of an insider threat situation described in Section 3.1. With a controller based on MAPE-K, all download actions performed in the cloud were monitored. In Alice's case, the system detects that there was a high number of downloads in a short time, characterising abnormal behaviour. By identifying this abuse coming from a single user (Alice in this case), the system would characterise it in the scenario SCE #1. Once the insider threat scenario was detected, the possible

Table 2 Summary of actions in response to perceived insider threats

Acronym	Meaning
DU	Disable user
DR	Disable role
ER	Exchange user role to one with stricter permissions
RRA	Restrict role actions by modifying the permissions of a role regarding a service action
RUR	Remove user role by removing the role associated with the user in Keystone
DUT	Disassociate user's tenants by removing access to all tenants the user has access to
TSO	Turn the service off

Table 3 Summary of impacts

Impact	Description
IMP1	User does not have access permissions to the cloud
IMP2	Access permissions are revoked for all users associated with a particular role
IMP3	Role is disabled in the system
IMP4	With the new role access permissions for the user are restricted
IMP5	Service must be configured with new access permissions and restarted for deploying modifications
IMP6	User does not have access permissions to any resource in the cloud
IMP7	Service must identify which role is used to abuse, when the user is assigned several roles
IMP8	Service(s) will become unavailable

Table 4 Analysis of insider threats scenarios

Scenarios	Number of			Response		Impact
	Users	Roles	Services	Keystone	Service	
SCE#1	1	1	1	DU	-	IMP1
				DR	-	IMP2 and IMP3
				ER	-	IMP4
				-	RRA	IMP4 and IMP5
SCE#2	1	1	N	RUR	-	IMP2
				DUT	-	IMP6
				-	RRA	IMP4, IMP5 and IMP6
SCE#3	1	N	1	DU	-	IMP1
				ER	-	IMP4
				-	RRA	IMP4, IMP5, and IMP7
SCE#4	1	N	N	DU	-	IMP1
SCE#5	N	1	1	DR	-	IMP2 and IMP3
				-	RRA	IMP4, IMP5 and IMP6
SCE#6	N	1	N	DR	-	IMP2 and IMP3
SCE#7	N	N	1	-	TSO	IMP8
SCE#8	N	N	N	-	TSO	IMP8

responses would be to: disable the user (**DU**), disable the user role (**DR**), exchange the user role (**ER**) or restrict user action by modifying the role permission on the service.

Those alternative responses bring different impacts (as described in Table 3). For instance, in the first scenario (SCE#1), if the response is to disable the user in Keystone (DU), the user does not have access permissions to the cloud (IMP#1), while disabling a role (DR) impacts all users that are assigned that particular role (IMP#2), which might hinder the use of the role in the future (IMP#3). Although disabling a role might be an inappropriate response when dealing with scenario SCE#1, this response might be more efficient for scenario SCE#5, which considers the case in which several users, with the same role, abuse a service.

4.2 Implementation

In order to validate the proposed approach, we have implemented a prototype consisting of MAPE-K controller on top of OpenStack Keystone. Figure 6 presents a general view of our solution in terms of a UML diagram in which depicts the package structure. The prototype was developed using the Java language with specific libraries of the Jboss Drools [4].

The package `resource::rules` contains the files used by Drools. We have implemented probes for some of the OpenStack services. These probes observe the logs created by OpenStack Keystone, Swift and Nova. Each probe is composed by a main class (`App`) and two auxiliary classes. Class `LogFileTailer` monitors a log file, and notifies the `LogFileTailerListener` whenever a new entry is added to the log. This has been achieved by employing threads that listen to the log file in real-time. Probes captures information in a raw format, which is then standardized according to the model of `LogInfo` class. The controller package contains classes that implement the MAPE-K loop functionalities, i.e., Monitor, Analyse, Plan and Execute activities. Similar to probes, we have implemented an effector for each OpenStack service. These effectors use the REST API provided by OpenStack. Each response has been implemented as a parametrised script in order to allow the modification of users, roles or permissions involved in the attack.

Figure 7 describes the behaviour of our prototype from reading of logs by probes, until activation of the analysis module that will detect if it is necessary to adapt or not. A probe monitors a Log File. Whenever a log file is altered with a new entry (`getEntry`), the probe sends it to the `Monitor` component (`sendEntry`). The `Monitor` receives the raw data sent by the probes, does some processing, including converting the data into a format that can be manipulated by the other components of the MAPE-K, and stores it in the `Knowledge` component (`preprocessing`). The `Knowledge` component

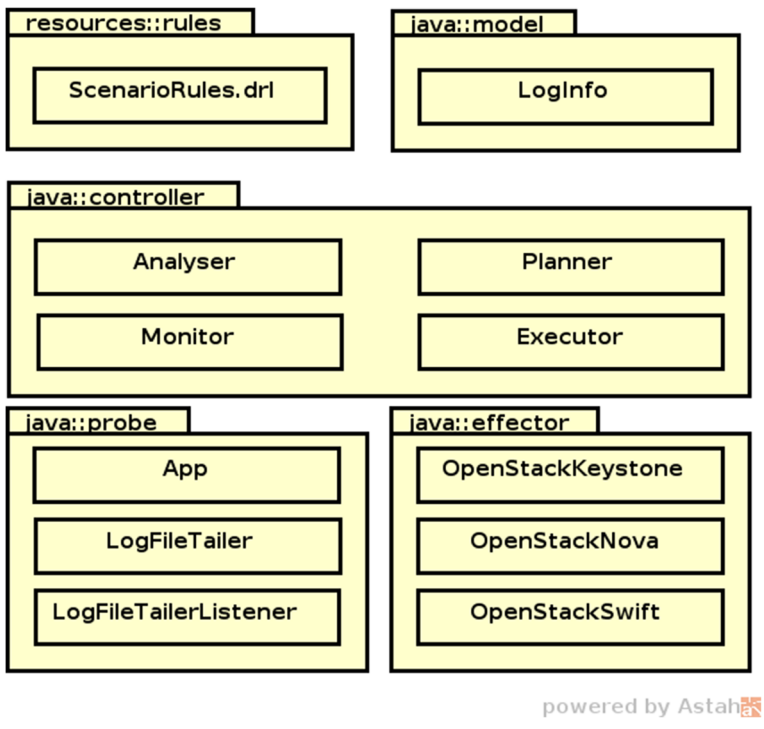

Fig. 6 Package Diagram of the Prototype

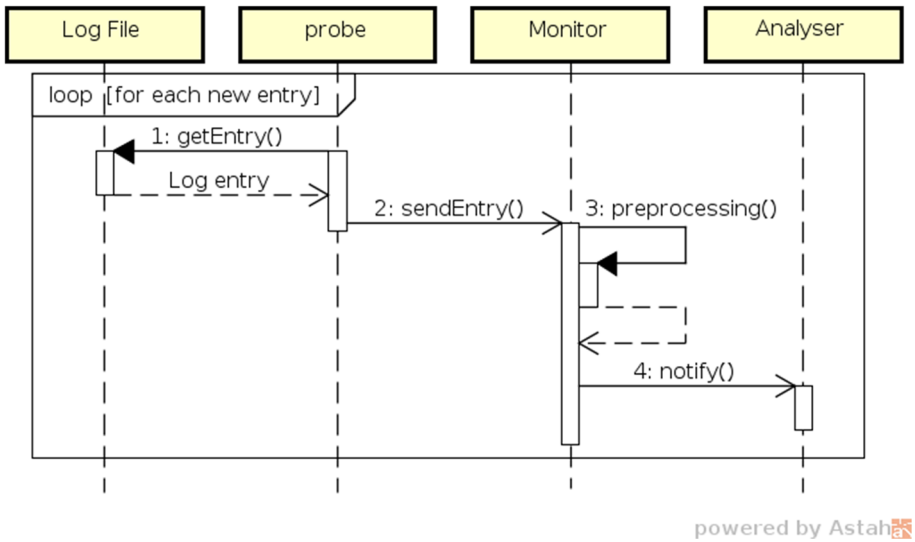

Fig. 7 Probe/Monitor sequence diagram

contains information regarding the action performed (e.g, upload, download, list, delete, etc), the user that performed the action, the tenant id, the user roles, a timestamp, etc. Finally, the `Monitor` notifies the `Analyser` component that a new entry has been saved. This loop is repeated for each new entry of the `Log File`.

Figure 8 describes the behaviour of our prototype once the analyses stage is started. The `Analyser` component employs the `Drools` rule engine for analysing the newly stored information against all registered rules. A Drools rule is a two-part structure using first order logic for reasoning over knowledge representation: *when* <*conditions*> *then* <*actions*>. In this way, we represent all the scenarios of Table 1 in terms of rules that, when triggered, capture the identification of insider threats scenarios. These rules consider the user, role and service accessed inside a pre-defined time interval and acceptable threshold. As an example, consider that an abuse is characterised by a high download rate with a threshold of 5 objects in less than one second. In order to detect such threat, we employ simple counters that are incremented whenever a rule (objects are downloaded by a user using the same role in less than one second) is triggered. If the counter reaches the defined threshold, then a threat has been detected. It is important to notice that a rule might be an indicator of more than one scenario. For example, the rule created for SCE#1 also contributes for detecting scenarios SCE#2, SCE#5 and SCE#6. `Drools` returns all activated scenarios to `Analyser`, which then notifies the `Planner` component informing

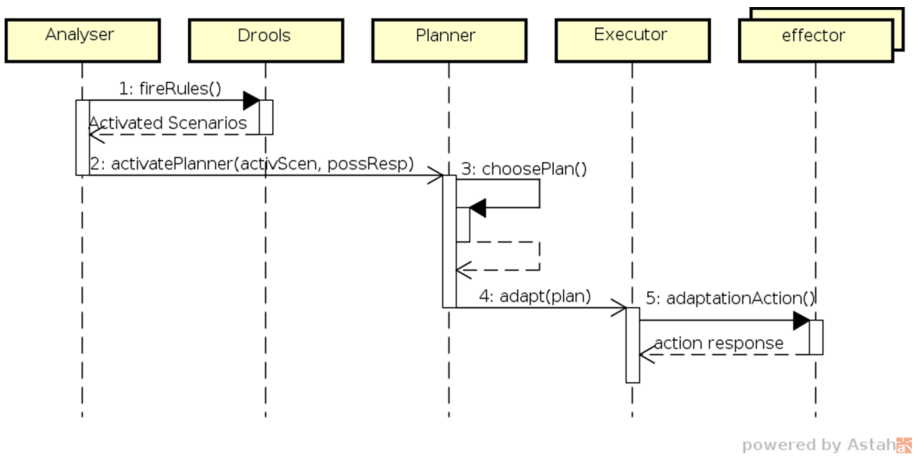

Fig. 8 Analyse, plan and execute workflow

the activated scenarios and all possible responses for mitigating threats in those scenarios (`activatePlanner (activScen, possResp)`). The `Planner` needs to make a decision about which response to employ among the ones available to deal with the respective scenario (`choosePlan`), and builds an adaptation plan that is sent to the `Executor`. `Executor` activates the effectors to perform the adaptation actions in their respective service (`adaptationAction`). It means that, if the response must be performed over `Keystone`, `Executor` will activate `Keystone Effector`.

5 Evaluation

As described in the previous section, the incorporation of self-adaptive authorisation into OpenStack comprises many steps. In this section, we evaluate our approach, present some results, and make some considerations about its behaviour. For that, we have deployed an OpenStack environment, together with MAPE-K controller (described in Section 5.1), for performing two sets of experiments with the objective of evaluating the feasibility and performance of our approach. The feasibility experiments (Section 5.2) focus on the validation of the adaptation rules, and for that we have considered a low number of users and requests. While the performance experiments (Section 5.3) consist of load test simulations involving a larger number of users and requests. The results of the experiments are presented in Section 5.4. Finally, Section 5.5 presents a brief discussion of the results.

5.1 Environment Description

The developed prototype has been applied in the monitoring and controlling OpenStack version Queens, which was deployed as a private cloud. Figure 9 presents the structure of the deployment of our experimental prototype, which is distributed amongst six nodes. Each node is a virtual machine with 4GB RAM, 2 VCPUs and 100GB disk. Two nodes are dedicated to storage (Storage Nodes), two nodes dedicated to Processing Nodes, and one acting as the

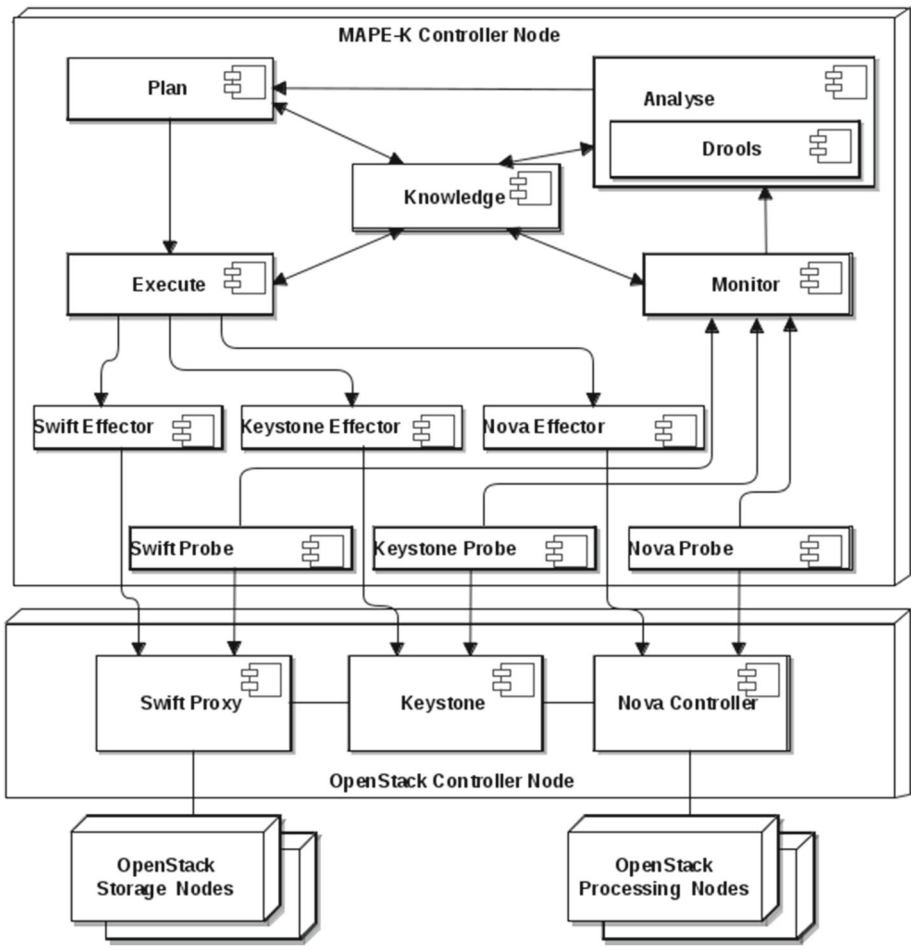

Fig. 9 Architecture of our experimental deployment

OpenStack Controller Node. The OpenStack Controller Node contains the following OpenStack components: Swift Proxy, Nova Controller and Keystone. Swift Proxy is the component responsible for managing access to the storage service, while Nova Controller takes care of virtual machine management, and Keystone deals with identity management. The MAPE-K controller is hosted in the "MAPE-K Controller Node".

Additionally, we have an extra node acting as a client of the OpenStack cloud (not shown in the diagram of Fig. 9). Each node of our prototype was then integrated into Zabbix[5] for monitoring CPU and memory consumption.

5.2 Feasibility Experiments

The feasibility experiments have the objective of verifying if we could identify the insider threat scenarios presented in Section 3.2, thus validating our analysis rules. These experiments have also been used to evaluate the respective response and impact for each identified scenario.

The feasibility experiments were partitioned between a private cloud (as represented in Fig. 9) and a simulated environment. The simulated environment consisted of a MAPE-K controller, whose inputs are from synthesized logs containing entries representative of a real environment, corresponding to a user performing cloud operations. This approach has been adopted for allowing us to verify the detection rules under a controlled environment. In both situations, we have been able to detect the eight insider threat scenarios, previously identified.

In order to evaluate the impact caused by each response, we have configured the controller to respond to each scenario in a pre-defined manner. For example, starting with SCE#1, where *one user exploits one role for abusing one specific service*, we initially configured the response of the controller to disable the user. After, we observe which impact on Table 3 is ensued. We repeated this process for each response of each scenario, which have confirmed our analysis of insider threat scenarios, their possible responses and respective impacts. Once the rules have been experimented and confirmed, we carried out further experiments with a larger number of user loads and requests.

5.3 Performance Experiments

For the performance experiments in our private OpenStack cloud, we have considered a population of up to 1000 users, which is a reasonable number considering other similar works. For instance, Feng et al [15] used *Symantec Box cloud* file-sharing access log data to detect insider threats in a population of 688 users, and Yaseen et al [25] used up to 500 users to assess the performance of models to detect insider threats.

Those 1000 users were created on Keystone, and assigned the same role on a single tenant, such that all 1000 users have the same access permissions. Additionally, each user has been assigned an individual role and tenant, to characterize its own "private" area on the cloud.

The service considered for the performance experiments was the OpenStack Swift, since Swift presents a low response time for confirming the results of an operation when compared with Nova, for example.

In order to generate the request load, we have used JMeter 1.0 [6]. It was configured to import a *csv* file containing the credentials of 1000 users. JMeter was configured to perform two HTTP requests. The first one is an authentication request, where JMeter uses all user entries in the *csv* file to acquire the access token and saves them internally in a variable for each user. The second HTTP request attaches the token and performs a downloading a file on the cloud. If the request is successful, it will return the status 200, otherwise it will return an error status. Based on this, the experiments consisted in generating request loads that could violate or not a pre-established threshold according with the scenario being considered. In these experiments, it was established that if there were more than 20 downloads from the cloud in less than 5 seconds, this would be considered as abnormal behaviour.

5.4 Results

In these experiments, two sets of metrics were captured. The first set of metrics is related to the time necessary for detecting and reacting to an abnormal behaviour. The second set of metrics is related to resource consumption of the MAPE-K controller node and the OpenStack controller node.

Regarding the time necessary for detecting and reacting, we have considered a population of users ranging from 10 up to 1000 in the following steps: 10, 50, 100, 500, and 1000. Depending on the scenario being evaluated, we have defined the number of malicious users on the population.

The detection time consists on the interval of time between the instant in which a malicious user starts an operation (corresponding to the instant in which we start the JMeter with the abnormal behaviour), and the instant in which our controller detects the anomalous behaviour. The reaction time considers the time elapsed between the instant of detection and the instant in which the controller receives a HTTP 200 status from the cloud API, indicating that the actions to mitigate the abnormal behaviour were completed with success. From this moment on, all requests made by a malicious user by JMeter receives an error status indicating access denied, and the experiment was finished. Following normal practice [12, 31], each performance experiment was repeated ten times, and the results presented correspond to their average, and respective standard deviation.

Table 5 presents the results for the experiments with scenario #1, which involves 1 user exploiting 1 role for

Table 5 Obtained Times for the scenario #1

#Users	Detection Time		Reaction Time	
(Common / Malicious)	avg (secs)	std dev	avg (secs)	std dev
10 / 1	2.775	0.258	1.321	0.131
50 / 1	3.617	0.966	1.291	0.098
100 / 1	5.050	0.423	1.449	0.250
500 / 1	52.007	4.640	1.464	0.034
1000 / 1	68.340	3.530	6.198	1.064

abusing 1 service. In this case, while the total population varied according with the steps already mentioned, only one user was abusing the cloud. We noticed that it takes around five seconds to detect one malicious user when the total population is of 100 users. However, there is a substantial increase on the detection time when the population jumps to 100 and 500 users. This happens due to the nature of our detection rules, which examines each line on the produced log whilst looking for abusers.

On the other hand, the reaction time revolves around one second, which is the time necessary for obtaining a valid admin-enabled token from Keystone, and performing the request to deactivate the offending user.

Table 6 presents the results obtained when considering scenario #5, where N users exploit 1 role for abusing 1 service. In this case, we have kept fixed the number of common users to 100, whilst varying the number of abusing users, according to existing similar experiments [25]. We noticed a detection time between 6 and 7 seconds for different numbers of malicious users. This happens because the way our detection rules have been defined. For detecting scenario #5, we look at the number of requests with a particular role made to a particular service. Thus, once a sufficient number of requests is made, our system is able to detect an exploitation of scenario #5. For example, considering that 10 abusing users is enough for detecting an abuse, any number of users above 10 will trigger the detection rule with the 10th user's request.

Regarding reaction time, the experiments with scenario #5 presented a result between 1.4 seconds for 10 and 20 malicious users, which correspond to a total population of 110 and 120 users respectively. This is consistent with the results obtained in our previous experiment. With a total population of 150 users (100 common and 50 malicious) the reaction time is around 4.8 seconds, while a total population of 175 users (75 malicious users) presents a reaction time of around 9 seconds.

Table 6 Obtained Times for the scenario #5

#Users	Detection Time		Reaction Time	
(Common / Malicious)	avg (s.ms)	std dev	avg (s.ms)	std dev
100 / 10	6.257	0.343	1.416	0.168
100 / 20	5.955	0.274	1.486	0.045
100 / 50	6.342	0.430	4.823	0.036
100 / 75	7.052	0.507	9.095	0.680

The difference between experiments with scenario #1 and scenario #5 is that, in the latter, after detecting a violation, the MAPE-K controller needs to identify the role employed in the abuse, which is then used in a request to OpenStack Keystone in order to obtain its identification. Finally, another request is made to disable the offending role. This happens because the code responsible for detecting a violation, and the code responsible for reacting to it, run in the same node.

During these experiments, we have collected the resource consumption of the different nodes of our infrastructure. In particular, we focused on the OpenStack Controller Node, and on the node that runs our MAPE-K controller.

The CPU consumption on the OpenStack Controller Node never went above 25% when the total number of users was equal or below to 100. During the experiments with 500 and 1000 users, the CPU consumption reached 100% whenever the JMeter was generating load, with a constant use of memory of around 2.0 GB throughout the full experiment. On the other hand, the MAPE-K controller node presented a maximum CPU utilization of 25% during the experiments with 500 and 1000 users. Regarding memory consumption, we observed that the MAPE-K node jumped from 2.0 GB of used memory to around 3.3 GB whenever a request load was being generated by JMeter. This behaviour happened with all number of users (from 10 to 1000).

These results are consistent with what we expected. Our MAPE-K controller does not impose any changes to OpenStack components, and does not interfere with the normal operation of OpenStack flow. However, the load on the OpenStack Controller Node was expected given the number of requests being made during the experiments. The detection of an abuse is made by the MAPE-K control loop, not an OpenStack component, which in turn needs to process each line of log produced by OpenStack, resulting on more resource consumption. Both aspects are further elaborated on the next section.

5.5 Discussion

Our experiments with synthesized logs have demonstrated that we are able to detect all the identified insider threat scenarios described in Table 1. However, smarter analysis techniques are needed to associate detected insider threats with decisions regarding what response to apply and when to apply it. Our detection rules considered that some scenarios are built based on others. For example, SCE#7 can be seen as an evolution of scenarios SCE#1, SCE#3 or SCE#5, which can lead to a kind

of progressive blocking situation, where n applications of response A might escalate to response B. Although, we reached the expected response, a malicious user might perceive the "progressing blocking" and change its attack strategy.

Although we simplified the decision making process associated with the selection of a response, we have confirmed the impact caused by each response. These provide valuable insight on some of the criteria and trade-offs that must be considered in this decision making, even for those responses that might look too excessive. For example, scenarios SCE#7 and SCE#8 characterise a massive abuse on the cloud platform, and may justify turning the affected services off, while further investigation is conducted for determining the root cause and deciding in a more appropriate response. The identified scenarios, and respective responses, cover the main situations considering users, roles and services, and by no means try to be exhaustive.

As our approach does not impact OpenStack operations, legitimate users will not experience any overhead as a result of our solution. However, users may still experience some overhead of an overloaded OpenStack caused by too many legitimate users. This situation might be representative of the fact that the supporting infrastructure is not properly dimensioned, which would fall outside of the scope of our solution. Regarding the proposed solution with respect to a larger number of users, the effectiveness and efficiency should be affected because more checks need to be performed during run-time, and these should affect the performance when mitigating an insider attack. Nevertheless, our experiments have shown that our solution based on self-adaptation is scalable since the response time has increased linearly with the number of users. Equally, since we employed a simple log analysis looking at each line of log, it is expected that the memory consumption to increase. Optimized log handling solutions, such as the Elasticsearch[7] software, could be applied to improve performance on log processing.

The self-adaptation of authorisation policies in the context of OpenStack Keystone remains a challenge. Our prototype has employed the Drools tool, a generic rules engine in which threshold based rules can be easily defined, thus facilitating the incorporation of new authorisation policies. However, a sophisticated solution for the self-adaptation of authorisation policies in the context of OpenStack Keystone is outside the scope of this paper. The problem is not restricted to the synthesis of the policies, these need to be verified before being deployed. An example of a solution that can support the self-adaptation of policies can be found in [3]. The synthesis of new policies should be based on the state of the authorization system, the evolution of that state, and the detected insider threats. Since the current solution relies on profiling the behavior of subjects, and the usage of resources, history is an integral part in the synthesis of detectors. How far in the past should the solution rely, it depends on the type of detectors that are needed. For example, for detecting slow attacks, the detectors should rely on large windows of observability depending on the specified attack model.

6 Related Work

Although there are several contributions regarding security in the cloud, little has been done regarding the application of self-adaptation to solve security problems, and in particular, looking into solutions for handling insider threats. An overall view of insider threats and its categories in cloud environments is presented in [13]. Although it elucidates the profile of insider attackers, whether it is a human or a bot, it does not provide any insight on how to deal with this type of threat. In terms of specific application domains, concerns have been raised of malicious insiders attackers towards healthcare systems [16]. Although the solution proposed prevents insiders from modifying medical data by using a combination of cryptographic techniques and watermarking, it is not sufficient for protecting the system from information theft if someone has legitimacy to access a resource. Another solution proposes the use of disinformation against malicious insiders by preventing them from distinguishing the real sensitive customer data from fake worthless data [30]. However, if an attacker knows precisely what he/she is looking for, disinformation might not hinder theft of information. On the other hand, sample data sniffers have a great potential in mitigating attacks during virtual machines (VM) migrations [14] since once a VM is reallocated to different hypervisor, a malicious attacker could exploit the vulnerabilities and obtain a large amount of data. From the above, and to the best of our knowledge, it is clear that no similar attempts have been made in using self-adaptation techniques in order to deal with uncertainties related to insider threats in cloud computing.

A similar approach to ours is the Self-Adaptive Authorisation Framework (SAAF) [2], which also adapts authorisation policies during run-time. A major restriction of SAAF is that it is implemented around PERMIS [7]. This dependence reduces its applicability and scope in way that it cannot be applied to cloud computing in general. For example, in an OpenStack Keystone context since SAAF is tailored very specifically towards PERMIS. Since the authorisation flow in OpenStack Keystone is quite different from that of PERMIS, it would not be simple task to refactor SAAF controller to OpenStack Keystone context. Hence the approach being proposed that is target to a particular cloud environment.

Another form of self-protection in access control is SecuriTAS [24], a tool that enables dynamic decisions in awarding access, based on a perceived state of the system

and its environment. A differentiating aspect of this work compare to ours that targets cloud computing is that SecuriTAS is aimed essentially towards physical security. SecuriTAS may change the conditions for accessing an office, for example, based on the presence of high cost resources, or the presence of highly authorised staff. This is achieved through an autonomic controller that updates and analyses a set of models (that define system objectives and vulnerabilities, threats to the system, and importance of resources in terms of a cost value) at run-time.

In the area of access control, there are some approaches [8, 29] based on the concept of Risk Adaptive Access Control [22], in which access control decisions takes into account an estimated risk for granting or denying access to resources. These works focused on methods for calculating risk based on historic information [29] or defining different levels of risk threshold for decision making [8]. However, these approaches require that the access control policies include risk related information that can be used for access control decisions. Different from these approaches, our work is focused on providing the means for adapting access control policies in response to detected abuse.

In our approach, we have decided to build an autonomic controller from scratch instead of using an existing one, like Rainbow [27] because the adaptation of authorisation policies is parametric rather than structural [1]. When adapting authorisation policies, which can be considered as parameters in system configuration, there is no need to deal with the structural representation of the system, which is a key aspect of Rainbow.

7 Conclusions

This paper has presented an architectural solution for handling insider threats in cloud platforms. Our solution incorporates self-adaptation into OpenStack access control mechanisms. In order to achieve this, a first step in the integration of an autonomic controller into OpenStack was to identify how the OpenStack authorisation components implement the role based access control (RBAC) model. A fully working prototype was built, and several scenarios representative of insider threats were identified, together with their possible responses and respective impact.

These scenarios were used to experiment and evaluate the impact of self-adaptive authorisation approaches into cloud platforms. We have observed how an autonomic controller handles insider threats, and the time it takes from detection to its response. From the results obtained, we have confirmed the potential, in terms of effectiveness and efficiency, that self-adaptation can provide in mitigating and protecting cloud platforms against insider threats. Starting from the fact that self-adaptive authorisation infrastructures are able to dynamically react to insider threats by adapting during run-time its access control policies [3].

However, several challenges lie ahead for obtaining more comprehensive solutions. One of these challenges is related to detecting and handling a wider range of scenarios representative of insider threats. We have employed simple rules for detecting such situations, which limits the scope of action of the controller. Moreover, it is necessary to obtain meaningful behavioural patterns from several distributed logs that are not semantically synchronised. This is quite relevant when dealing with insider threats since individually the logs provided by OpenStack services would not be sufficient to detect insider threats. Another challenge is related to the need of dealing with uncertainty originated from different and disparate sources of information. Regarding handling of insider threat, the challenge is related to the ability to generate, during runtime the appropriate policies to fight previously unknown insider threats.

Another interesting point is about establishing what characterizes an insider threat. Our approach was simplistic regarding the mutability of the behaviour of each user or user type. For example, to user A, it could be normal to perform downloads at a rate of 100 downloads per second. To user B, that could be characterised as an unusual behaviour. This type of analysis of user profile can become more complex as the number of users increases, or in the presence of group attacks. Regarding the planning stage of the MAPE-K controller, more advanced strategies should be developed to choose the best way to adapt the target system when in the presence of an attack. This can be done based on the impact that each scenario generates. For example, by associating weights to impacts, and analyse what would be the cost-benefit of taking action A rather than action B, or even both.

Since in self-adaptive solutions a lot of the responsibility is shifted from the security administrator to the autonomic controller, assurances need to be provided, at runtime, that the decisions taken by the controller are indeed the correct ones. Another important issue that needs to be investigated is the new types of vulnerabilities that might be introduced into the system since the security administrator is being replaced by an autonomic controller.

Endnotes

[1] https://docs.openstack.org/keystone/latest/

[2] The CERT Division is part of the Software Engineering Institute, which is based at Carnegie Mellon University

[3] https://www.openstack.org/software/

[4] http://www.drools.org/

[5] https://www.zabbix.com/

[6] https://jmeter.apache.org/

[7] https://www.elastic.co/

Funding
This work is partially supported by the SmartMetropolis Project[8]. Nelio Cacho is supported in part by CAPES - Brazil (88881.119424/2016-01).

Availability of data and materials
The datasets used and analysed during the current study are available from the first author on reasonable request.

Authors' contributions
CES made contributions to the conception and design of the proposed solution, validating the prototype implementation, and contributing for the analysis of the experimental data and reviewing the manuscript. TD was responsible for the majority of the technical work, implementing the prototype, conducting experiments and collecting the obtained results, drafting the initial version of the article. NC participated on the drafting and critically reviewing the article for important intellectual content, and made substantial contributions to the analysis of the experimental data. RDL made substantial contributions to the conception and design of the proposed solution, to designing the experiments conducted, and critically reviewing the article for important intellectual content. All authors read and approved the final manuscript.

Competing Interests
The authors declare that they have no competing interests.

Publisher's Note
Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Author details
[1]Metropole Digital Institute, Federal University of Rio Grande do Norte (UFRN), Av. Senador Salgado Filho, S/N – 59.098-970 Natal, RN, Brazil. [2]Departament of Informatics and Applied Mathematics, Federal University of Rio Grande do Norte (UFRN), Av. Senador Salgado Filho, S/N – 59.098-970 Natal, RN, Brazil. [3]School of Computing, University of Kent, Canterbury CT2 7NF, Kent, UK.

Received: 8 November 2017 Accepted: 27 June 2018
Published online: 16 September 2018

References

1. Andersson J, de Lemos R, Malek S, Weyns D. Modeling dimensions of self-adaptive software systems. In: Cheng BH, de Lemos R, Giese H, Inverardi P, Magee J, editors. Software Engineering for Self-Adaptive Systems. chap 2. Berlin: Springer-Verlag; 2009. p. 27–47. https://doi.org/10.1007/978-3-642-02161-9_2. http://dx.doi.org/10.1007/978-3-642-02161-9_2.
2. Bailey C, Chadwick DW, de Lemos R. Self-adaptive federated authorization infrastructures. J Comput Syst Sci. 2014;80(5):935–52. http://dx.doi.org/10.1016/j.jcss.2014.02.003. http://www.sciencedirect.com/science/article/pii/S0022000014000154.
3. Bailey C, Montrieux L, de Lemos R, Yu Y, Wermelinger M. Run-time generation, transformation, and verification of access control models for self-protection. In: Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. SEAMS. New York: ACM; 2014. p. 135–144. https://doi.org/10.1145/2593929.2593945. http://doi.acm.org/10.1145/2593929.2593945.
4. Brun Y, Marzo Serugendo G, Gacek C, Giese H, Kienle H, Litoiu M, Müller H, Pezzè M, Shaw M. Engineering self-adaptive systems through feedback loops. In: Cheng BH, de Lemos R, Giese H, Inverardi P, Magee J, editors. Software Engineering for Self-Adaptive Systems, Lecture Notes in Computer Science, vol 5525. Berlin: Springer-Verlag; 2009. p. 48–70. https://doi.org/10.1007/978-3-642-02161-9_3. http://dx.doi.org/10.1007/978-3-642-02161-9_3.
5. Cappelli DM, Moore AP, Trzeciak RF. The CERT Guide to Insider Threats: How to Prevent, Detect, and Respond to Information Technology Crime, 1st ed. Addison-Wesley Professional; 2012.
6. Chadwick DW. Federated Identity Management. In: Foundations of Security Analysis and Design V, Lecture Notes in Computer Science, vol 5705. Berlin: Springer; 2009. p. 96–120. https://doi.org/10.1007/978-3-642-03829-7_3.
7. Chadwick DW, et al. PERMIS: A Modular Authorization Infrastructure. Concurr Comput : Pract Exper. 2008;20(11):1341–57. https://doi.org/10.1002/cpe.v20:11. http://dx.doi.org/10.1002/cpe.v20:11.
8. Cheng PC, Rohatgi P, Keser C, Karger PA, Wagner GM, Reninger AS. Fuzzy multi-level security: An experiment on quantified risk-adaptive access control. In: IEEE Symposium on Security and Privacy (SP '07); 2007. p. 222–230. https://doi.org/10.1109/SP.2007.21.
9. Clercq JD. Single Sign-On Architectures. London: Springer-Verlag; 2002. p. 40–58. http://dl.acm.org/citation.cfm?id=647333.722879.
10. Cole DE. Insider threats and the need fot fast and directed responde. Tech. rep.: SANS Institute InfoSec Reading Room; 2015.
11. Colwill C. Human factors in information security: The insider threat - who can you trust these days?. Inf Secur Tech Rep. 2009;14(4):186–96. https://doi.org/10.1016/j.istr.2010.04.004.
12. Dou Z, Khalil I, Khreishah A, Al-Fuqaha A. Robust insider attacks countermeasure for hadoop: Design and implementation. IEEE Syst J. 2018;12(2):1874–85. https://doi.org/10.1109/JSYST.2017.2669908.
13. Duncan A, Creese S, Goldsmith M. Insider Attacks in Cloud Computing. In: Trust, Security and Privacy in Computing and Communications (TrustCom), 2012 IEEE 11th International Conference on; 2012. p. 857–862. https://doi.org/10.1109/TrustCom.2012.188.
14. Duncan A, et al. Cloud Computing: Insider Attacks on Virtual Machines during Migration. In: Trust, Security and Privacy in Computing and Communications (TrustCom), 2013 12th IEEE International Conference on; 2013. p. 493–500. https://doi.org/10.1109/TrustCom.2013.62.
15. Feng W, Yan W, Wu S, Liu N. Wavelet transform and unsupervised machine learning to detect insider threat on cloud file-sharing. In: 2017 IEEE International Conference on Intelligence and Security Informatics (ISI); 2017. p. 155–157. https://doi.org/10.1109/ISI.2017.8004896.
16. Garkoti G, Peddoju S, Balasubramanian R. Detection of Insider Attacks in Cloud Based e-Healthcare Environment. In: Information Technology (ICIT), 2014 International Conference on; 2014. p. 195–200. https://doi.org/10.1109/ICIT.2014.43.
17. Garlan D, Cheng SW, Huang AC, Schmerl B, Steenkiste P. Rainbow: Architecture-based self-adaptation with reusable infrastructure. Computer. 2004;37(10):46–54. https://doi.org/10.1109/MC.2004.175. http://dx.doi.org/10.1109/MC.2004.175.
18. George Silowash AM, Cappelli D, et al. Common sense guide to mitigating insider threats. Tech. rep. CERT Carnegie Mellon; 2012.
19. Hu VC, et al. SP 800-162. Guide to Attribute Based Access Control (ABAC) Definitions and Considerations. Tech. rep., National Institute of Standards and Technology. VA: McLean and Clifton; 2014.
20. Kephart JO, Chess DM. The Vision of Autonomic Computing. IEEE Comput. 2003;36(1):41–50. http://dx.doi.org/10.1109/MC.2003.1160055.
21. de Lemos R, Giese H, Müller H, Shaw M, Andersson J, Litoiu M, Schmerl B, Tamura G, Villegas N, Vogel T, Weyns D, Baresi L, Becker B, Bencomo N, Brun Y, Cukic B, Desmarais R, Dustdar S, Engels G, Geihs K, GÃĤschka K, Gorla A, Grassi V, Inverardi P, Karsai G, Kramer J, Lopes A, Magee J, Malek S, Mankovskii S, Mirandola R, Mylopoulos J, Nierstrasz O, Pezza M, Prehofer C, Schafer W, Schlichting R, Smith D, Sousa J, Tahvildari L, Wong K, Wuttke J. Software engineering for self-adaptive systems: A second research roadmap. In: de Lemos R, Giese H, Müller H, Shaw M, editors. Software Engineering for Self-Adaptive Systems II, Lecture Notes in Computer Science, vol 7475. Berlin: Springer; 2013. p. 1–32. https://doi.org/10.1007/978-3-642-35813-5_1.
22. McGraw RW. Risk adaptable access control(radac). Tech. rep. National Institute of Standards and Technology. VA: McLean and Clifton; 2009.
23. Mell PM, Grance T. SP 800-145. The NIST Definition of Cloud Computing. Tech. rep., National Institute of Standards and Technology. MD: Gaithersburg; 2011.
24. Pasquale L, et al. SecuriTAS: A Tool for Engineering Adaptive Security. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. FSE '12. New York: ACM; 2012. p. 19:1–19:4. https://doi.org/10.1145/2393596.2393618. http://doi.acm.org/10.1145/2393596.2393618.
25. Qussai Y, Qutaibah A, Brajendra P, Yaser J. Mitigating insider threat in cloud relational databases. Secur Commun Netw. 2016;9(10):1132–45. https://doi.org/10.1002/sec.1405.
26. Sandhu RS, et al. Role-Based Access Control Models. Computer. 1996;29(2):38–47. https://doi.org/10.1109/2.485845. http://dx.doi.org/10.1109/2.485845.
27. Schmerl B, et al. Architecture-based Self-protection: Composing and Reasoning About Denial-of-service Mitigations. In: Proceedings of the 2014 Symposium and Bootcamp on the Science of Security. HotSoS '14.

New York: ACM; 2014. p. 2:1–2:12. https://doi.org/10.1145/2600176.2600181. http://doi.acm.org/10.1145/2600176.2600181.
28. Schultz E. A framework for understanding and predicting insider attacks. Comput Secur. 2002;21(6):526–31. https://doi.org/10.1016/S0167-4048(02)01009-X.
29. Shaikh RA, Adi K, Logrippo L. Dynamic risk-based decision methods for access control systems. Comput Secur. 2012;31(4):447–64.
30. Stolfo S, Salem M, Keromytis A. Fog Computing: Mitigating Insider Data Theft Attacks in the Cloud. In: Security and Privacy Workshops (SPW), 2012 IEEE Symposium on; 2012. p. 125–128. https://doi.org/10.1109/SPW.2012.19.
31. Tanzim KM, Shawkat AABM, A WS. Classifying different denial-of-service attacks in cloud computing using rule-based learning. Secur Commun Netw. 2012;5(11):1235–47. https://doi.org/10.1002/sec.621.
32. Yuan E, Esfahani N, Malek S. A systematic survey of self-protecting software systems. ACM Trans Auton Adapt Syst. 2014;8(4):17:1–41. https://doi.org/10.1145/2555611. http://doi.acm.org/10.1145/2555611.

Submit your manuscript to a SpringerOpen journal and benefit from:

► Convenient online submission
► Rigorous peer review
► Open access: articles freely available online
► High visibility within the field
► Retaining the copyright to your article

Submit your next manuscript at ► springeropen.com