Secure Logical Isolation for Multi-tenancy in Cloud Storage

Michael Factor*, David Hadas*, Aner Hamama*, Nadav Har'el*, Elliot K. Kolodner*,
Anil Kurmus†, Alexandra Shulman-Peleg*, Alessandro Sorniotti†
* IBM Haifa Research Lab
† IBM Zurich Research Lab

Abstract— **Storage cloud systems achieve economies of scale by serving multiple tenants from a shared pool of servers and disks. This leads to the commingling of data from different tenants on the same devices. Typically, a request is processed by an application running with sufficient privileges to access any tenant's data; this application authenticates the user and authorizes the request prior to carrying it out. Since the only protection is at the application level, a single vulnerability threatens the data of all tenants, and could lead to cross-tenant data leakage, making the cloud much less secure than dedicated physical resources. To provide security close to physical isolation while allowing complete resource pooling, we propose Secure Logical Isolation for Multi-tenancy (SLIM). SLIM incorporates the first complete security model and set of principles for the safe logical isolation between tenant resources in a cloud storage system, as well as a set of mechanisms for implementing the model. We show how to implement SLIM for OpenStack Swift and present initial performance results.**

1 INTRODUCTION

Cloud systems achieve economies of scale by serving multiple customers from a shared pool of resources [1], [2]; each customer (which could be a company or enterprise) is a *tenant* of the cloud infrastructure. Physical resource pooling enables load balancing, homogeneity for management and much higher utilization rates. Sharing of physical resources is important for compute clouds as well as storage-centric clouds. In storage-centric clouds, the pooled resources include the physical media and the servers controlling the media. In a cloud where all physical resources are pooled, any given device may have data from multiple, unrelated tenants.

A major concern expressed by many businesses over moving to a public cloud delivery model is security [3]. This concern stems from the commingling of the data of different tenants on shared physical resources.

It is common for cloud storage systems to provide application-level security, in which components that authenticate and process user requests run with sufficient privileges to access any tenant's data; the code of each component is responsible for authorizing requests based on the requester's credentials. This architecture is used by OpenStack Swift [4] and other publicly available cloud storage systems.

Application-level security only provides a single level of defense; a vulnerability that allows bypassing the security check, such as a confused deputy attack [5], can compromise all data stored in the cloud. This is very weak isolation. If each tenant had its own segregated physical resources, it would be much less likely that a single vulnerability could jeopardize all tenants' data.

To provide security similar to physical isolation while allowing complete pooling of resources, we propose Secure Logical Isolation for Multi-tenancy (SLIM). SLIM adds an orthogonal tenant isolation mechanism over existing application-level security by leveraging the Linux process isolation mechanisms that have been thoroughly tested for over 20 years by the Linux community and enhanced with mechanisms such as SELinux. SLIM therefore enables resource pooling while decreasing the likelihood that a single vulnerability could jeopardize all tenants data.

In particular, SLIM provides this additional isolation across tenants by following the principle of least privilege [6]: each system component runs with the least set of privileges required to complete its task. Moreover, such privileges are designed to be tenant-specific: for example, we define separate privilege classes to access authentication material of tenant A and tenant B. As a consequence, whenever possible, SLIM contains breaches within a tenant by leveraging process isolation. For the remaining components, which need to be trusted to operate on data of multiple tenants, we minimize their attack surface.

Employing these principles in an efficient manner is difficult. First, each request must be correctly associated with a tenant and properly split into appropriate sub-pieces each of which is processed under the principle of least privilege [7]. For instance, we use different privileges for authenticating a user and for accessing a disk.

Second, we need to address security as we move between components in the cloud implementation, either between different processes implementing different parts of the function on a single server or communicating with a different server, e.g., for data replication. In both cases we need to correctly track the tenant identity and use the appropriate privilege.

Finally, in real world systems we may need to use existing resources like a distributed cache (e.g., memcached [8]) or a shared data store (e.g., Cassandra [9]) that do not implement all of our principles. While we cannot provide the same degree of isolation here as we provide in the rest of the system, we still want to maximize the degree of secure logical isolation between data of different tenants.

SLIM addresses these issues using the following mechanisms:

- a security gateway ensures a request is handled with the right tenant-specific privilege

The research leading to these results is partially supported by the European Community's Seventh Framework Programme (FP7/2001-2013) under grant agreement n 257019 - VISION Cloud Project.

- a proxy/guard mechanism prevents escalation of privileges as we move between components
- and a gatekeeper provides a secure wrapper for existing resources

These SLIM components are relevant for almost any cloud based object store system, and can be easily integrated to provide end-to-end isolation between tenants. Here, we describe an implementation of SLIM for OpenStack Swift; we are also implementing it for the VISION Cloud project [10].

Our paper makes the following contributions:

- We present the first complete security model and set of principles for safe logical isolation between tenant resources in a cloud storage system.
- We define a set of mechanisms for implementing secure, logically isolated cloud storage systems.
- We implement SLIM for Swift and present initial performance results, determining that process recycling is the most critical factor influencing performance.

The next section describes various approaches to security isolation for multi-tenancy and provides background on Swift. Section 3 presents SLIM and its implementation in Swift. Section 4 presents initial performance results. Section 5 presents an informal analysis of SLIM's security. Section 6 presents our conclusions.

2 Problem statement

Cloud storage systems come in multiple flavors that differ in their abstraction, ranging from block storage systems, to key-value and object stores; in this work, we select object stores as our case-study. Although the internal architecture of such services is rarely disclosed, we can identify a number of common features. First, services run over commodity hardware. Second, they are symmetric and decentralized: there is no hierarchy among the pool of machines that provide the service, and such machines run essentially the same code. More significantly in our context, these systems commingle data from different users on the same physical resources. This approach is often referred to as multi-tenancy.

Ideally a multi-tenant cloud storage system serves requests of multiple customers (tenants) in such a way that (1) computing and storage resources are shared among such customers and (2) this sharing of resources does not weaken system security. In practice, multi-tenancy is a trade-off between security and costs: the wider the subset of resources shared (e.g., same physical machine vs. same OS), the more the cloud system can amortize costs and increase utilization. However, this sharing leads to weaker isolation and consequently higher security risks.

In a typical object store architecture a client contacts a web front-end to issue a request. The front-end passes the request to a request processor, which accesses (e.g., read or write) the data usually through a file system on disks. The request processor may also access a supplementary data store, such as a distributed key/value store. It also performs security-related tasks like authentication, authorization and access control enforcement. Usually, the front end authenticates and authorizes a request, prior to executing the request under a single cross-tenant privilege.

As a concrete example for our analysis we consider Swift, the popular open-source, cloud object store, which is part of the Openstack Framework. Swift has a two tier architecture, consisting of client facing proxy servers, which handle authentication, authorization and access control enforcement, and storage servers which store objects and manage the associated metadata. A typical installation will consist of multiple proxy and storage servers possibly sharing the same hardware. Each Swift server (proxy or storage) is coupled with an integrated HTTP server, employing WSGI and serving REST requests via a dedicated port. A client request may be addressed to any of the proxy servers, which forwards it to the appropriate storage server(s).

Swift, like most other cloud storage systems, employs application-level isolation to address multi-tenancy and the entire service runs with a single privilege level, allowing access to all tenant and user data from the Swift code. Swift enforces isolation through pluggable access control modules such as Keystone or tempAuth. Swift supports container ACLs, which are retrieved from a storage server and then cached by the proxy using memcached [8], allowing their fast retrieval. While this approach makes Swift lightweight and able to serve a large number of tenants, it has the negative impact that finding and exploiting a single vulnerability gives an attacker the ability to access resources belonging to any user of any tenant. Furthermore, it is hard to prevent attacks on a given Swift component, since it commingles information for different tenants, including the use of memcached to store sensitive access control data.

3 Secure Logical Isolation for Multi-tenancy

SLIM addresses the weaknesses highlighted in the previous Section, incorporating a security model and set of principles for the safe logical isolation between tenant resources for a cloud storage system, as well as a set of mechanisms for implementing the model.

3.1 System model

SLIM provides services for a number of entities that we call *users*. Users board the system through an enrollment process that equips them with credentials, which can be later on produced to gain access to the system and its resources. Users are grouped into administrative entities called *tenants*. In what follows we only capture simple hierarchies where there is a set of users, a set of tenants and each user is associated to a single tenant; however, SLIM can be generalized to more complex n-level hierarchies involving sub-tenants.

Users interact with a set of service nodes that can serve requests of any user belonging to any tenant; in particular, we do not require dedicated hardware for each tenant, which would undoubtedly achieve comparable, if not better, security guarantees at a much higher total system cost. Service nodes run a commodity operating system and do not require virtualization techniques. Each service node may use local storage and/or shared storage to host user data/metadata.

3.2 Security model

We consider two classes of attackers. In both cases, the attacker is a system user, i.e., the attacker owns a valid credential to access the system. *Class 1* attackers attempt to access resources belonging to another user of the same tenant, whereas *class 2* attackers attempt to access resources belonging to the user of another tenant.

We do not assume dedicated hardware or dedicated virtual machines for given users/tenants; however, we do trust the underlying hardware, as well as the *multi-user* operating systems to provide secure isolation between processes of different *OS users*; in the rest of the paper, we will use *uid* to refer to an OS user. This requirement translates into the assumption that the process of one uid cannot access resources (e.g., memory, files, sockets) belonging to another uid. If a process is compromised by an attacker, we assume the attacker can run arbitrary code under the uid that started the process.

The assumption above does not take into account the fact that some of the tasks that the service nodes need to perform require higher privileges than those of a single uid, e.g., admin privileges to create processes restricted to a particular uid. A process with such privileges, is called a *privileged process* and is by definition capable of accessing any resource in the system and can therefore bypass the uid restrictions. As a consequence, if an attacker can compromise a privileged process, it can achieve the objectives of both attacker classes.

Finally, nodes partake in a number of internal protocols, through which a node can trigger the execution of a function on another node on the system (e.g., trigger the replication of a file from one node onto another one). We therefore need to ensure that a successful attack – which by the assumptions mentioned above should be confined to the privileges of the owner uid only – cannot trigger the execution of a function with higher privileges on another node.

3.3 Design Principles

The design of our solution was guided by the following four principles. (1) *Least privilege* requires that every subcomponent operate using the least set of privileges required for its task. A consequence of this principle is the need to use separate processes with different uids for handling the various stages of a user request. (2) *Tenant containment* requires security isolation of tenant-related resources, i.e., each tenant has its own privileges and hence its own uids. Thus, there is the need to use different processes for each tenant. (3) *Escalation avoidance* ensures that during the lifetime of a process it will never gain a different, potentially higher privilege. This implies that if a process with a tenant's UID has been used to serve one tenant's requests, it cannot be reused later to process another tenant's request. (4) *Minimizing the attack surface* keeps the code of any privileged processes simple, small and easy to audit.

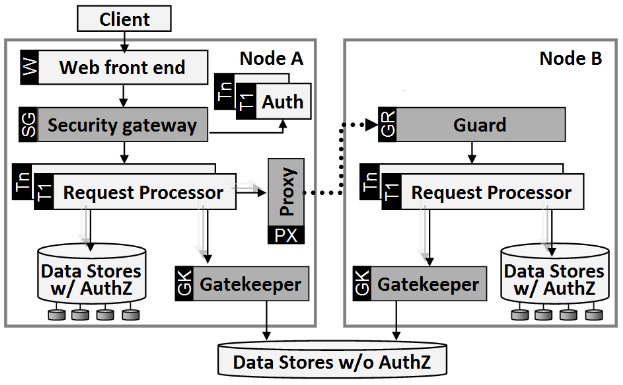

Fig. 1. The architectural components of SLIM (dark gray) added to a typical object store architecture (light gray).

3.4 Design and Implementation

Complying with the principles described above requires introducing end-to-end tenant isolation in the cloud storage system, where we isolate both the processing and the access to all tenant related information stored on the disks and in supplementary data stores. The architectural elements of SLIM are illustrated in Figure 1. They include the following privileged processes: (1) a *Security Gateway*, which splits the execution of a request into subtasks running under the proper uids with confined privileges; (2) a *Gatekeeper*, which protects the access to shared resources without built-in access control, e.g., a key-value store; and (3) *Guard and Proxy* components to maintain tenant identity and privileges across processes running on the same or different nodes. The remaining processes are tenant-specific: (4) a *Tenant Authenticator* to authenticate the tenant named in a request; and (5) a *Request Processor* to execute the request. Below we detail the design of these components as well as their implementation over Swift.

Security Gateway: A user request first arrives at the web front end, which delivers it to a security gateway. The security gateway introduces privilege separation by splitting the execution of a request into subtasks, executing each subtask under a dedicated uid corresponding to the required privilege of the specific tenant. It begins by sanitizing the request and verifying the validity of its parameters. It extracts the authentication credentials from the request, e.g., from the HTTP headers, and passes them to a tenant authenticator process, which authenticates the tenant claimed in the request. Upon approval from the authenticator, the security gateway delivers the request to an appropriate request processor that has the privilege of the tenant. To prevent an attacker from tampering with requests, SLIM requires that requests be protected, e.g., signed, so the first action of the request processor is to authenticate the request. Mechansims for protecting requests are outside of the scope of SLIM.

Gatekeeper for shared resource protection: In some cases a request processor may need to access a shared resource. This can be a distributed data store, like Cassandra [9], e.g., used as a metadata catalog by the VISION Cloud, or a shared cache like memcached [8], e.g., used by Swift, which do not have built-in, fine-grained access control mechanisms, yet are used to store data of multiple tenants. Despite the potential security threats they can not always be replaced by traditional data stores like RDBS, that provide access control, yet may limit performance and scalability.

Each such resource has its own gatekeeper process. For each request, the gatekeeper verifies the true tenant identity of the process requesting access to the resource. Once the gatekeeper has uniquely identified the tenant, its goal is to protect the corresponding resource (e.g., keys and values) in a way that it will be inaccessible to other tenants. To isolate the keys under which the data is stored, the gatekeeper labels them with a unique tenant identifier. When integrity and confidentiality are also important, this key may be cryptographically signed or encrypted with a unique key belonging to the tenant. The values stored under these keys should also be signed or encrypted according to the selected level of protection. To prevent any backdoor attacks, all data store access requests that do not originate from the gatekeeper are blocked (e.g., by protecting the socket with ACLs). This allows the gatekeeper to isolate views of the shared resource that each tenant has, ensuring that each tenant has access only to its own keys and values. This prevents cross-tenant data leakage and malicious modifications of the stored keys and values.

Guard and Proxy to protect inter-node communication: In some cases a request processor (a first process) may need to assign a task to a second process, possibly on another node. This is done through a guard and a proxy that ensure the identity of the tenant is maintained when the task is executed by the second process. The request to the second process must go through a proxy. Furthermore, the second process may execute the task only if requested by a guard. Guards and proxies maintain a trusted communication channel (e.g., using a VPN or privileged ports in a well controlled environment). These restrictions are applied by using operating system access control and firewall rules.

SLIM maintains the tenant identity and privilege between the request processor and the second process using the following three stages: (1) The proxy, which runs on the same node as the first process, extracts the true set of privileges of the first process (e.g., by using OS-level primitives such as SCM_CREDENTIALS). (2) The proxy sends a description of the privileges together with the request of the first process to the guard, which resides on the same node as the second process. (3) The guard delivers the request to the second process that has the appropriate set of privileges.

4 Performance analysis

We conducted performance tests which evaluated the penalty of adding SLIM to an object storage system of Swift. We compared throughput for `PUT` and `GET` requests of objects of various size (on a 8-core, 2.33GHz, x86 server with 16GB RAM, having Swift's SAIO configuration with 9 disks divided into 3 zones). For each combination of operation and object size, we ran *swift-bench* for five minutes. To select the best configuration in which the resources are maximally utilized, we monitored the CPU and disk utilization. We observed that *swift-bench* with request concurrency of 15 gave almost 100% CPU utilization for SLIM, while for vanilla Swift, it was maximized at a concurrency of 150.

Not surprisingly, vanilla Swift has the highest performance, which was especially noticeable for small objects. For example, for 16KB objects it supports the throughput of 4.8 and 16.8 MB/s for `PUT` and `GET` operations respectively. However, vanilla Swift recycles processes between tenants which can lead to cross-tenant leakage. We observed that using the apache server with CGI for creating a fresh process for executing each tenant request leads to severe performance degradation, supporting the throughput of only 0.08 and 0.3 MB/s for `PUT` and `GET` of 16K objects respectively. Thus, we conclude that process recycling is the most critical factor influencing the performance and we are working on its optimization. The Security Gateway and the Gatekeeper were not observed to influence performance and can be safely added to achieve multi-tenant isolation. Interestingly, when the object size was increased the differences between the systems became less noticeable as both vanilla Swift and SLIM saturated the network, which became the major bottleneck.

5 Security analysis

The security model presented in this work considers two types of attacks: attacks within a tenant (class 1) and cross-tenant (class 2). Possible attacks can target vulnerabilities in the web front end, request processor, and privileged components such as the kernel, data stores lacking authorization mechanism, security gateway, proxy and guard, and gatekeeper. The attacks we consider here range from confused deputy attacks to full compromise of a process with arbitrary code execution. The attackers goal is either to access another user's data from the same tenant (class 1) or other tenant data (class 2).

In this section we show that SLIM protects against class 2 attacks and decreases the likelihood that a single vulnerability could jeopardize all tenants' data. Figure 2 illustrates the defense line added by SLIM to achieve this goal as we detail below.

An imminent risk to the system could come from a potential vulnerability in the web front end or the request processor as these components are both directly exposed to an attacker and are large software packages, such as Apache and Swift. To duly protect tenant data, we assume such vulnerabilities exist and either the web front end and/or the request processor could be fully compromised. SLIM ensures that even under such a compromise, tenant data remains secure by: (1) restricting a request processor, which is privileged to access tenant-data, to a tenant-specific UID by the security gateway as well as the proxy/guard components when inter-node communication is involved; (2) restricting a tenant-specific authenticator to a tenant-specific UID; and (3) restricting access to data stores lacking authorization mechanisms through the gatekeeper.

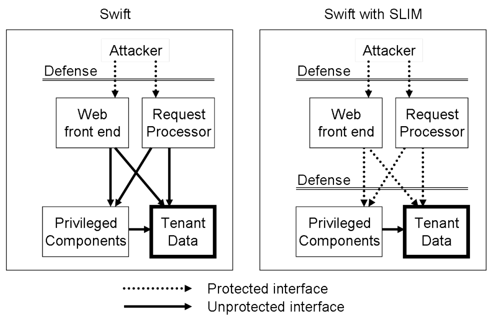

Fig. 2. The defense line added by SLIM protects tenants using Swift from unauthorized access to their data when the Swift request processor and/or web front end is compromised.

Below, we detail the attacks SLIM protects against by examining the cases where various components are either compromised and used as a stepping stone for attacking further modules, or are under a confused deputy attack.

Web front end: A compromised web front end cannot access any tenant data directly since it is not privileged to access the data stores. However given that TLS termination occurs within this component, an attacker could mount a man-in-the-middle attack against any tenant, potentially accessing that tenant's data. Yet, the attack is limited in time (the attacker can only intercept requests occurring within the timeframe of the compromise). Additionally, the system may authenticate the requests (e.g., using signatures) and test for authenticity at the request processor; in this way the attacker can be prevented from tampering with the requests.

Request processor: In case a request processor is compromised, the attacker's process can only access the corresponding tenant-data, and is limited to performing tenant-specific queries through the security gateway and proxy. This does not prevent the attacker from accessing data from another user of the same tenant, but confines the attack within the tenant.

Data stores lacking authorization mechanism: In case the data store is compromised, the attacker could mount attacks that will lead to the compromise of all tenants' data. However, because the interface to the data store is highly restricted by the gatekeeper, which only exposes a simple interface, such an event is less likely.

Kernel: If local access is obtained, this could put the attacker in a better position to perform further attacks to elevate his privileges. In particular, the Linux kernel exposes a very large attack surface to local attackers [11], [12]. Although this is out of the scope of our work, this attack surface can be greatly reduced (see for example [13], [14], [15], [11]). We note that the request processor and authenticators do not have network access (this is enforced by the use of the proxy as well as host-firewall rules), hence preventing network-based attacks being used for privilege elevation.

Security gateway, the gatekeeper, or the proxy/guard: Similarly, the attacker could target one of the remaining privileged processes: the security gateway, the gatekeeper, or the proxy/guard. However, their very small size (less than 500 LOC) makes them easy to audit, making such attacks unlikely.

6 Conclusion

We presented SLIM, an end-to-end approach to tenant isolation in a multi-tenant cloud storage system that allows all resources to be shared. SLIM shows how to adapt a cloud storage architecture to incorporate well-established security principles such as least privilege and privilege separation. SLIM addresses the new architectural requirements posed by these principles in the context of cloud storage solutions, where in principle any action taken on behalf of tenant should be executed in a process with a tenant-specific privilege. We contrast this with typical cloud storage systems where the only mechanism for isolation is provided by the application-level cloud implementation.

References

[1] P. Mell and T. Grance, "The NIST definition of cloud computing," National Institute of Standards and Technology, Information Technology Laboratory, Tech. Rep., July 2009.

[2] Y. Chen, V. Paxson, and R. H. Katz, "What's new about cloud computing security?" EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2010-5, Jan 2010.

[3] E. Davis, P. Fersht, and E. Herrera, "Cloud will transform business as we know it: the secret's in the source," HfS Research, Tech. Rep., December 2010.

[4] "Welcome to Swift's documentation!" September, 2012. [Online]. Available: http://http://docs.openstack.org/developer/swift/

[5] N. Hardy, "The confused deputy: (or why capabilities might have been invented)," *SIGOPS Oper. Syst. Rev.*, vol. 22, no. 4, pp. 36–38, Oct. 1988.

[6] J. Saltzer and M. Schroeder, "The protection of information in computer systems," in *Proceedings of the 4th ACM Symposium on Operating System Principles*, October 1973.

[7] N. Provos, "Preventing privilege escalation," in *Proceedings of the 12th USENIX Security Symposium*, 2003, pp. 231–242.

[8] B. Fitzpatrick, "Distributed caching with memcached," *Linux J.*, vol. 2004, no. 124, pp. 5–, Aug. 2004.

[9] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *Operating Systems Review*, vol. 44, no. 2, p. 35, 2010.

[10] E. K. Kolodner, *et al.*, "A cloud environment for data-intensive storage services," in *Proceedings of the 2011 IEEE Third International Conference on Cloud Computing Technology and Science*, ser. CLOUDCOM '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 357–366.

[11] H. Chen, Y. Mao, X. Wang, D. Zhou, N. Zeldovich, and M. F. Kaashoek, "Linux kernel vulnerabilities: state-of-the-art defenses and open problems," in *Proceedings of the Second Asia-Pacific Workshop on Systems*, ser. APSys '11. New York, NY, USA: ACM, 2011, pp. 5:1–5:5.

[12] A. Kurmus, R. Tartler, D. Dorneanu, B. Heinloth, V. Rothberg, A. Ruprecht, , W. Schröder-Preikschat, D. Lohmann, and R. Kapitza, "Attack Surface Metrics and Automated Compile-Time OS Kernel Tailoring," in *Proceedings of the 20th Network and Distributed System Security Symposium*, 2013, (to appear).

[13] A. Kurmus, A. Sorniotti, and R. Kapitza, "Attack Surface Reduction for Commodity OS Kernels," in *Proceedings of the Fourth European Workshop on System Security*, ser. EUROSEC '11. New York, NY, USA: ACM, 2011, pp. 6:1–6:6.

[14] "Overview of the seccomp sandbox," 2009. [Online]. Available: http://code.google.com/p/seccompsandbox/wiki/overview

[15] R. Tartler, A. Kurmus, A. Ruprecht, B. Heinloth, V. Rothberg, D. Dorneanu, R. Kapitza, W. Schröder-Preikschat, and D. Lohmann, "Automatic OS kernel TCB reduction by leveraging compile-time configurability," in *Proceedings of the Eighth Workshop on Hot Topics in System Dependability*, ser. HotDep '12, 2012.