Inevitable Failure: The Flawed Trust Assumption in the Cloud

Yuqiong Sun
Department of Computer Science and Engineering
Pennsylvania State University
yus138@cse.psu.edu

Giuseppe Petracca
Department of Computer Science and Engineering
Pennsylvania State University
gxp18@cse.psu.edu

Trent Jaeger
Department of Computer Science and Engineering
Pennsylvania State University
tjaeger@cse.psu.edu

ABSTRACT

IaaS clouds offer customers on-demand computing resources such as virtual machine, network and storage. To provision and manage these resources, cloud users must rely on a variety of cloud services. However, a wide range of vulnerabilities have been identified in these cloud services that may enable an adversary to compromise customers' computations or even the cloud platform itself. Using the motivation for adding mandatory access to commercial operating systems, we argue for the development of a *secure cloud operating system* (SCOS) to enforce mandatory access control (MAC) over cloud services and customer instances. To better understand the concrete challenges of building a SCOS, we examine the OpenStack cloud platform from two perspectives: (1) how attacks propagate across cloud services and (2) how adversaries leverage vulnerabilities in cloud services to attack hosts. Using this information, we review the application of three MAC approaches employed by âĂIJsecureâĂİ commercial systems to evaluate their practical effectiveness for controlling cloud services. While MAC enforcement can improve security for cloud services, several threats remain unchecked. We outline a set of additional security policy goals that a SCOS must enforce to control threats from potentially compromised cloud services comprehensively. While we do not actually construct a SCOS in this paper, we hope that this study will initiate discussions that may lead to practical designs.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection

General Terms

Design, Experimentation, Security

Keywords

cloud computing, security, mandatory access control

1. INTRODUCTION

Cloud computing platforms rely on a variety of *cloud services* to manage the execution of customer instances. For example, in the OpenStack cloud platform [42], cloud services authenticate customers, process customer requests to run and update compute instances, manage storage resources, schedule compute resources, etc. Such services often must collaborate to process a request, so OpenStack also has a central database for system state and a messaging service for communication among services.

Recently, a wide variety of vulnerabilities have been identified in these cloud services, ranging from authentication bypass [13, 12, 14] to denial-of-service [25, 23, 24] to failed input validation [33, 32, 31]. These vulnerabilities are particularly problematic because the OpenStack platform appears to assume complete trust among services. Thus, a compromise in any one service may impact the integrity of other cloud services. Further, compromised cloud services can launch a variety of programs as `sudo` "root" processes, enabling adversaries to install rootkits on cloud service hosts.

While cloud platform vendors take countermeasures to protect cloud services from attack and protect the host running cloud services, current approaches are incomplete. In OpenStack, the goal is to authorize requests before forwarding them to services. However, each request is converted to several method calls among services, and there is no validation that each method call invoked is safe. Further, the trust among services means that such attacks can be easily propagated through the service network. Also, cloud vendors are now careful to run cloud services under limited user identities to protect the hosts from attack by compromised services. For example, cloud services deployed as web applications (i.e., Horizon) run under web server identities (e.g., Apache) and other services run under service-specific user identities. However, many cloud services have sudo privilege and/or use other processes that run with full privilege. Thus, a variety of methods are available to an adversary to leverage vulnerabilities in a cloud service to launch a local exploit.

Researchers have mainly focused their attention on attacks originating from the hosted computing. For example, research in hypervisors aims to protect one customer from another when the host operating system be compromised [62]. Also, researchers have explored a variety of attacks that leverage covert or side channels on co-located processing [47, 63, 55]. Commercial vendors have proposed "cloud operating systems" that mainly focus on how resources are virtualized among customer computing [16, 17]. While security is not the primary goal in most cases, these operating systems often offer features that aim to enhance security by protecting customer data on hosts, reducing the amount of trusted code, etc. However, these operating system projects do not focus on the bigger picture that includes cloud services.

In 1998, Loscocco *et al.* published a paper called the "The Inevitability of Failure" [35], which highlighted the problem of having application processes manage the security of a system. At the

CCSW'14, November 7, 2014, Scottsdale, Arizona, USA.

http://dx.doi.org/10.1145/2664168.2664180.

time, the important security decisions in commercial systems were made by a variety of applications: authenticating users, configuring access control, installing kernel modules, and launching privileged processes. As application programmers tended to add security mechanisms as an afterthought based on the specific vulnerabilities detected, application security mechanisms are often ad hoc and inconsistent among applications. Further, remote adversaries were often able to access and exploit vulnerabilities in these applications, leading to compromise of the entire host. Loscocco *et al.* argued for fundamental security mechanisms, such as access control, to be implemented by a "secure operating system" that would govern applications. The idea is that the secure operating system should implement *mandatory access control* (MAC) enforcement [1] to ensure that a system-defined access control policy is correctly enforced across all applications.

We posit that a similar situation is occurring in cloud computing. In cloud computing, the cloud services, like applications previously, are responsible for the security decisions over cloud resources. Similarly, a variety of vulnerabilities have been discovered in cloud services. Cloud services trust one another completely, enabling adversaries to propagate vulnerabilities across services. Finally, cloud services have powerful permissions on their hosts, enabling a compromised cloud service to take over the entire host. As a result, the cloud resources and cloud platform are both vulnerable to cloud services in much the same way that the host operating system was vulnerable to application processes as described by Loscocco *et al.*.

In this paper, we argue for the development of a "secure cloud operating system" (SCOS) to enforce mandatory access control over cloud services and customer instances. We examine the possible designs for a SCOS from three perspectives. First, a SCOS must enforce a mandatory access control policy that protects the cloud resources and cloud platform (i.e., the SCOS) from compromise. As a result of the Loscocco *et al.* paper, commercial operating systems were enhanced with security mechanisms to enforce mandatory access [61, 60, 52, 38], but these mechanisms were used to enforce a variety of policies. So, in Section 4 *we review the application of mandatory access control approaches employed by "secure" commercial systems to evaluate their practical effectiveness*. Second, a SCOS must be able to enforce these policies and ideally address weaknesses of previous MAC policy models. In Section 5, *we outline a strawman design for a SCOS that models MAC policies over collections of protection domains associated with customers and requests*. While we do not actually construct a SCOS in this paper, we hope that this study will initiate discussions that may lead to practical designs. Third, it may be argued that despite the addition of MAC security mechanisms in commercial systems, we still see a wide variety of vulnerabilities. Thus, in Section 6 *we enumerate ideal policy goals and highlight the benefits of the SCOS relative to those goals and the challenges in achieving those goals.*

To better understand the concrete challenges of building a SCOS, we examine these questions in the context of the OpenStack cloud platform. We study the OpenStack cloud platform from two perspectives. First, we examine how customer requests are processed by the OpenStack services to collect the expected interactions among services. Second, we study a trace of the permissions actually used by an OpenStack deployment to examine the efficacy of different mandatory access control approaches. Using this data, we produce estimates about the MAC policies that would result from three different policy models employed in commercial systems. Our findings are similar to those found for the commercial MAC enforcement approaches on hosts [15]: a significant number of services cannot be completely confined, so current MAC enforcement alone does not close the entire attack surface.

2. BACKGROUND

In this section, we motivate the idea that a cloud platform is analogous to classical multiuser systems across a distributed system using the OpenStack cloud platform as example. Other cloud platforms, such as OpenNebula [41], CloudStack [2] and Eucalyptus [26] are designed in a similar way.

2.1 Background on OpenStack

OpenStack is an open-source, cloud software stack for building public or private IaaS clouds [42]. It aims to enable customers to configure, run, and manage virtualized computations, called *instances*, with minimal effort and maximal scalability. To do this, OpenStack provides customers with the ability to launch virtual machines (VMs) on cloud nodes managed by OpenStack. Customers have full flexibility for deciding how to configure their OpenStack instances by choosing the system image (e.g., operating system distribution), choosing applications to launch in the instance, and uploading their application data to the instance by submitting *requests* using the OpenStack API [43].

OpenStack embraces a modular architecture of *services* for processing customer requests as shown in Figure 1. An API service converts requests to the OpenStack API into *method calls* that other services execute. Services communicate by submitting method calls through a messaging system that uses the Advanced Message Queue Protocol (AMQP). Typical OpenStack services are stateless. Any state or configuration (e.g., users, credentials, and instances) necessary to execute a method call must be retrieved from a central database that only the OpenStack services are supposed to access. Services are authorized to perform privileged operations using `sudo`. By using `sudo`, only some programs invoked by services run with full privilege, limiting the attacks available to adversaries. However, some root vulnerabilities that exploit sudo processes have been reported (e.g., [20]).

Customers and cloud administrators can communicate with the network-facing cloud services through either a web-based interface called OpenStack dashboard or by directly accessing the OpenStack API Service (`nova-api`) which is a RESTful web service endpoint. All requests are forwarded to the OpenStack API service as shown in Figure 1. Using the OpenStack API, customers may use services to provision (setup) and manage (update) their cloud instances. All other operations use a separate network and do not involve the cloud services. Thus, the cloud services are not performance critical, and in OpenStack, the compute-service assumes that only one request is being processed on an instance at a time (i.e., no synchronization to prevent race conditions).

To better illustrate the modular design of OpenStack and various OpenStack services, we show a simplified view of how a customer request is processed by OpenStack in Figure 1. We will focus on request to launch an instance, which is just an example. There are over 100 OpenStack request APIs and OpenStack processes, which operate in a similar way. In order to launch an instance, a customer submits a request (step 1 in Figure 1) via either the dashboard (web interface) or by directly accessing `nova-api` service (through nova CLI). The *nova-api* requests authentication of the customer and check if the customer is authorized to create a new instance by consulting the `Keystone` service (step 2). After authorization, the `nova-api` receives the request, it creates a database entry describing the instance (step 3) and sends the handler to the scheduler (step 4). The scheduler is an OpenStack service named `nova-scheduler`. The scheduler selects a specific cloud node

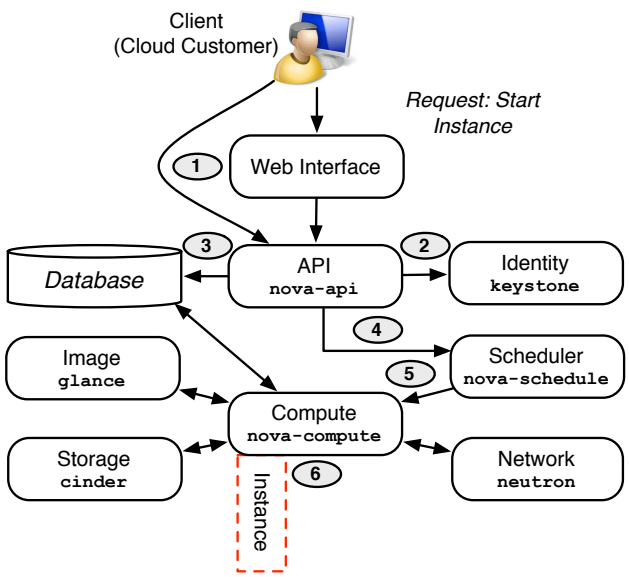

Figure 1: Illustrating the OpenStack modular service architecture and communications necessary to create a new instance.

running a `nova-compute` service to host the new instance based on predefined scheduling policies. The scheduler submits a request (step 5) to create a new instance containing parameters that define the new instance, such as a base image, instance type and so on. Based on these parameters, the `nova-compute` service will then retrieve the base image, network configuration, and other resources necessary to launch the image as specified from the associated services, such as the image, volume, and network services (step 6).

2.2 OpenStack Vulnerabilities

A wide variety of vulnerabilities have been discovered in OpenStack services. In this section, we review these vulnerabilities to demonstrate that although few OpenStack services are directly accessible to remote adversaries and run on a separate network from customer instances, the discovered vulnerabilities enable compromise of services, propagation of attacks to other services, and even compromise of the host operating systems.

One fundamental countermeasure cloud platform vendors take to protect cloud services is network isolation. The network used by cloud services is isolated from the network used by instances, and most of cloud services are not addressable from the Internet. However, in order to serve customers, cloud services such as API service (e.g., nova-api, neutron-server) and authentication service (e.g., Keystone) must be made public facing, thus accessible to remote adversaries. Many vulnerabilities have been identified in such public facing cloud services. One reason is due to the large size and complexity of the OpenStack API.

Such attacks can easily propagate through the network, even to services that do not face the public network, because services fully trust one another. As example, the `nova-compute` service accepts file paths from *nova-api* service specifying locations to upload data to an instance. However, if the file path is not validated by the *nova-compute* service, it can cause a directory traversal attack [20, 21, 22] and permit the `nova-api` service to overwrite arbitrary files on the compute node. As another example, OpenStack object storage service (SWIFT) uses the `pickle` Python module to load metadata from the internal network which is assumed to be trusted. Since the `pickle` Python module allows its âĂIJpickledâĂİ data to be executed, attackers could escalate their privileges from network access to arbitrary code execution on SWIFT nodes [53].

Vulnerabilities in cloud services may further allow adversaries to gain control over the hosts running those services. For example, multiple directory traversal vulnerabilities were found in the `nova-api` service, allowing attackers to overwrite arbitrary files on service host. As another example, OpenStack image service (Glance) executes commands constructed based on metadata of images uploaded by cloud users without properly validating it. Attackers could then execute arbitrary commands on the Glance service host by crafting the metadata of an image and uploading it to the cloud [28]. Some of these vulnerabilities may be leveraged to compromise the entire host because many of the cloud services have `sudo` privilege and/or use other `setuid` processes that run with full privilege. For example, the directory traversal vulnerability [20, 21, 22] discussed above was found in a function where `nova-compute` tried to write files on the host via sudo. The consequences are catastrophic, because arbitrary files on the compute node, even those ones owned by `root`, could be overwritten.

2.3 Cloud as a Multiuser System

We observe that the cloud ecosystem that is evolving in OpenStack (and likely in other cloud platforms) is starting to resemble a traditional multiusers system, albeit one which is distributed across several hosts. In a traditional multiuser system, multiple users can utilize the system resources to execute a common set of application programs. To keep users from interfering with one another, each user is given her own storage, typically in a file system subtree rooted at her home directory. Multiuser systems also run several system applications that have privileges that typical users do not. In particular, these system applications may perform vital and security-sensitive operations for the system, so their compromising would lead to a compromised system.

Just this problem befell multiuser systems in the late 1990s and early 2000s as one system application after another was compromised several times with large-scale impact. While vulnerabilities were soon patched, it was not long before the next vulnerability was exploited and the cycle was repeated. More significant countermeasures were necessary. One such proposed countermeasure was to extend commercial operating systems with mandatory access control (MAC) enforcement [1]. The main goal was to provide a barrier that adversaries could not easily circumvent if they compromised a single system application. A variety of MAC enforcement mechanisms were introduced, particularly for the Linux operating system [40, 44, 3, 34, 29, 39]. The motivation for such efforts was captured in the paper by Loscocco *et al.* according to which, commercial multiuser systems in the late 1990s were prone to compromise because of their heavy dependence on system applications to manage security [35].

Current cloud platforms parallel traditional multiuser systems in several ways. First, cloud services make all the decisions regarding the security of cloud resources. They authenticate customers and authorize their use of cloud resources. Second, such security decisions are distributed among the services. Keystone controls access to the compute service, the network service controls network resources, and a variety of storage services manage resource access. Third, there is no system security policy that governs how the cloud operates in case some services are compromised. Finally, although services are not `root` (admin) processes, they essentially have full privilege over cloud resources. These privileges can be exploited

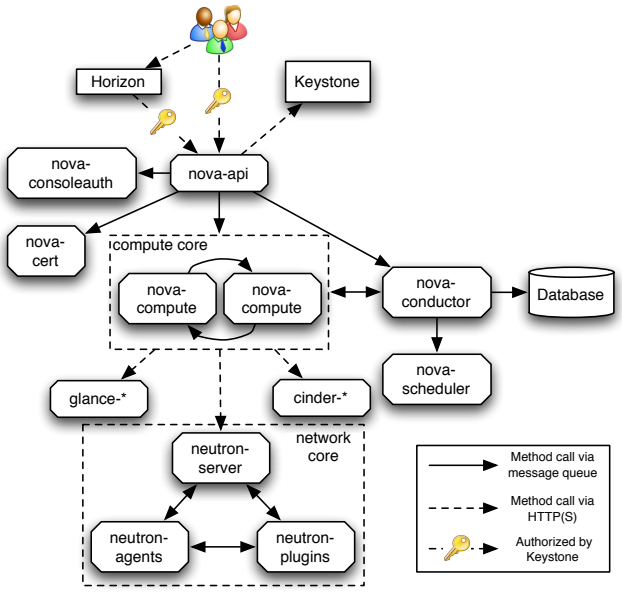

Figure 2: Method calls and caller-callee relationships among OpenStack services.

to launch a variety of attacks against the hosts in which they run. There is mutual trust among services, so any service will respond to any method call from any other service.

It appears that, the OpenStack platform relies too heavily on network controls to protect its services, allowing an adversary free rein to breach those defenses, as was the case in the late 1990s. As a result, we argue that it is time to explore the impact of leveraging MAC enforcement for protecting cloud services. To do this, we propose to estimate the MAC policies for the most common MAC policy models applied in commercial systems. We collect traces of method calls invoked by services, as well as, system calls that are issued by cloud services to process method calls. We use these traces as input to estimate the MAC policies. We then propose a strawman architecture for a "secure cloud operating system" and examine the impact of this architecture on the solution to some difficult security enforcement problems.

3. EXPERIMENTS

We designed a set of experiments to better understand the two problems identified above: (1) how attacks propagate across cloud services and (2) how adversaries leverage vulnerabilities in cloud services to attack hosts. For the first problem, we investigated interactions among cloud services by collecting the method calls generated to perform customer requests (e.g., launching a new instance). For the second problem, we investigated how the cloud services use `sudo` privileges. In both cases, we collected such information using a dynamic analysis, we ran a variety of common cloud operations (boot, delete, suspend, migrate, get-console and create-network) on an OpenStack cloud and collected the traces of system calls executed. The testbed cloud is built from OpenStack Icehouse release with compute core (nova services) and network core (neutron services) installed.

For the first experiment, we studied the interactions between cloud services when performing certain cloud operations. Cloud services interact with each other in two ways, by sending messages over a messaging system or by submitting HTTP(S) requests. Either way, the interactions can be viewed as command invocations where the caller service will send a command call, including a command name and arguments, to the callee service and the callee service will execute that command. The callee service may or may not return responses to the caller services. Figure 2 shows a call graph that captures interactions among cloud services for the cloud operations we performed. Based on the graph, we make the following initial observations. First, not all cloud services need to interact with each other. For example, `nova-api` service does not need to talk to the image service (`glance-*`) or volume service (`cinder-*`) directly. Second, even if two cloud services need to communicate, not all methods that one service supports will be invoked by the other service. For example, from the graph we can see that two `nova-compute` services can send method calls to each other. However, the only scenario that two compute services need to talk is when instances are being migrated.

For the second experiment, we investigated the system calls that cloud services invoked on their hosts, in order to study how adversaries can leverage service vulnerabilities to compromise hosts. For example, an attacker could turn a successful attack on a service into a persistent threat on the host, which would last across service updates and even host reboots. We collected system call traces from a variety of cloud services, while performing cloud operations mentioned above. We want to answer the following questions: (1) which cloud service used additional privileges than a normal application in order to run? (2) If a cloud service requires additional privileges, what are they and how are they being used?

Results are shown in Table 1. For each cloud operation performed, the first number in the table is the total number of system calls issued by the service, and the second is number of system calls issued via `sudo`. [1]

Our first finding is that only few cloud services require root privileges to run. As shown in the Table 1, the majority of cloud services such as `nova-api`, `neutron-server`, and `nova-conductor` do not require root privilege to process the customer requests. These services do not manage instances or instance networks directly; they generally provide support for the cloud infrastructure, such as handling user requests, accessing the database, and so on. On the other hand, services that directly manage instances and instance networks, such as `nova-compute` and `neutron-plugin-agent` require root privilege, as they need to obtain and manage host resources.

Another thing we noticed is that only certain cloud operations require root privilege. Take `nova-compute` as an example. It requires root privilege only for some cloud operations (e.g., launching, destroying, or migrating instances), while not for others (e.g., suspend, get-console). We performed a simple code analysis of `nova-compute` service, and found that 46 out of 75 of its methods involve exercising root privilege[2]. However, the reasons why programs need to be executed via `sudo` are diverse. For example, `nova-compute` sets up the firewall for instances by modifying iptables of the host. This is done through Linux iptables tools (e.g., `iptables-save` and `iptables-restore`), which are only accessible by root. As another example, `neutron-plugin-agent` sets up OVS bridges to connect the vNIC of instances to the physical network via Open vSwitch tools (e.g., `ovs-vsctl` and `ovs-ofctl`). These tools can only be executed by root as well.

[1]Cloud services perform synchronizing tasks periodically. Therefore the number of system calls measured is greater than the actual number of system calls for processing the method call.

[2]Interestingly, there are 75 methods of `nova-compute` in total, but only 64 of them are actually being used by other cloud services.

Table 1: System calls issued on different cloud operations

	boot		delete		suspend		migrate		get-console		create-network	
cloud service	total	sudo	total	sudo	total	sudo	total	sudo	total	sudo	total	sudo
nova-api	21,324	0	13,025	0	11,874	0	28,437	0	10,735	0	125	0
nova-conductor	7,214	0	4,361	0	1,976	0	9,324	0	1,106	0	236	0
nova-consoleauth	278	0	120	0	133	0	176	0	202	0	105	0
nova-compute	120,503	25,861	53,168	10,156	17,389	0	193,832	37,862	979	0	568	0
neutron-server	4,046	0	3,363	0	1,331	0	10,115	0	996	0	6,073	0
neutron-agents (network controller)	1,365	0	1,225	0	850	0	2,041	0	1,068	0	2,231	0
neutron-plugin-agents (compute node)	195,034	34,351	182,826	31,026	153	0	369,275	67,252	144	0	156	0
neutron-plugin-agents (network controller)	32,528	12,721	22,175	12,013	136	0	56,924	27,467	121	0	25,361	11,075

However, there are cases where root privilege is not necessary but still used by a cloud service. For example, before spawning an instance, `nova-compute` will upload customer-specific data to the instance (e.g., customer's SSH keys) by writing the data to the instance image. In this case, no root privilege is necessary as both the data and the instance image are accessible to `nova-compute`. Nevertheless, we found that `nova-compute` still asserts root privilege to perform this operation. What `nova-compute` does is to first mount the instance image to the local filesystem of the host, and then writes data to mounted directory. Mount can only be executed via `sudo` and the data write also requires root privilege because after the mount, the target directory within the instance image (e.g., /root/.ssh) may also be only root accessible. The consequences of unnecessary root privilege can be catastrophic, as now adversary can easily launch a root exploit against the host leveraging vulnerabilities in the services (e.g., directory traversal [20]).

4. MANDATORY ACCESS CONTROL

In this section, we examine the methods that commercial operating system vendors have employed to enforce mandatory access control (MAC). Then we estimate the MAC policies that should be applied to cloud services using the traces of the previous section. Commercial operating system distributors have mainly employed three approaches to mandatory access control: multi-level security [7] (MLS), least privilege [48], and targeted least privilege (typically used for network-facing daemons). The challenge faced by each approach is to prevent all unsafe operations while permitting operations required by the services. As we see, while significant number of vulnerabilities can be blocked by each option, how they deal with conflicts between security and functionality is critical to how effective they are.

4.1 Multi-level Security

The first MAC policy models that were adopted by commercial systems were based on the principle of multi-level security (MLS). A MLS policy model enforces two security properties: (1) the *simple security property* and (2) the **-security property*. The simple security property restricts read operations by subjects, while the *-security property restricts write operations by subjects. The nature of the restriction depends on the policy model, but each MLS policy model enables enforcement of both properties.

The most well-known MLS policy models are Bell-La Padula [7] and Biba [8] models, which protect data secrecy and integrity, respectively. For both policy models subjects and objects are associated with a lattice of security levels [19]. Bell-La Padula model aims to protect data secrecy. Its simple-security property prevents subjects from reading objects of a higher security (secrecy) level (no "read-up") and its *-security property prevents subjects from writing object of a lower security (secrecy) level (no "write-down"). On the other hand, Biba integrity model protects data integrity by enforcing a simple-security property that prevents subjects from reading objects of a lower security (integrity) level (no "read-down") and a *-security property that prevents subjects from writing objects of a higher security (integrity) level (no "write-up"). Note that the lattices of security levels for Bell-La Padula and Biba policies may be independent.

Multi-level security has been applied in commercial operating systems since the Multics system in the 1970s [18]. Since then a few commercial operating systems have also been enhanced to enforce MLS policies [52, 6, 49], although these systems only enforce secrecy protection (i.e., Bell-La Padula only). Other systems that enforce mandatory access control have been extended to enforce MLS as well, such as SELinux [51]. Researchers have explored methods to extend these MLS systems with Biba enforcement features [37, 50], but they have not been adopted in practice. Microsoft did introduce a limited form of the Biba integrity model, the *Biba ring policy* [8], in their User Account Control [38] (UAC). The Biba ring policy (and UAC) only enforces the *-security property, so processes may still read untrusted input.

To estimate the MLS policy for the cloud services, we leverage the method call graph in Figure 2. We find that it will be difficult to enforce either secrecy or integrity in the current OpenStack system. Regarding secrecy, each of the services processes data from all clients, so the operating system could not enforce Bell-La Padula. Note that the services are stateless, so in theory Bell-La Padula could be enforced only if the services are associated with customers and/or requests as we discuss in the following sections. Regarding integrity, it is difficult to pinpoint a practical Biba integrity policy for the OpenStack system. As shown in Figure 2, the `nova-api` service receives untrusted customer requests, but also all the flows necessary to implement requests originate from the `nova-api` service. Presumably, the assumption is that the `nova-api` service is trusted, but it suffers from several vulnerabilities and propagates untrusted inputs that may enable adversaries to reach input validation vulnerabilities. If it is not trusted then `nova-compute`, Keystone, and other services must be trusted to protect themselves from untrusted input.

These observations are consistent with what was observed in traditional multiuser systems [35], where systems often run privileged processes that must communicate with unprivileged processes, providing an avenue for data leakage and use of untrusted data. Even half-measures, such as Windows UAC [38], has been proved too restrictive, at least on initial release. To enable privileged processes to interact with untrusted parties, MLS systems permit some privileged subjects to be designed as *trusted readers and writers*. Such subjects (and the processes run as these subjects) run without the restrictions of MLS. Thus, processes in MLS systems are either contained or trusted to run with no containment. Often many subjects have to be designated as trusted for commercial systems to run properly. In SELinux over 30 subjects were deemed trusted regarding MLS for secrecy [51]. There are no MLS integrity protections in SELinux or other MLS security mechanisms in commercial sys-

tems. Further, we rely on isolation to protect customer instances from one another[3].

4.2 Least Privilege

To address the limitations of MLS enforcement, researchers began to explore alternative MAC policy models based on the notion of *least privilege*. Least privilege was included as one of the principles of computer security in the seminal paper by Saltzer and Schoeder [48], where they stated that "Every program and every user of the system should operate using the least set of privileges necessary to complete the job." The main benefit of least privilege is that if the subjects truly have all the permissions that they need to operate correctly, then there is no need for "trusted" subjects that circumvent the policy. However, least privilege does not enforce any well-defined security property, as MLS policy models do. That is, some of the permissions that a subject may require to function may also be leveraged by adversaries to attack the subject.

The first MAC policy model designed to enforce least privilege was the *type enforcement* model [9], which has variants [5]. Type enforcement associates subjects and objects with labels, as is the case in MLS policy models, but the permissions available to subjects may be assigned flexibly, enabling the specification of permissions independently for each subject. A least privilege policy is typically configured by dynamic analysis of each subject [45, 39, 4]. First, each object is assigned a label. Then, each subject is assigned a label and starts running. The subject label is granted a permission to perform an operation on an object label when dynamic analysis shows that the subject performed that operation on an object of that object label.

In commercial operating systems, least privilege MAC policies are enforced by the SELinux module [36]. Instead of running privileged processes with full privilege (e.g., as "root"), each of these processes are associated with a subject that limits those processes to a set of least privilege permissions. Thus, if an adversary compromises such a process, the permissions available to propagate the attack (e.g., install a rootkit) are limited. While SELinux is still supported by several Linux distributions, the true least privilege policy, called the *reference policy* [54], is only supported by RedHat. Others focus on targeted security as described below.

Using the experimental data in Table 1, we can see that several subjects do not perform `sudo` operations. Thus, presumably we could define a least privilege policy that would restrict these processes from executing privileged processes via `sudo`. Of course, other attack vectors may be present in the system, including launching of setuid processes and modifying resources accessible to privileged processes. Thus, further study is required to identify if any of these permissions are used and assess the safety of such permissions.

The main concern with SELinux policies is their complexity. In theory, the upper bound on the number of policy rules in an SELinux policy is the sum of the number of files accessed for each subject in the system. For a policy governing complete Linux OS distributions, the number of policy rules is order of tens of thousands. However, if we only aim to govern services, we would need policies for each service and each `sudo` and `setuid` process invoked by those services. In OpenStack, for example, a total of 61 programs are launched by compute services via `sudo` and 25 by neutron services. Depending on the operations performed, these programs may be executed multiple times with different parameters (e.g., `ovs-vsctl`). A brief code analysis shows that the number of `sudo` processes that can be invoked by cloud services is approximately 300, counting all cloud operations.

[3]MLS does not address covert information flows, so additional mechanisms are necessary to prevent data leakage due to covert channels.

4.3 Targeted Security

An alternative to maintaining least privilege policies for all the subjects in a system is to focus on blocking the most likely attack paths in the system. This is the aim of the *targeted security* MAC policies. The goal of a targeted policy is to confine those subjects accessible to adversaries to prevent kernel compromise [39].

In principle, a targeted policy requires identification of the subjects targeted for confinement and identification of permissions that could enable kernel compromise. To confine the targeted subjects, the targeted subjects must not be granted that would enable kernel compromise should those targeted subject themselves be compromised. Identifying targeted subjects depends on the threats of concern, such as remote attacks. Subjects accessible to remote adversaries are the typical targeted subjects. Identifying the set of permissions that would enable kernel compromise involves enumerating the set of permissions that could impact kernel integrity, such as the kernel memory, modules, etc. However, we may also want to protect other subjects that are accessible to adversaries that also have these permissions from our targeted subjects.

The targeted policy concept was first employed in a commercial system by the AppArmor Linux module [39]. The AppArmor policy identifies network-facing daemons as the targeted subjects. Since network-facing daemons are accessible to remote parties and historically have run with `root` privileges, it certainly makes sense to explore confinement of those subjects. A SELinux module was also developed to enforce a targeted policy. Chen *et al.* compared two targeted policies [15] by computing the attack paths resulting from these policies, including the attack path for a remote attacker to install a rootkit. They found that AppArmor had 3 attack paths of length one (i.e., compromise one process, which then installs rootkit), whereas SELinux had 6 of such length-one paths. In some cases, the programs accessible to adversaries were not chosen as targets, but in other cases, the daemon simply needed unsafe permissions to run (sshd needs kernel module installation permissions). Other methods, such as *privilege separation* [46], may restrict adversary access to such permissions, but the attack path is still present. We examine other defenses in the next section.

We utilize the experimental data in Table 1 and the method call graph in Figure 2 for estimating the targeted policy for OpenStack services. First, from the method call graph we see that the only network-facing process is *nova-api*, so it is our targeted subject. However, `nova-api` does not have any `sudo` invocations. Thus, our impression, without studying other host attack paths, is that `nova-api` can be confined on its host. However, as we saw in Section 2.2, some root vulnerabilities were caused by `nova-api` propagating malicious requests to other services (`nova-compute`). Thus, the targeted policy is only a first layer of defense when compared to a full least privilege policy, but the targeted policies may be far more manageable.

5. STRAWMAN SCOS ARCHITECTURE

Figure 3 shows a strawman architecture for a SCOS where each cloud node runs an element of the SCOS. Like any operating system, the SCOS manages a set of abstractions for cloud resources, such as instances (processes), volumes (storage), and network (IPC). Also, the SCOS would also provide mechanisms for scheduling use of these resources and securing access to these resources. In theory, the SCOS would support an API analogous to the POSIX API, except that this API would be specified in terms of the SCOS abstractions.

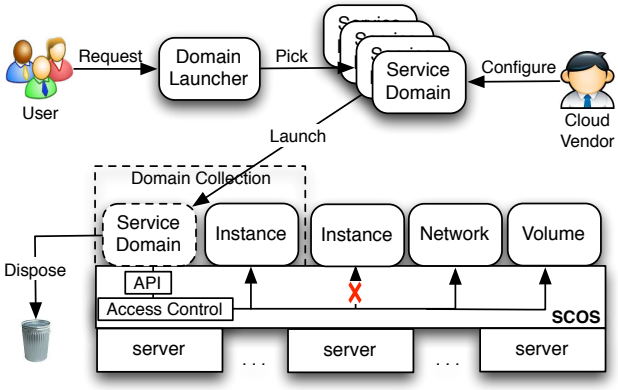

Figure 3: A strawman architecture for Secure Cloud Operating System (SCOS).

Although this strawman SCOS is relatively intuitive on the surface, a few key decisions are evident on a closer examination. This SCOS assumes that instances and services are *protection domains* [48], launched and controlled using the same SCOS mechanisms. Also, the SCOS spans all the hosts in a cloud, but the ability to confine processing by customer and/or request inspires the construction of temporary *domain collections* that partition the protection domains. We briefly discuss the challenges inherent in making these horizontal and vertical cuts below.

We have purposely underspecified the SCOS, but it should be clear that the SCOS API should not refer (directly) to low-level abstractions, such as virtualizing a physical resource or mounting a file system. Such low-level operations will be performed by the SCOS internally, where API provides access to the cloud abstractions built from these resources. However, as we have seen in Section 3, currently cloud services implement method calls through a combination of unprivileged and privileged processes launched by the services, which has caused the line between the OS and services to be blurred in current systems. Thus, the SCOS strawman proposes that a clear *privilege separation* [46] be made between the privileged service processing that operates on OS and virtualization resources and unprivileged processing that operates on cloud resources.

To illustrate how the proposed architecture works, we examine how the request for uploading a customer's SSH key to an instance image would change. Instead of having the service mount the instance file system on the host (a privileged operation), the SCOS would provide the service with an API over an instance file volume and individual resources in that volume. If the service has the permission (for the request) for accessing the customer's instance, the instance's volume, and the volume's key file, then the operation will be authorized. The SCOS chooses how to make the volume available to the service in a secure manner, separating the privileged operations into the SCOS.

A challenge in cloud computing is whether a resource is managed in a centralized or decentralized manner. In OpenStack, a central database maintains all the stateful information. The same problem occurs for security. It appears that researchers are interested in exploring decentralized approaches to security, so the SCOS design aims for decentralization. For example, the self-service clouds [11] (SSC) approach to cloud computing advocates having cloud-specific and customer-specific domains running on the same host. SSC improves the customers' ability to manage their instances by creating their own monitoring without requiring buy-in or introducing security problems for the cloud platform.

For SCOS strawman, we extend this idea by allowing customers to collect a set of domains into domain collections. We would envision that such collections would be associated with a customer and a request. As customer requests for their instances are infrequent, not performance critical, and linearized (at least in OpenStack), such domain collections of services and instances could be created as necessary to implement a specific request. Once the request is completed, then the collection may be disbanded.

6. CLOUD SECURITY POLICY GOALS

While commercial vendors aim to confine target processes that are accessible to adversaries, complete confinement is not practical. Thus, to design a secure cloud operating system (SCOS) we must examine in more detail the security policy goals desired for cloud computing. We focus on two problems: (1) limiting services to the permissions necessary to perform actual requests and (2) protecting services and hosts from compromise due to adversary-controlled inputs when processing requests. In this section, we propose a set of policy goals for the SCOS design and examine the open challenges in achieving those goals. Each of these problems are long-standing challenges in secure operating system design, but perhaps the structured nature of the cloud will enable effective solutions.

6.1 Limiting Request Permissions

The first problem is to prevent attacks that may occur during request processing, even if one or more services becomes compromised. For this we examine two types of attacks. First, each service is potentially capable of misusing permissions over the objects it manages either maliciously or as the result of an error. Customers would like to know that a SCOS ensures that the permissions that services use to process a method call are limited by the requirements of the request. Second, currently every service trusts every method call it receives, so a compromised service may be able to submit malicious method calls to other services. For example, the `nova-api` and Keystone services may not have many resources of their own, but vulnerabilities in those services may enable them to propagate malicious method calls to other services. Further, a compromised service may generate malicious method calls at any time. Customers would like to know that even compromised services must produce method calls in a manner that is compliant with customer requests.

MAC enforcement may be used to limit the permissions of services to only those resources necessary to implement customers' requests. However, in OpenStack, each service performs duties for any customer, so a compromised service may maliciously modify or leak the data of a customer when processing the request of another. Thus, one possible attack is a *confused deputy attack* [30], where one customer uses a service to gain unauthorized access to another customer's or the cloud's resources.

Typically, in order to prevent one process running on behalf of one customer from compromising the resources of a process running on behalf of another customer, the server will create a new process for each request. Then, the permissions of each process may be constrained to those necessary for the customer and request. Fortunately, most OpenStack services are stateless by design, meaning that a new service instance can be created with specialized permissions for each request and/or method call. We envision that the SCOS strawman proposal would be able to control such services.

A problem is that some services are inherently multiuser. For example, the database stores the state of the cloud needed by each

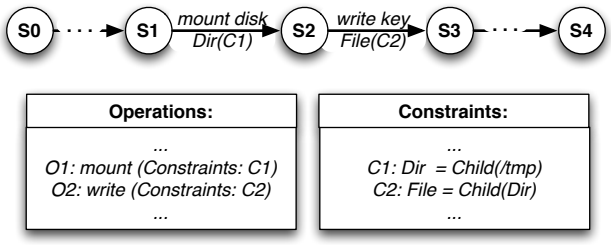

Figure 4: Automaton for run_instance method of nova-compute.

service. However, the proposed SCOS could still identify the domain collection (customer and request) that produced a method call to the database, enabling a multiuser-aware service to reason about confused deputy attacks. However, this is often error-prone. Thus, we will first focus on using the SCOS to limit cloud services to their least privilege, which ideally are the permissions necessary to process each customer request.

Policy Goal 1 (Prevent Confused Deputy Attacks): Limit the processing of each customer request to that customer's permissions and those permissions necessary for the request only.

Using current MAC enforcement, we could also limit the set of method calls that any service could produce. This would prevent a single compromised service from launching arbitrary attacks on a cloud service of its choice. However, cloud services have complex APIs and interactions. The `nova-compute` service can invoke requests on storage, database, and network services, as shown in Figure 2. Thus, a compromised `nova-compute` could submit to method calls referring to the objects of any customers, even ones that do not have instances running on that compute node.

In general, to ensure that a service is only processing a legitimate request, a trusted party could authenticate the request sent by each cloud service as originating from a customer, using traditional techniques for secure communication using the Keystone credentials (e.g., SSL). The problem with this approach is that cloud services deconstruct the customers' high-level requests into several individual method calls to other services, which in turn result in several system calls over system resources. For example, Figure 2 shows that a single request to create an instance results in many method calls among services. Each of these method calls causes a service to retrieve system resources necessary to implement the method (e.g., retrieve system images) and create the new resources (e.g., virtual machine). While this modularity provides flexibility for services to determine how to implement a method call, services implementing such methods cannot check whether the method call was legitimate.

To understand this challenge, we explore leveraging methods to predict the legal sequences of system calls from program code originally used for intrusion detection [58, 27]. These methods propose representing policies as state machines of possible system call sequences, but they have had difficulty predicting the authorized values of system call arguments, leading to mimicry attacks [59]. For the cloud services, we find that such argument values are easily predicted from request input and cloud configuration. As an example, we generated an automaton policy describing the behavior of `nova-compute` from runtime traces. For each method of `nova-compute`, an automaton is generated to prescribe the operations to be performed as the result of the method invocation. The automaton places constraints on `nova-compute` in two ways: 1) the sequence of operations that can be performed by `nova-compute` and 2) arguments used in each operation. Figure 4 shows a simplified automaton for the run_instance method of `nova-compute`. In this simplified example, `nova-compute` can follow only one possible execution path, and each operation on the path has a constrained set of parameters. For example, the mount operation can only mount instance image under the `/tmp` directory, and the write operation can only write to files under the directory just mounted.

Policy Goal 2 (Enforce Request-specific Least Privilege): Limit each service to invoke only the method calls necessary to implement authenticated customer requests.

6.2 Reducing Attack Surfaces

We now examine how we can reduce the attack surface of a service and the host on which it runs. The objectives are to limit the ability of adversaries to compromise the service (which would enable the attacks described in the last section) or the hosts themselves (which would compromise the cloud system).

Limiting the services to invoke only method calls based on the authenticated requests prevents compromised services from generating malicious requests and limiting each service invocation to only the requesting customer's permissions prevents one customer's request from attacking another customer's requests when the service is run per-request.

Even when enforcing these restrictions, attacks against customer instances are still possible. One attack of concern is that a service processing a request for one customer may be compromised if it uses resources controlled by an adversary of that customer. For example, an adversary may control an image booted by the customer's instance. The cloud often allows customers to reuse resources whose provenance is not tracked. The cloud assumes that the customers are capable of choosing resources for their instances. However, customers can make mistakes and choose malicious or poorly configured resources [10]. From a customer's perspective, they should be able to ensure that their instances are loaded from approved images, use customer-defined keys, etc. However, understanding what flexibility is necessary for customers while protecting instances is an open problem.

In addition, a version of the same problem may befall services. Services may also retrieve resources controlled by their adversaries (e.g., customers) when they expect protected resources. For example, the system should be able to guarantee that a service runs using only system libraries. However, an additional challenge is to distinguish resources used by the customer and resources used by the service. This is particularly a problem for multi-user services, as customers may use the service to attack other customers.

Policy Goal 3 (Prevent Unexpected Attack Surfaces): Prevent a service from retrieving resources controlled by unauthorized parties for the particular system call, method call, etc.

Finally, we examine the problem of protecting the host from attack from a compromised service. As we see in Section 3, both the `nova-compute` and Neutron services are capable of launching processes running with full privilege. Thus, these services have the means to easily compromise kernel integrity. However, even if `sudo` capabilities are removed from these services, systems often have several latent vulnerabilities that may allow attacks through *local exploits*.

Thus, our aim is analogous to that of the MAC targeted policy, where services should be confined from having permissions that enable host compromise. However, the same challenges arise here, as services are responsible for making decisions over the cre-

ation and use of security-critical resources, such as VLANs and VMs. As described in Section 3, more than half of the methods in `nova-compute` leveraged root privileges. In the SCOS strawman, these methods would be implemented in the operating system, but significant design effort will be necessary to separate the privileged operations from the unprivileged. Thus, SCOS should further enable confinement, but the design effort will be non-trivial.

Policy Goal 4 (Confine Unprivileged Method Calls): Prevent the processing of unprivileged method calls from using privileged host permissions.

One approach to enforce policy goals 3 and 4 is the Process Firewall [57]. The Process Firewall is a kernel mechanism that protects processes by blocking retrieval of resources that either lead to unexpected attack surfaces or confused deputy attacks [56]. The Process Firewall is able to provide such protection by introspecting into processes to determine whether the resources being retrieved are safe for the program context. For the SCOS, understanding what resoures are unsafe at what times is more complex, as we discuss above, yet the basic Process Firewall mechanism may still be appropriate. We will explore extension of the Process Firewall mechanism to cloud services.

7. CONCLUSION

How to secure cloud computing in existence of vulnerable cloud services is an open problem. In this paper, we analyzed design requirements of a Secure Cloud Operating System (SCOS), arguing that mandatory access controls over cloud services are critical in order to protect cloud computing from vulnerable cloud services. Based on a trace study of permissions actually used by an OpenStack deployment, we then proposed a set of policy goals for the SCOS design and examined open challenges in achieving these goals. We hope that this can stimulate further research into mandatory access control mechanisms and policies in the cloud.

8. REFERENCES

[1] ANDERSON, J. P. Computer security technology planning study. Tech. Rep. ESD-TR-73-51, The Mitre Corporation, Air Force Electronic Systems Division, Hanscom AFB, Badford, MA, 1972.

[2] Apache CloudStack. `http://cloudstack.apache.org/`.

[3] Security starts with your operating system. `http://www.argus-systems.com/home3.shtml`, 2008.

[4] Selinux/audit2allow. http://fedoraproject.org/wiki/SELinux/audit2allow.

[5] BADGER, L., STERNE, D. F., SHERMAN, D. L., WALKER, K. M., AND HAGHIGHAT, S. A. A domain and type enforcement UNIX prototype. In *Proceedings of the 5th USENIX Security Symposium* (1995).

[6] XTS-400 Trusted Computer System, from BEA Systems, 2008. `http://www.baesystems.com/ProductsServices/bae_prod_csit_xts400.html`.

[7] BELL, D. E., AND LAPADULA, L. J. Secure computer system: Unified exposition and Multics interpretation. Tech. Rep. ESD-TR-75-306, Deputy for Command and Management Systems, HQ Electronic Systems Division (AFSC), L. G. Hanscom Field, Bedford, MA, March 1976. Also, MITRE Technical Report MTR-2997.

[8] BIBA, K. J. Integrity considerations for secure computer systems. Tech. Rep. MTR-3153, MITRE, April 1977.

[9] BOEBERT, W. E., AND KAIN, R. Y. A practical alternative to hierarchical integrity policies. In *Proceedings of the 8th National Computer Security Conference* (1985).

[10] BUGIEL, S., NÜRNBERGER, S., PÖPPELMANN, T., SADEGHI, A., AND SCHNEIDER, T. AmazonIA: When elasticity snaps back. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)* (2011), pp. 389–400.

[11] BUTT, S., LAGAR-CAVILLA, H. A., SRIVASTAVA, A., AND GANAPATHY, V. Self-service cloud computing. In *Proceedings of the 2012 ACM conference on Computer and communications security* (New York, NY, USA, 2012), CCS '12, ACM, pp. 253–264.

[12] CVE-2012-3542. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-3542`.

[13] CVE-2012-4456. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-4456`.

[14] CVE-2014-0167. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0167`.

[15] CHEN, H., LI, N., AND MAO, Z. Analyzing and comparing the protection quality of security enhanced operating systems. In *NDSS* (2009).

[16] Cloudlinux. `http://www.cloudlinux.com/`.

[17] Oracle solaris 11. `http://www.oracle.com/us/products/servers-storage/solaris/solaris11/overview/index.html/`.

[18] CORBATÓ, F. J., AND VYSSOTSKY, V. A. Introduction and overview of the multics system. In *Proceedings of the November 30–December 1, 1965, Fall Joint Computer Conference, Part I* (New York, NY, USA, 1965), AFIPS '65 (Fall, part I), ACM, pp. 185–196.

[19] DENNING, D. A Lattice Model of Secure Information Flow. *Communications of the ACM 19*, 5 (1976).

[20] CVE-2012-3360. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-3360`.

[21] CVE-2012-3361. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-3361`.

[22] CVE-2012-3447. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-3447`.

[23] CVE-2013-0247. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-0247`.

[24] CVE-2013-0270. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-0270`.

[25] CVE-2014-2828. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-2828`.

[26] Eucalyptus. `https://www.eucalyptus.com/`.

[27] GIFFIN, J. T., JHA, S., AND MILLER, B. P. Detecting manipulated remote call streams. In *Proceedings of the 11th USENIX Security Symposium* (Berkeley, CA, USA, 2002), USENIX Association, pp. 61–79.

[28] CVE-2014-0162. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0162`.

[29] HALLYN, S. E., AND KEARNS, P. Domain and type enforcement for Linux. In *Proceedings of the 4th Annual Linux Showcase and Conference* (Oct. 2000). At `http://www.sagecertification.org/publications/library/proceedings/als00/2000papers/papers/full_papers/hallyn/hallyn_html/index.html`.

[30] HARDY, N. The confused deputy. *Operating Systems Review 22*, 4 (Oct. 1988), 36–38.

[31] CVE-2011-4596. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-4596.

[32] CVE-2013-0208. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-0208.

[33] CVE-2014-0187. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0187.

[34] LIDS Secure Linux System. http://www.lids.org/, 2008.

[35] LOSCOCCO, P. A., SMALLEY, S. D., MUCKELBAUER, P. A., TAYLOR, R. C., TURNER, S. J., AND FARRELL, J. F. The Inevitability of Failure: The flawed assumption of security in modern computing environments. In *Proceedings of the 21st National Information Systems Security Conference* (October 1998), pp. 303–314.

[36] MAYER, F., MACMILLAN, K., AND CAPLAN, D. *SELinux by Example: Using Security-Enhanced Linux*. Addison-Wesley, 2006.

[37] MCILROY, M. D., AND REEDS, J. A. Multilevel security in the UNIX tradition. *Software–Practice and Experience 22* (1992), 673–694.

[38] NARAINE, R. Russinovich: Malware will thrive, even with Vista's UAC, April 2007. http://blogs.zdnet.com/security/?p=175.

[39] AppArmor Linux application security. http://www.novell.com/linux/security/apparmor/, 2008.

[40] Security-enhanced linux. http://www.nsa.gov/selinux.

[41] OpenNebula. http://opennebula.org/.

[42] OpenStack Open Source Cloud Computing Software. http://www.openstack.org//, 2008.

[43] Openstack api quick start. http://docs.openstack.org/api/quick-start/content/.

[44] OTT, A. Rsbac: Extending Linux security beyond the limits. http://www.rsbac.org/, 2008.

[45] PROVOS, N. Improving host security with system call policies. In *Proceedings of the 2003 USENIX Security Symposium* (August 2003).

[46] PROVOS, N., FRIEDL, M., AND HONEYMAN, P. Preventing privilege escalation. In *Proceedings of the USENIX Security Symposium* (Aug. 2003).

[47] 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 *Proceedings of the 16th ACM conference on Computer and communications security* (2009).

[48] SALTZER, J. H., AND SCHROEDER, M. D. The protection of information in computer systems. *Proceedings of the IEEE 63*, 9 (September 1975).

[49] SCHELL, R., TAO, T., AND HECKMAN, M. Designing the GEMSOS security kernel for security and performance. In *Proceedings of the National Computer Security Conference* (1985).

[50] SCHELLHORN, G., REIF, W., SCHAIRER, A., KARGER, P. A., AUSTEL, V., AND TOLL, D. Verification of a formal security model for multiapplicative smart cards. In *Proceedings of the European Symposium on Research in Computer Security* (2000), pp. 17–36.

[51] Selinux/mls. http://fedoraproject.org/wiki/SELinux/MLS.

[52] SUN MICROSYSTEMS. Trusted Solaris 8 Operating System. http://www.sun.com/software/solaris/trustedsolaris/, Feb. 2006.

[53] CVE-2012-4406. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-4406.

[54] Reference Policy. http://oss.tresys.com/projects/refpolicy, 2008.

[55] VARADARAJAN, V., KOOBURAT, T., FARLEY, B., RISTENPART, T., AND SWIFT, M. M. Resource-freeing attacks: improve your cloud performance (at your neighbor's expense). In *ACM Conference on Computer and Communications Security* (2012), pp. 281–292.

[56] VIJAYAKUMAR, H., GE, X., PAYER, M., AND JAEGER, T. JIGSAW: Protecting resource access by inferring programmer expectations. In *Proceedings of the 23rd USENIX Security Symposium* (Aug. 2014), pp. 973–988.

[57] VIJAYAKUMAR, H., SCHIFFMAN, J., AND JAEGER, T. Process Firewalls: Protecting processes during resource access. In *Proceedings of the Eighth ACM European Conference on Computer Systems (EuroSys)* (2013), pp. 57–70.

[58] WAGNER, D., AND DEAN, D. Intrusion detection via static analysis. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2001), SP '01, IEEE Computer Society, pp. 156–.

[59] WAGNER, D., AND SOTO, P. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2002), CCS '02, ACM, pp. 255–264.

[60] WATSON, R. N. M. TrustedBSD: Adding trusted operating system features to FreeBSD. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference* (2001), pp. 15–28.

[61] WRIGHT, C., COWAN, C., SMALLEY, S., MORRIS, J., AND KROAH-HARTMAN, G. Linux Security Modules: General security support for the Linux kernel. In *Proceedings of the 11th USENIX Security Symposium* (August 2002), pp. 17–31.

[62] ZHANG, F., CHEN, J., CHEN, H., AND ZANG, B. Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 203–216.

[63] ZHANG, Y., AND REITER, M. K. Düppel: retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *ACM Conference on Computer and Communications Security* (2013).