Secure Cloud Maintenance

Protecting workloads against insider attacks

Sören Bleikertz Anil Kurmus Zoltán A. Nagy Matthias Schunter

IBM Research - Zurich

{sbl,kur,nag,mts}@zurich.ibm.com

ABSTRACT

Malicious insiders are a substantial risk for today's cloud computing infrastructures. A single malicious cloud administrator can eavesdrop or damage business-critical or personally identifiable information and computations of thousands of cloud customers. To protect cloud users against such insiders, we propose a novel approach that enables a security team to protect privacy and integrity of cloud users' workloads against attacks by system administrators during operation and maintenance. We achieve this by managing the privileges of administrators during operation and maintenance while re-establishing the security of a compute node once administration is completed. By default, administrators' access to cloud servers is disabled since cloud operation is automated. For manual maintenance operations, we propose five fine-grained privilege levels that balance the security objectives of cloud users with the operational requirements of cloud administrators. We demonstrate how existing cloud architectures need to be extended to incorporate our approach. We prototyped our management approach using the OpenStack cloud platform. Policy enforcement has been prototyped by leveraging SELinux type enforcement in the KVM compute nodes, in order to demonstrate the practical feasibility of our approach.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

General Terms

Security

Keywords

virtualization, cloud, insider attacks, workload protection

ASIACCS '12, May 2–4, 2012, Seoul, Korea.

1. INTRODUCTION

In recent years, *Cloud Computing* has gained remarkable popularity due to the economic and technical benefits provided by this new way of delivering computing resources. Businesses can offload their IT infrastructure into the cloud and benefit from rapid provisioning, scalability, and cost advantages. While cloud computing can be implemented on different abstraction levels, we focus on *Infrastructure Clouds* such as Amazon EC2 [2] that provide virtual machines, storage, and networks.

Although the benefits of Cloud Computing are evident and end-users demand cloud services, security is a major inhibitor [14] and various security risks have been identified [4, 5]. A malicious insider, such as a cloud administrator, can easily inspect the virtual machines of cloud users and retrieve sensitive information [17]. Insider attacks are constantly identified as a high-impact risk where a few malicious insiders can affect the security of many users. Furthermore, this risk of insider attacks is amplified by the fact that administration is often outsourced and thus trust in administrators is sometimes limited.

1.1 Protecting Cloud Users during Server Maintenance

While normal operations are automated and are based on trusted system management processes [19], an open challenge is how to protect end-users and their workloads during maintenance. In practice, maintenance requires substantial privileges and is often performed manually. In particular resolving complex problems requires full access where administrators can see and modify all parts of a system. Today, no technology can protect against insider attacks during such maintenance activities.

Our focus is on today's dark data centers that are remotely managed and where only a few trusted administrators have physical access to the servers. This reduces the risk imposed by human-related mistakes (cf. [21]) and allows to outsource system administration for cost reduction. We introduce three administrator roles where we require separation of duties. The first role is the hardware maintenance team, which is the only one allowed to access the datacenter and is responsible for adding and removing hardware. The second role is the security team that defines the security policy for the datacenter. This includes approval of the infrastructure cloud software executed on the systems. The third role are remote maintainers. Their responsibility is to maintain the individual compute nodes. A particular task is to perform problem determination and resolution by logging into individual in-

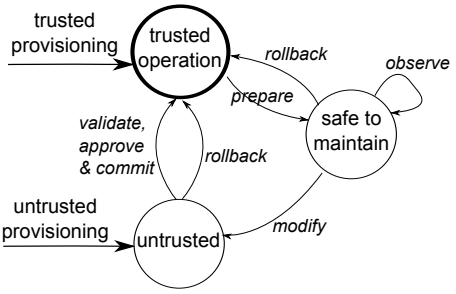

Figure 1: State Diagram for a Compute Node per Customer

frastructure elements. While the first two groups are rather small, the latter maintenance team poses the biggest security risk since this task requires a large team that is often outsourced and delegated to third parties with limited trust.

We ensure that such remote administrators cannot affect the integrity and confidentiality of customer workloads. Our approach is based on two key concepts. The first is a consistent enforcement of the principle of least privileges [18]: We define several levels of increasing privileges that administrators can select. This allows administrators to elevate their privileges as needed. Depending on the actual privileges chosen, we then protect privacy and integrity of the workload during the complete maintenance life-cycle. Furthermore, we reassess or restore the integrity of the platform before returning the compute node back into normal operation. Technically, this required us to resolve two key challenges.

The first challenge is to ensure that for low privilege roles the platform remains trusted under maintenance. The second challenge is to ensure that once a platform can no longer be trusted due to high privileged access by the administrator, the integrity of the platform can be restored and modifications of the maintenance can either be rolled back (typical for most troubleshooting operations), or more seldom, the modifications can be submitted for approval to the cloud service provider's security team for future integration of the changes on all compute nodes. For security-aware customers demanding higher transparency, it is also possible to notify or get approval from the customer.

Figure 1 illustrates the resulting life-cycle of our systems. By default, a compute node is operating and no operator can perform maintenance. If maintenance is performed, the compute node is moved into maintenance mode and its payload is protected. If the node is only observed, then it can be rolled back into operation. If changes were made that may affect the integrity of the system, the node enters an untrusted state. This node can re-enter production only after an externally verifiable rollback or "approval and commit" of the performed modifications.

Note that unlike existing proposals, we are able to protect a core trusted computing base and thus do not require special hardware such as Trusted Platform Modules (TPMs). This makes our approach feasible for commodity cloud infrastructures.

2. REQUIREMENTS & THREAT MODEL

We now specify our functional and security requirements in detail. We do not cover the functional requirements of the cloud platform and focus on the maintenance requirements of the administrator and the security requirements of the cloud user.

2.1 Maintainability Requirements of Cloud Administrators

Protecting cloud users against malicious cloud administrators by entirely disallowing access to the Cloud Computing infrastructure is only feasible during normal operations when no problems occur. A new cloud architecture that protects users against insider attacks has to balance between the security objectives of the users, the functional requirements of the administrators, and operational usability.

The following operations are common tasks for maintenance and trouble shooting of server systems: Reading log files, configuration files, and system parameters; Modifying configuration files; Patching and Installation of system binaries; Full administrative access for complicated troubleshooting (e.g., kernel related); Running existing or newly installed executables for testing purposes and root-cause analysis. While some of these maintenance tasks do not pose a threat to the end user, others such as full access have the potential to violate security and privacy of the cloud users.

2.2 Security Objectives of Cloud Users

Our main concern is the exposure of sensitive or personally identifiable information belonging to the cloud users to an administrator acting as a malicious insider. The security objectives apply only to compute nodes, and we exclude the network from our protection approach since existing approaches, such as VPNs, can be applied by the end-users.

Confidentiality requires that any remote administrator of the cloud provider must not be able to read information stored or processed by the cloud user. This includes information stored in memory and on the hard disk. A cloud user may grant an exception to an individual administrator if required due to maintenance reasons and prior informed consent is given by the user. An example motivating such consent is a problem that disappears once a VM is stopped and that cannot be reproduced with a test VM. **Integrity** requires that any remote administrator must not modify any data or executables belonging to cloud users. This includes the guarantee that the administrator cannot modify the runtime platform in a way that is not approved by the security team. **Availability:** Any remote administrator must not be able to degrade the availability of a compute node except during maintenance operations.

2.3 Threat & Trust Model

We consider curious, careless, and malicious inside attackers. We assume that an attacking administrator wants to read and/or modify data belonging to a cloud customer, which is stored or processed within the virtual machine and storage of that particular user. Due to their privileged role within the cloud provider, administrators have access to the servers hosting the virtual machines of the users. Typically this access is highly privileged (*root* access) and allows inspection and modifications of users' virtual machines. Dark datacenters with remote administration means that we assume that insiders cannot attack the physical hardware, e.g., through tampering.

Our aim is to protect against attacks by these remote administrators and we assume that the following entities behave correctly. *Infrastructure Management Software* is

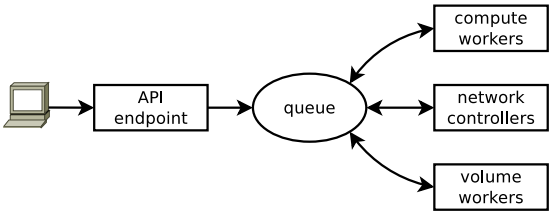

Figure 2: OpenStack Architecture

responsible for acting upon requests from the user and keeps state about the infrastructure. *Cloud Provider's Security Team* is responsible for approving server templates used for provisioning. In a high-security setting, this assumption may be reduced by requiring template approval from actual end-customers. *Server Provisioning* is based on templates approved by the security team. Other means of provisioning a server are disabled.

3. EXTENDING AN INFRASTRUCTURE CLOUD ARCHITECTURE

In this section we briefly present current and commonly used architectures for infrastructures cloud using OpenStack as an example. We introduce our extensions to these architectures in order to fulfill the goal of minimizing administrator privileges.

3.1 Current Architectures of Infrastructure Clouds

We identified four components that are commonly used in infrastructure cloud architectures. We illustrate the overall architecture and explain the components in more detail for OpenStack. However, other infrastructure cloud architectures, such as OpenNebula or VMware's vSphere, are following the same principles. An architecture overview is depicted in Figure 2.

Management Interface: The management interface is responsible for serving requests from the infrastructure cloud users and forward these requests to the appropriate components in the infrastructure. Typically, the management interface is provided in form of a web service or a specific API server. In the case of OpenStack, the *API endpoint* fulfills the role of the management interface.

Management Communication and State: The management of infrastructure clouds is composed of multiple components interacting which each other. Furthermore, the state of the infrastructure clouds is required for management decisions, e.g., for virtual machine placement based on current load of compute nodes. In the case of OpenStack, the *queue* provides a distributed message queuing platform and the *Scheduler* maintains a state of virtual machine placement.

Compute Nodes: These nodes provide computational resources to the cloud users in form of virtual machines. In case a user requests a new machine, the management infrastructure will select a compute node, e.g., based on its current load, and provision a new virtual machine for the user there. On each compute node, an interface is provided in order to manage the computational resources on this particular node. The *compute workers* fulfill this role in OpenStack.

Network & Storage Nodes Besides computational resources, infrastructure clouds also provide network and storage resources to cloud users. Similar to compute nodes, there exist nodes providing network functionality and nodes providing storage volumes, which are also controlled via a management interface. In OpenStack, *network controllers* and *volume workers* fulfill these roles respectively.

3.2 Architecture Extensions for Minimizing Administrator Privileges

We propose to extend current architectures of infrastructure clouds with the following components, in order to achieve a minimization of administrator privileges. Figure 3 illustrates the overall architecture.

Maintenance Agent: The maintenance agent (MA) is the major component in realizing an infrastructure cloud with minimized administrator privileges. It is running on each compute node and is responsible for managing administrator privileges on such a node. The maintenance agent extends the functionality of current compute nodes and their management, and it is essential for meeting the security objectives.

Maintenance Management: The management interface has to be extended for the administrators, in order to enable them to switch a compute node into maintenance mode. The extended management interface cooperates with the maintenance agent and the database of trustworthiness for switching a node into maintenance mode and record the trust level of that node.

Database of Trustworthiness of Compute Nodes: In the case of maintenance of a compute node, the trustworthiness of this node might be reduced depending on the cloud users' trust towards the provider and the criticality of their workloads. In addition, the system needs to track what nodes are in production and what nodes are in maintenance mode. Therefore, a database is required that records the state and trustworthiness of each compute node and is consulted during virtual machine placement, e.g., provisioning of a newly requested VM or due to migration.

Virtual Machine Security Labels: We introduce different security labels for virtual machines, in order to differentiate between the criticality of the workloads. A simple classification can be based on *Low*, *Medium*, and *High*. The criticality influences the placement of virtual machines when consulting the database of trustworthiness. Furthermore, it will impact the cost of switching from maintenance mode back to normal operational mode of a compute node. This requires an extension to the management interface and the management state, in order to enable the specification and recording of security labels.

Software & Template Repository: In order to start with an initial trusted state of the compute nodes, we assume a secure provisioning of the nodes with a server template approved by the security team of the cloud provider. In our architecture, we introduce a repository for software and server templates that is managed by the security team. All compute nodes are provisioned by these templates and software can only be installed from this repository. In most Linux distributions signed software packages are the norm and we can leverage this existing infrastructure.

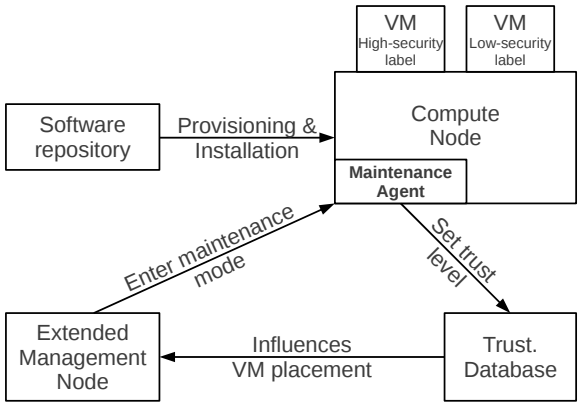

Figure 3: Extended Architecture Overview

4. MINIMIZING PRIVILEGES OF ADMINISTRATORS DURING MAINTENANCE

In Figure 1 we illustrate the overview of the maintenance life-cycle of a compute node. In this section we discuss the different states in detail and describe how the state transitions are performed. We propose a set of fine-grained privilege levels an administrator can request to enable maintenance tasks, and discuss how these levels can potentially impact the security objectives of cloud users. We explain how privileges are elevated using our maintenance agent and how eventually the impact of privileged maintenance tasks can be assessed before returning to a normal operational state.

4.1 Initial Trusted State

The initial state of compute nodes within the maintenance life-cycle of a node is their normal operational state. In that state, the system is trusted by assumption, namely that the system was securely provisioned using server templates approved by the trusted security team of the cloud provider (cf. Section 2.3). Furthermore, administrators have no privileges on the compute node and the maintenance agent is running on that node.

4.2 Privilege Levels

Based on the analysis of the desired tasks and the resources and permissions of the cloud platform, we have identified five distinct privilege levels. The privilege levels monotonically increase in capabilities, as well as their potential impact, and are modeled in consideration of the maintainability requirements of administrators (cf. Section 2.1). An overview of the levels is given in Table 1.

No access ($P_\emptyset$): This level provides no privileges at all on compute nodes and it is the default level for all administrators. For this level no impact on the cloud users' security exists.

Read-only access (P_{read}): The next privilege level provides read-only access on the compute node. The purpose is to gather system data, e.g., log files or system parameters, for initial investigations and trouble-shooting. Read access to data related to cloud users' virtual machines is prohibited, in order to maintain the security goal of confidentiality. Since this level only provides read-only access, the integrity of the users' data is not at risk.

White-listed write access ($P_{write_\circ}$): In this privilege level write access is allowed but limited with a white-list approach, i.e., we are limiting writing to a set of predefined files and resources. P_{read}'s restrictions also apply to this level. We are allowing software installation, update and removal, exclusively through a trusted repository. The software modifications are recorded in a package manager transaction log, that needs to be protected against modification. Furthermore, we allow the modification of a specified set of system parameters.

This level allows an administrator to install new software or revert software to an older version, fine-tuning system parameters, and changing configuration files. Since only software from the trusted repository (which has been approved by the security team) can be installed and modifications of files and resources are limited by a (conservative) white-list, the confidentiality and integrity of cloud users data is not at risk.

Black-listed write access ($P_{write_\bullet}$): In the case that the white-listed write access is still too restrictive for the trouble-shooting process, we can increase in privilege level to a write access with a black-list approach. Write access to any file or resource is allowed except for the following vital system and security components: bootloader, kernel, policy enforcement, maintenance agent, file system snapshots, package manager transaction logs, and certain dangerous system parameters. Since a black-list write access is much less restrictive compared to the previous white-list approach, we expect potential impact on the security objectives of cloud users. Therefore, the measures that will be explained in Section 4.4 have to be taken.

Full access (P_∞): At this level, the administrator has full access to the system and no restrictions are applied. Since an administrator can modify and read anything on the system (including the kernel) the security objectives can not be maintained at this level. Similar to $P_{write_\bullet}$, measures have to be taken for workload protection.

4.3 Policy Enforcement for Privilege Levels

The capabilities and restrictions of the privilege levels discussed in Section 4.2 are specified in a security policy that is enforced on each compute node. The privilege levels are modeled as roles and administrators are dynamically assigned to these roles. The policy enforcement system is based on *Mandatory Access Control* (MAC) and *Role-based Access Control* (RBAC). An example of such a policy enforcement system is *Security Enhanced Linux* (SELinux) [10], which we are using to illustrate our security policy.

SELinux associates a *security context* with each file, socket, device and process. A security context is essentially a *(user, role, type)* triple that is either directly specified, e.g., by *labeling* the files, or dynamically computed by SELinux. The dynamic computation is based on transition rules specified in a *security policy*. For example, the policy can specify that a process created by executing a file of type *httpd_exec_t* will transition into the type *httpd_t*. In order to cope with fine grained access control, the security policy also contains a white-list of operations, such as executing a file, binding a name to a socket, or sending a signal to a process. The white-list allows these operations to be performed by a given source type (such as a process) on a target type (such as a file, device, socket or process). Basically, SELinux policies uses these two essential mechanisms – type transition rules and allow rules – to implement the least privilege principle.

No access ($P_\emptyset$): The policy module implementing this

Level	Capabilities	Restrictions	Security Impact
$P_{\emptyset}$	none	all	none
P_{read}	read-only access	No read to virtual machines related data	none
$P_{write_{\circ}}$	white-listed write access	P_{read}'s restrictions apply. Modifications limited to software installations, white-listed files, and certain system parameters	none
$P_{write_{\bullet}}$	black-listed write access	no modifications to bootloader, kernel, policy enforcement, maintenance agent, file system snapshots, package manager transaction logs, and certain system parameters	potential disclosure and modification of virtual machine data
P_{∞}	full access	none	any disclosure and modification of platform and data

Table 1: Overview of Privilege Levels

privilege level has no allow rules at all, which means it will not allow any forms of login, e.g., spawn a system shell.

Read-only access (P_{read}): This is a restricted user shell based on SELinux's *userdom_restricted_user_template* macro, which creates a new role and type for restricted users. Furthermore, access to system and process information along with the read privilege on all file types except any type related to virtual machines (i.e., having the attribute *virt_domain* or *virt_image_type*) have been granted.

White-listed write access ($P_{write_{\circ}}$): Resource white-listing is done by specifying allow rules for the corresponding types. We explain below how we allow trusted software updates and modifications to selected system parameters.

In most systems, package management is usually not done directly by calling the package manager (rpm/dpkg), but by using a frontend (yum/apt-get). For this domain type, the policy contains a rule allowing execution of this frontend and a type transition rule forcing this program to run under its own domain after being started. The available repositories only contain signed packages, and repository changes are not permitted. New repositories could be added by modifying the frontend's configuration, but the administrator has no write access to these files. Furthermore, unsigned binaries could be installed using the actual package manager (rpm or dpkg), but there exists no transition rules for these package managers – only for the frontend – and the installation would not succeed.

System parameters are usually changed using the `sysctl` tool, but neither SELinux nor this tool does permit granting access to selected parameters in a fine-grained way. Some parameters can be used maliciously by administrators to compromise the integrity of the platform. Therefore, we construct a privileged wrapper for sysctl, which allows access to selected system parameters, that can be invoked by the administrator instead of the original sysctl tool. This is realized by giving the wrapper access to the *sysctl_t* type and specifying a transition from the user's domain type to the wrapper's domain type.

Black-listed write access ($P_{write_{\bullet}}$): The permissions of this role are based on the unconfined SELinux domain, whereby a process is permitted all operations without any restrictions. Since SELinux's policy description language does not provide deny rules, we associate a *protected_type* attribute with types corresponding to the kernel, boot process, maintenance agent and SELinux, and write allow rules for operations on all types except those with such an attribute. Since the types we are associating with the protected attribute could have allow rules defined for other types, this user domain contains no transition rules. To illustrate this with an example, a yum process ran by an administrator won't transition into yum's privileged domain type which would have access, for example, to the kernel image files. Hence, administrators can use yum to update all packages, except those associated with protected types.

Full access (P_{∞}): The user will be assigned to SELinux's unconfined role. This allows unrestricted access to the administrator, while still keeping the running services confined.

4.4 Privilege Elevation

By default administrators have no privileges ($P_{\emptyset}$). Privileges can then be gradually elevated. The maintenance agent allows at most one administrator at the time to perform maintenance on a compute node, in order to prevent potential conflicts in the integrity recovery, as well as auditing concerns. Figure 4 illustrates the overall elevation process for the different privilege levels. In this section we will discuss the generic steps involved in the elevation of privileges. Furthermore, we will explain the special transition between the $P_{write_{\circ}}$ and $P_{write_{\bullet}}$ privilege levels, which has to cope with potential impact on the cloud users' security objectives.

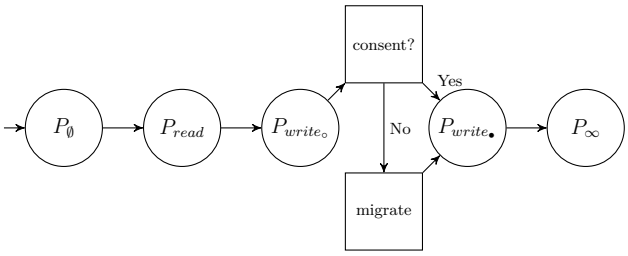

Figure 4: Overview of Privilege Elevation Process

Generic Elevation.

The *Maintenance Agent* provides an interface that allows an administrator to elevate his privileges from the current level to the next one. The ordering of privilege levels is shown in the chain-like illustration of Figure 4. Elevating privileges means that the administrative user is assigned to an SELinux user type, which is associated with the role type in the policy enforcement system that possesses the required privileges. This assignment to roles is generally sufficient for elevating to the next privilege level except for the transition from $P_{write_{\circ}}$ to $P_{write_{\bullet}}$. In case any failures arise in the elevation process, the privileges are not granted to the administrator and the security team has to intervene.

User Consent and Virtual Machine Migration ($P_{write_\circ} \rightarrow P_{write_\bullet}$).

In the case of the transition between $P_{write_\circ}$ and $P_{write_\bullet}$, we have to introduce additional mechanisms during the elevation, in order to cope with the potential impact on the cloud users' security objectives. We introduce the principle of a user consent that allows cloud users, who have virtual machines running on that particular compute node, to assess the impact of a privilege elevation and provide a consent that their virtual machines remain on the host. Imagine a scenario that a cloud user experiences problems with the cloud infrastructure and the problem only arises in combination with his workload. The user can consider to give their consent that an administrator gains $P_{write_\bullet}$ privileges, while the virtual machines remain on the host, for trouble-shooting purposes.

We envision that the user consent can be obtained by the maintenance agent in the form of sending an email to all the users served on the particular compute node. The email contains a link to a web service call at the cloud provider, which requires user authentication, that would redirect the user consent to the maintenance agent. Incorporating a nonce, i.e., a random value, in the link would guarantee the freshness of the user consent. In order to have a more efficient consent process that places less burden on the end-users, we can consider the *Virtual Machine Security Label* (cf. Section 3.2): for *Low* implicit consent is given, for *Medium* an automated decision at the cloud provider-side is made, and for *High* a human administrator (e.g., from the cloud provider's security team or the customers) has to make the decision.

In the case that consent is not provided, e.g., due to explicit deny or due to time-out, the corresponding virtual machines are securely migrated to a different compute node. The migration has to contact the *Database of Trustworthiness of Compute Nodes* (cf. Section 3.2), in order to decide on the selection of the target compute node. Furthermore, the maintenance agent has to record the untrustworthiness of the current compute node for the particular user in the database. Remaining traces of a virtual machine, e.g., swap storage and access to shared storage, have to be removed. For performance reasons, compute nodes have swapping disabled, therefore no traces are left in swap storage. Access to shared storage need to be revoked by the maintenance agent.

Limiting Privilege Exhaustion.

We are following the least privilege principle for the maintenance tasks, i.e., an administrator should only obtain the privileges required for a specific task. Therefore, administrators start in $P_\emptyset$ and have to gradually increase their privileges; instead of elevating privileges immediately from $P_\emptyset$ to P_∞. Furthermore, a barrier for the elevation needs to be introduced, in order to prevent gradual but immediate elevation to P_∞. This barrier can be implemented as a time delay, i.e., the administrator has to wait before the next elevation, obtaining approval from another administrator, or by audit logs where each administrator logs a reason for the requested privilege. A global limit prevents an administrator to obtain privileges on a majority of compute nodes, in order to prevent denial of service attacks by placing all nodes in a maintenance state.

4.5 Privilege Revocation and Recovery

Revocation of privileges is the counterpart to the elevation described previously. However, privilege revocation alone is not sufficient for guaranteed return to an initial trusted state once a maintenance task is completed. Our system has to assess the modifications performed during the maintenance task and act on these modifications based on the recovery strategy selected by the administrator. The administrator can decide between *rollback* and *commit*. Rollback dismisses all modifications to the system performed during the maintenance. These modifications can affect the state of the machine (processes, system parameters) and data at rest (files on disk). On a contrary, commit allows to keep the modifications, but they have to be approved by the cloud users and/or the security team using a consent process.

Figure 5 illustrates the overall revocation and recovery process when revoking privileges from $P = \{P_{read}, P_{write_\circ}, P_{write_\bullet}, P_\infty\}$ back to $P_\emptyset$. The *rollback* recovery eventually leads back to a trusted state. Recovery using *commit* however might lead to an untrusted state when the user consent is not obtained. For $P_{write_\bullet}$ and P_∞ we can only perform a partial commit recovery (illustrated by the dashed edges), because modifications of the system state cannot be detected under the influence of such a high privilege access. The revocation is performed analogously to *Generic Elevation* in Section 4.4, i.e., the administrative user is assigned to the role of privilege level $P_\emptyset$ by the maintenance agent. In the following we describe the recovery process.

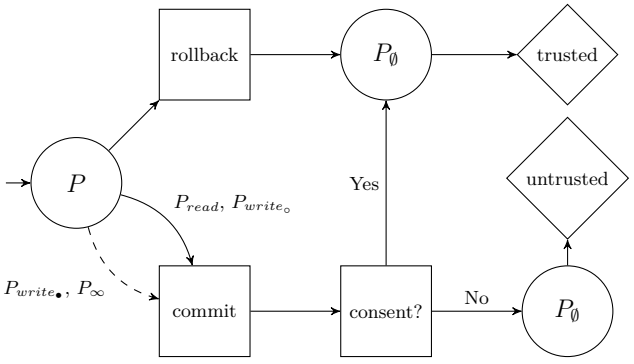

Figure 5: Overview of Revocation and Recovery Process

Preparing Recovery during Privilege Elevation: Before elevating to privilege levels that have write access, i.e., $P_{write_\circ}$, $P_{write_\bullet}$, and P_∞, the maintenance agent has to initiate the creation of a snapshot of the system. This snapshot covers data at rest (file system snapshot) and the system state (processes, system parameters). For $P_{write_\circ}$ and $P_{write_\bullet}$ one can create a system snapshot locally that will be protected by the policy enforcement. A snapshot of the file system can be created using the logical volume manager (LVM) or snapshot functionality in modern filesystems, such as Btrfs[1]. We also have to take a snapshot of the run-time system parameters (e.g., using `sysctl`). In the case of P_∞, where the administrator has full access and the snapshots cannot be protected locally anymore, we either create the snapshot remotely or use the more expensive recovery strategy of re-provisioning.

[1] `https://btrfs.wiki.kernel.org/index.php/Main_Page`

Remote snapshot requires that compute nodes are using an external storage provider.

Recovery from Read-only Access (P_{read}): Read-only access with program execution privilege only leads to modifications of the system state, i.e., running processes. This could impact the availability of the compute node, e.g., due to a started process with high resource requirements, therefore the recovery mechanism is needed. The maintenance agent identifies programs started by the administrator based on the process' security context, where the user equals the administrator. *Rollback:* Processes started by the administrator are killed. *Commit:* Processes started by the administrator need to be verified. The following information is sent to the verifiers: hash of the binary, binary for download, program arguments.

Recovery from White-listed Write Access ($P_{write_\circ}$): Using the snapshot of the file system, we generate a diff between the current file system and the snapshot. Furthermore, we generate a diff between the system parameters using the system snapshot. *Rollback:* Changed files are replaced by their copies from the snapshot. Using the transaction log of the package manager, we can rollback all modified packages. System parameters are restored with the values recorded in the snapshot. Recovery from started programs is also applied (cf. P_{read} recovery). If there exists changes in configuration files, we use the packages meta-data, i.e., which files belong to a package, to identify the corresponding service and restart it after reverting the configuration file.*Commit:* Updates to files and system parameters as well as the transaction log of the package manager are sent to the verifier for review. The *commit* from P_{read} recovery also applies.

Recovery from Black-listed Write Access ($P_{write_\bullet}$): Recovering from $P_{write_\bullet}$ is similar to the $P_{write_\circ}$ recovery. However the system state cannot be assessed, because arbitrary changes to binaries are possible, e.g., modifying processes in memory. *Rollback:* Same file system rollback as shown for $P_{write_\circ}$. A reboot is required to rollback the changes in the state. *Commit:* Same file system commit as shown for $P_{write_\circ}$. A commit for the system state is very challenging and practically not feasible, because we would need to detect in-memory modifications of processes. Therefore a reboot is required.

Recovery from Full Access (P_∞): The recovery from P_∞ is the most expensive operation, because of the large possible impact of this high privileged access. Two possible strategies are available: recovery using remote file system snapshots or re-provisioning. *Rollback:* A reboot is required to rollback the system state. For file system rollback, we either revert from a remote snapshot or re-provision the node. *Commit:* In the case of remote snapshots, a file system diff can be computed on the storage node and sent to a verifier (e.g., security team or even the customer in some cases). Otherwise, *commit* is not available for the file system. It is also not available for the system state (cf. $P_{write_\bullet}$).

Returning to a Trusted State: The objective of the revocation and recovery process is to return from the maintenance mode back to an operational and trusted state. Figure 5 illustrates that the rollback recovery strategy leads to a trusted state after the privileges are revoked back to $P_\emptyset$. Alternatively, the commit recovery involves a user consent approach similar to the one introduced in the elevation (cf. Section 4.4). Based on the *Virtual Machine Security Label* we can achieve a more efficient consent process that also places less burden on the end-users: for *Low* implicit consent is given, for *Medium* an automated decision at the cloud provider-side as well as providing a succinct description of the maintenance operations to the users whose VMs were kept on the node during maintenance, and only for *High* a human verifier (e.g., from the cloud's security team or from each user who gave consent) is involved in the decision.

In the case where we involve each user (or a delegated authority) that is or was hosted on that particular compute node in approving the modifications performed during the maintenance task, depending on the outcome of the consent process, the system is in a trusted or untrusted state from each user's perspective. If there exists users who did not give consent, the associated VMs have to be migrated away. Overall, this might lead to a scenario where a compute node is in a trusted with respect to some users, and in an untrusted state with respect to others. This trust fragmentation of the system degrades its efficiency. Therefore the security team should employ a "garbage collection" process, where administrators are asked to either integrate their changes into an official server template or the affected nodes are re-provisioned.

In the case that a trusted state is reached, the trustworthiness of the compute node is updated for the users that provided consent in the trustworthiness database. When the system recovers from $P_{write_\bullet}$ or P_∞, the virtual machines that were migrated away during elevation are migrated back. In the case of an untrusted state, this compute node remains unusable for the affected users.

5. SECURITY DISCUSSION

We now discuss the actual security achieved by our system. We aim at satisfying the security objectives described in Section 2.2 based on the mechanisms described in Section 4. Our focus is to prevent mis-use of obtained privileges. We do not consider attacks through physical attacks, platform vulnerabilities, and covert channels since both are orthogonal to our approach.

We discuss how each privilege level may impact the security goals of integrity, availability and confidentiality, and how we ensure that the impact is mitigated by our system. $P_\emptyset$ is the trivial case with no privileges, therefore no impact on the security goals exists.

5.1 Integrity

The first objective is integrity. This includes integrity of all user resources (VM memory, VM disk image) as well as integrity of the computing platform. However, in our approach we exclude integrity of information transmitted over the network, as the cloud user should employ other protection means, such as Virtual Private Networks.

Read-only access (P_{read}) prohibits write modifications and the integrity of the platform and resources is ensured. Furthermore, we prevent read access to users related data that might contain other forms of access credentials with write privileges. However, the administrator is allowed to execute programs that have to be terminated once the maintenance is completed.

White-listed write access ($P_{write_\circ}$) is tightly controlled and limited to modifications of white-listed files and system parameters, and software installations. The integrity can be restored in the rollback process by replacing modified files with copies from a snapshot, restore system parameters to

previous values, and rollback all software installations using a package manager transaction log. The integrity of the snapshot and the transaction log is ensured by the policy enforcement system. Modifying system parameters could amplify the risk of software vulnerabilities due to the disablement of protection mechanisms. However, we are not concerned about software vulnerabilities in our approach.

Black-listed write access ($P_{write_\bullet}$) allows any modifications except of vital system and security components, namely, bootloader, kernel, policy enforcement, maintenance agent, snapshots, package transaction logs, and dangerous system parameters. The integrity of user related data cannot be ensured at this privilege level anymore, therefore the workload is either migrated away or the user accepts the risk by giving consent. Migration has ensured that all traces of the workloads are removed, i.e., swap files are disabled which could contain old memory pages and access to shared storage is centrally disabled. Overall it is crucial that the administrator cannot modify the policy enforcement, which we ensure by prohibited access to any SELinux related data. Integrity of the platform's file system can be restored by the rollback process (cf. $P_{write_\circ}$), but it requires a correct and thereby protected maintenance agent as well as protected snapshots and transaction logs. The state of the system can be arbitrarily modified, e.g., by modifying programs directly in memory, therefore a reboot ensures a rollback of the system state. Since the bootloader and kernel are protected, a reboot will lead to a trusted state again.

Full access (P_∞) allows any modifications to the platform. User data was either migrated away or the user accepted the risk before entering $P_{write_\bullet}$. Platform integrity can only be restored by reboot and provisioning or reverting of remote snapshots. We are aware of potential attacks that could circumvent the re-provisioning process, i.e., stealth and persistent malware such as an attack demonstrated by Wojtczuk and Rutkowska [23]. However, these are out of scope for our approach and fall in the category of platform vulnerabilities.

5.2 Confidentiality

Confidentiality requires that an administrator cannot see payload data of the users of the virtual infrastructure. This includes memory, disk and snapshots of past disk contents, processor state, log-files, and network traffic. A pre-condition of confidentiality is the integrity of the platform; if an administrator can modify the platform, then it may remove policy enforcement mechanisms.

Read-only access (P_{read}) is restricted by prohibiting access to cloud users related data, which have their own specific label in SELinux. This includes the guest VM's processor state and memory, as it is represented as a process with the SELinux label, as well as disk images, which are stored as labeled files. Similar to the integrity discussion of P_{read}, the risk of successful exploitation of software vulnerabilities could be increased by being able to read certain system parameters, e.g., reading memory maps of processes could defeat random address space layouts (ASLR[2]). In our approach, we do not consider the confidentiality of network traffic. Furthermore, log files in the hypervisor may only contain high-level information about the customer workloads, e.g., resource requirements, but not the actual workload or other sensitive information of the cloud users.

White-listed write access ($P_{write_\circ}$) has the same impact on confidentiality as P_{read}, which is covered by our approach.

Black-listed write access ($P_{write_\bullet}$) has potential impact on the confidentiality of cloud users data due to the possibility of platform changes. Therefore, the same measures apply as used for the integrity protection, namely, virtual machine migration or user consent.

Full access (P_∞) allows any disclosure of information, but user related data was either migrated away or consent was given.

5.3 Availability

The availability of a compute node can only be negatively influenced by an administrator when modifying the system state or the file system. For example, a process can be started that consumes all the resources on the node or modifications causes problems on the platform. Therefore, availability is tightly connected to platform integrity, and the integrity rollback will also ensure that availability returns to the prior level once maintenance is completed.

6. IMPLEMENTATION

Our prototype implementation is based on the open-source cloud platform *OpenStack*[3]. The maintenance agent is written in 200 lines of Python and has a RESTful API for providing the maintenance functionality. For the interaction with SELinux, e.g., for role assignment of users, we are leveraging the SELinux's Python interface. We are highlighting the implementation details for performing a commit operation with user consent. In order to assess the modifications, the agent performs the following operations: compute a diff between the sysctl settings before and after the maintenance; generate a list of filesystem changes by running `rsync` against the snapshot: for binaries we provide a hash, for non-binaries a text diff; determine programs started by the administrator by iterating `/proc` and looking for process IDs with the inherited administrator's user label; generate a list of package changes using yum's internal transaction log. The assessed modifications are collected in an email that is sent to the compute node verifiers with a link to the agent's web interface for approval or rejection.

7. RELATED WORK

This paper focuses on the security of infrastructure clouds. We build on related work from several areas: Virtual systems security aims at reducing the security risks introduced by virtual machines, network, and storage. Trusted computing enables stakeholders to verify the integrity of given IT systems. We focus on linux-based virtual machine monitors. As a consequence, a final area of related work are Linux security policies and their enforcement.

Infrastructure Cloud Security: The first area of related work is security of virtual machine monitors. Virtual systems introduce several new security challenges [7]. This knowledge is needed to underpin the user's individual decisions whether to trust a given component. Analysis of well-known attacks such as jailbreaks [22] allows one to detect vulnerable configurations. This includes information leakage vulnerabilities of today's infrastructure clouds that

[2] `http://pax.grsecurity.net/docs/aslr.txt`

[3] `http://www.openstack.org`

allow covert or overt communication between multiple tenants that should be isolated. Examples include co-hosting validation [16] and cache-based side channels [1, 15]. While these vulnerabilities are important in practice, they are orthogonal to our approach. Our goal is to prevent insiders from obtaining additional means of attack. We accept the fact that insiders can act as end-users and exploit potential vulnerabilities to attack a given system.

SELinux in Infrastructure Clouds: SELinux is used in many projects to provide fine grained access control: it is for example included by default in the widely used Red Hat Enterprise Linux distribution, mostly providing an additional sandboxing layer for various services that can be run on a Linux server. As such, it is not surprising that SELinux and other MAC technologies are used in infrastructure clouds [3, 9], mostly to enforce isolation of resources that are shared among cloud users (multi-tenancy). In contrast, we use SELinux for restricting administrative privileges.

Trusted Computing and Virtual Infrastructures: For Linux-based virtualization platforms such as KVM and Xen, important pieces of related work are [11, 12, 13] where a software architecture based on Linux is proposed that provides attestation of all executables and configuration files. Another approach to use trusted computing for verifying virtual infrastructures has been proposed in [6] where tamper-proof hardware is virtualized to allow for multiple concurrent yet isolated protected VMs. In [19] the authors have sketched how to use trusted computing for validating a cloud infrastructure. While each of these approaches provides certain assurance to the end users, none provides a concept for securing the maintenance of such clouds. As a consequence, they are useful during normal operations but only manage to declare a system untrusted once it is maintained.

8. CONCLUSIONS AND FUTURE WORK

In this paper we have mitigated the threat of insider attacks by remote administrators in dark datacenters. Unlike other approaches, ours is applicable to commodity cloud infrastructures such as OpenStack or OpenNebula. In contrast to existing security concepts for clouds, our approach includes maintenance, does not require special hardware, and does not have significant impact on the efficiency of the infrastructure cloud.

While we addressed the challenge for compute nodes, some open questions remain. The first is to develop a similar concept for storage nodes and their administrators. Our concept uses storage nodes as a building block without elaborating how to securely maintain them. In particular, protecting also storage nodes against inside attackers would be desirable. A second area of further investigation is to further validate our concept. This includes evaluation in large-scale practical field studies as well as a formal evaluation of the SElinux policies similar to [8].

9. REFERENCES

[1] Aciiçmez, O. Yet another microarchitectural attack: exploiting i-cache. In *CSAW '07: Proceedings of the 2007 ACM Workshop on Computer Security Architecture* (2007), ACM, pp. 11–18.

[2] Amazon. The Amazon Elastic Compute Cloud (EC2). Available at `http://aws.amazon.com/ec2/`, last accessed March 2010, 2010.

[3] Baiardi, F., and Sgandurra, D. Securing a community cloud. In *Proceedings of Distributed Computing Systems Workshops (ICDCSW)* (june 2010), pp. 32 –41.

[4] Cloud Security Alliance. Security Guidance for Critical Areas of Focus in Cloud Computing, December 2009.

[5] ENISA. Cloud Computing Risk Assessment. Tech. rep., ENISA, 2009.

[6] Garfinkel, T., Pfaff, B., Chow, J., Rosenblum, M., and Boneh, D. Terra: A Virtual Machine-Based Platform for Trusted Computing. *SIGOPS Oper. Syst. Rev. 37*, 5 (2003), 193–206.

[7] Garfinkel, T., and Rosenblum, M. When Virtual is Harder than Real: Security Challenges in Virtual Machine Based Computing Environments. In *HOTOS'05: Proceedings of the 10th conference on Hot Topics in Operating Systems* (Berkeley, CA, USA, 2005), USENIX Association, pp. 20–20.

[8] Jaeger, T., Sailer, R., and Zhang, X. Analyzing integrity protection in the selinux example policy. In *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12* (Berkeley, CA, USA, 2003), USENIX Association, pp. 5–5.

[9] Kurmus, A., Gupta, M., Pletka, R., Cachin, C., and Haas, R. A comparison of secure multi-tenancy architectures for filesystem storage clouds. In *Proceedings of the 12th International Middleware Conference* (Dec 2011).

[10] Loscocco, P., and Smalley, S. Integrating flexible support for security policies into the linux operating system. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference* (June 2001), USENIX Association.

[11] MacDonald, R., Smith, S., Marchesini, J., and Wild, O. Bear: An open-source virtual secure coprocessor based on TCPA. Tech. Rep. TR2003-471, Department of Computer Science, Dartmouth College, Hanover, NH, USA, 2003.

[12] Marchesini, J., Smith, S. W., Wild, O., and MacDonald, R. Experimenting with TCPA/TCG hardware, or: How I learned to stop worrying and love the bear. Tech. Rep. TR2003-476, Department of Computer Science, Dartmouth College, 2003.

[13] Marchesini, J., Smith, S. W., Wild, O., Stabiner, J., and Barsamian, A. Open-source applications of TCPA hardware. In *20th Annual Computer Security Applications Conference* (Washington, DC, USA, December 2004), ACM, IEEE Computer Society, pp. 294–303.

[14] Mell, P., and Grance, T. Effectively and Securely Using the Cloud Computing Paradigm, October 2009.

[15] Percival, C. Cache missing for fun and profit. `http://www.daemonology.net/papers/htt.pdf`, May 2005.

[16] Ristenpart, T., Tromer, E., Shacham, H., and Savage, S. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *CCS '09: Proceedings of the 16th ACM conference on Computer and Communications Security* (New York, NY, USA, 2009), ACM, pp. 199–212.

[17] Rocha, F., and Correia, M. Lucy in the sky without diamonds: Stealing confidential data in the cloud. In *Proceedings of the 1st International Workshop on Dependability of Clouds, Data Centers and Virtual Computing Environments (DCDV, with DSN'11)* (June 2011).

[18] Saltzer, J. H., and Schroeder, M. D. The protection of information in computer system. *Proceedings of the IEEE 63*, 9 (1975), 1278–1308.

[19] Santos, N., Gummadi, K. P., and Rodrigues, R. Towards trusted cloud computing. In *Proceedings of the 2009 conference on Hot topics in cloud computing* (Berkeley, CA, USA, 2009), HotCloud'09, USENIX Association, pp. 3–3.

[20] Van Dijk, M., and Juels, A. On the impossibility of cryptography alone for privacy-preserving cloud computing. In *Proceedings of the 5th USENIX conference on Hot topics in security* (Berkeley, CA, USA, 2010), HotSec'10, USENIX Association, pp. 1–8.

[21] Weisman, R. Dark Data Center Design . Processor Vol.28 Issue 35, Sep 2006.

[22] Wojtczuk, R. Adventures with a certain Xen vulnerability (in the PVFB backend). `http://invisiblethingslab.com/pub/xenfb-adventures-10.pdf`, October 2008.

[23] Wojtczuk, R., and Rutkowska, J. Attacking smm memory via intel® cpu cache poisoning. Tech. rep., Invisible Things Lab, 2009.