Towards Access Control for Isolated Applications

Kirill Belyaev and Indrakshi Ray
Computer Science Department, Colorado State University, Fort Collins, U.S.A

Keywords: Access Control, Service and Systems Design, Inter-application Communication.

Abstract: With the advancements in contemporary multi-core CPU architectures, it is now possible for a server operating system (OS), such as Linux, to handle a large number of concurrent application services on a single server instance. Individual application components of such services may run in different isolated runtime environments, such as chrooted jails or application containers, and may need access to system resources and the ability to collaborate and coordinate with each other in a regulated and secure manner. We propose an access control framework for policy formulation, management, and enforcement that allows access to OS resources and also permits controlled collaboration and coordination for service components running in disjoint containerized environments under a single Linux OS server instance. The framework consists of two models and the policy formulation is based on the concept of policy classes for ease of administration and enforcement. The policy classes are managed and enforced through a Linux Policy Machine (LPM) that acts as the centralized reference monitor and provides a uniform interface for accessing system resources and requesting application data and control objects. We present the details of our framework and also discuss the preliminary implementation to demonstrate the feasibility of our approach.

1 INTRODUCTION

The advancements in contemporary multi-core CPU architectures have greatly improved the ability of modern server operating systems (OS) such as Linux to deploy a large number of concurrent application services on a single server instance. The emergence of application containers (Docker Developers, 2016), introduction of support for kernel namespaces (Manual, 2016) allows a set of loosely coupled service components to be executed in isolation from each other and also from the main operating system. This enables application service providers to lower their total cost of ownership by deploying large numbers of application services on a single server instance and possibly minimize horizontal scaling of applications across multiple nodes.

In conventional UNIX or Linux OS, applications can be deployed in isolated (containerized) environments, such as chrooted jails. Such isolated environments limit the access of the applications beyond some designated directory tree and have the potential to offer enhanced security and performance. However, no mechanism is provided for controlled communication and sharing of data objects between isolated applications across such environments. A new type of access control is needed for such isolated applications for controlling access to OS resources, for regulating the access requests to the application data and control objects used for inter-application communication between service components.

For instance, consider a real-world service deployment scenario taken from the actual telecom service provider (n–Logic Ltd., 2016). A Linux server has three applications, namely, *Squid Web Cache Server*, *Squid Log Analyzer*, and *HTTP Web Server*, deployed in three separate isolated environments (chrooted jail directories), each under a distinct unprivileged user identifier (UID). Combined all three applications represent individual components of a single service – ISP web caching that caches Internet HTTP traffic of a large customer base to minimize the utilization of ISP's Internet backbone. *Squid Web Cache Server* component needs access to network socket I/O resources and some advanced networking capabilities of the Linux kernel in order to operate. However, we cannot allow this component to run under a super-user to obtain unrestricted root privileges. Allowing root access to an application opens possibilities for system security breaches and can even compromise the entire OS.

A new mechanism of access control needs to be enforced for such a component to run under unprivileged UID but still be able to access necessary OS

Belyaev, K. and Ray, I.
Towards Access Control for Isolated Applications.
DOI: 10.5220/0005970001710182
In *Proceedings of the 13th International Joint Conference on e-Business and Telecommunications (ICETE 2016) - Volume 4: SECRYPT*, pages 171-182
ISBN: 978-989-758-196-0

resources. *Squid Web Cache Server* component generates daily operational cache logs in its respective runtime environment. *Squid Log Analyzer* component needs to perform data analytics on those operational log files on a daily basis. It then creates analytical results in the form of HTML files that need to be accessible by the *HTTP Web Server* component to be available through the web browser for administrative personnel. In such a case, there is a need to access and share data objects across the applications in disjoint containerized environments. Note that, there are no shared data objects by which such communication can take place. Usual Inter-Process Communication (IPC) primitives such as message queues, memory-mapped files and shared memory may cause unauthorized access or illegal information flow. In fact, IPC could be disabled to minimize such security breaches. Moreover, legacy byte-level constructs, such as IPC, often do not fit under the development framework of modern applications that operate at the granularity of higher-level message objects. In such enterprise environments, a novel paradigm is needed for efficient and secure communication among isolated processes. Such coordination and sharing of data must take place in a controlled manner. Here again, an access control mechanism is needed that will allow this to happen.

We introduce a new framework referred to as *Linux Policy Machine* (and shorten it to just LPM in the rest of the paper) to address the identified challenges. Our LPM allows the management and enforcement of access to OS resources for individual applications and it also allows regulated inter-application communication.

We propose the notion of *capabilities policy classes*, each of which is associated with a set of capabilities. The capabilities policy classes differ from each other on the basis of capabilities they possess. Each application is placed in at most one capabilities policy class. The OS resources the application can access depends on the capabilities associated with that class. We have implemented and tested this concept on applications deployed under unprivileged (non-root) user entities in isolated (containerized) environments. This implementation forms a component of our LPM framework.

We also propose the concept of *communicative policy classes* for communication of applications that belong to different isolated environments. We adapt the generative communication paradigm introduced by Linda programming model (Gelernter, 1985) which uses the concept of tuple spaces for process communication. However, the traditional tuple spaces lack any security features and also have operational limitations. We enhance the original paradigm and also provide rules such that only applications belonging to the same communicative policy class can communicate using this approach. The communication between applications is mediated through a component of the LPM which is allowed limited access to an application's tuple space. Consider, for example, a described web caching data service that is partitioned into three components, each in a separate chrooted jail for security purposes. Our LPM allows a regulated way of coordinating and collaborating among components using tuple spaces. Note that, such coordination and collaboration will be allowed even if each of the components executes under different system UIDs and even group identifier (GID).

Our uniform framework provides a coherent business logic interface available to administrative personnel to manage access control for an application both at the level of OS resources and at the level of inter-application interaction. Such a centralized LPM construct that is resident in user-space acts as a reference monitor for a set of administered data services deployed on a single Linux server instance. We incorporate the access control modelling and decision control in user-space with robust and expressive persistence layer. This allows high interoperability and usage of the framework on any general-purpose Linux OS without a requirement for custom kernel patching (Krohn et al., 2007).

The rest of the paper is organized as follows. Section 2 gives a detailed overview of the model to manage capabilities using policy classes abstraction. Section 3 describes the proposed model for inter-application communication across isolated environments. Section 4 describes the design of the inter-application communication architecture. Section 5 demonstrates the feasibility of our approach by describing the initial prototype. Section 6 covers some related works. Section 7 concludes the paper.

2 CAPABILITIES POLICY CLASS

Our unified access control framework is designed for server-oriented applications deployed in isolated (containerized) environments dedicated to a single application or a small set of them in a security constrained manner. Most common examples include various system and user oriented services such as HTTP web and application servers, DNS server, NTP server, various relational database and key-value servers etc. The applications in these environments often need access to OS resources. In this section, we discuss how to confer such rights to the applications based on the principle of least privilege. We also dis-

cuss how applications having the access to the same resources are grouped into capabilities policy classes for ease of management and enforcement of the privileges. Such a model implemented as a component of a Linux Policy Machine makes the management and tracking of application-level capabilities feasible at the Linux endpoints.

In the Linux environments, the application runtime access control to the underlying OS resources has been traditionally regulated by root privileges which provides all permissions on system and user resources. The applications regulated by root privileges run with a special user identifier (UID = 0) that allows them to bypass access control checks. However, giving root permissions to an application violates the principle of least privilege and can be misused. Subsequently, in Linux kernels starting from version 2.1, the root privilege was partitioned into disjoint capabilities (Linux Developers, 2016). Instead of providing "root" or "non-root" access, capabilities provide more fine-grained access control – full root permissions no longer need to be granted to processes accessing some OS resources. For example, a web server daemon needs to listen to port 80. Instead of giving this daemon all root permissions, we can set a capability on the web server binary, like *CAP_NET_BIND_SERVICE* using which it can open up port 80 (and 443 for HTTPS) and listen to it, like intended. The new emerging concept of Linux application containers such as Docker service (Docker Developers, 2016) heavily leverages the Linux capabilities model. Despite the incorporation of capabilities in mainstream Linux and application containers, capabilities management in user-space is challenging and is yet to be addressed (Hallyn and Morgan, 2008).

Linux capabilities also may provide access to filesystem operations, such as, read/write on data objects. For example, *CAP_DAC_OVERRIDE* capability gives full read and write access to all files and devices on the system. The capability *CAP_DAC_READ_SEARCH* allows an application process to bypass the file read permission checks and directory read and execute permissions checks for any file or directory in the system. Currently, there are over 32 capabilities that can be applied to processes and executable files. Examples include *CAP_AUDIT_CONTROL*, *CAP_DAC_OVERRIDE*, *CAP_IPC_LOCK*, *CAP_NET_ADMIN* etc. Capabilities such as *CAP_DAC_OVERRIDE* bestow excessive access rights upon an application. In order to minimize the damage that could be caused by embedding possibly malicious code sections into applications, we provide only the smallest set of capabilities to an application that is needed to accomplish its functions. Such a task has to be assessed manually.

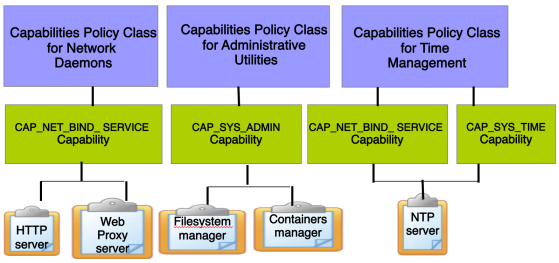

Figure 1: Capabilities Policy Classes.

Towards this end, we introduce the notion of a capabilities policy class that is associated with a set of Linux capabilities. Each capabilities policy class can have one or more applications. The applications in such a policy class have all the capabilities associated with this class and can therefore access the same set of OS resources as illustrated in Figure 1. Each application can belong to at most one capabilities policy class, but a class can have multiple applications. Note that, two distinct capabilities policy classes will be associated with different sets of capabilities.

For each new application we first identify the set of capabilities needed for the execution of its functions. We then check to identify if there is any capabilities policy class that has the exact same set of capabilities. If so, we assign this new application to that class. Otherwise, we create a new capabilities policy class. When the application is deleted, we remove the application from the capabilities policy class. When the functionality of the application changes and it needs a different set of capabilities than the class in which it belongs, the application is removed from the capabilities policy class and placed in a different one. When a policy class contains a single application and the capabilities needed by the application change, then we may add/remove capabilities from this policy class.

In order to support the above activities, the following high-level operations are supported by our capabilities policy class model: (i) create a capabilities policy class, (ii) add/remove capabilities to/from a policy class, (iii) show capabilities in a policy class, (iv) add/remove applications to/from a capabilities policy class, and (v) show/count applications in a capabilities policy class. Our current implementation of the LPM supports all of the above operations. The sample usage of the model is illustrated in Section 5.

3 COMMUNICATIVE POLICY CLASS

Often a group of applications may provide a service

and the individual applications that constitute this group are deployed in isolation as individual service components. Executing the individual service components in isolated containers has its benefits. If a single containerized application runtime is compromised by the attacker, the attack surface is limited in its scope to a single component. This theoretically limits the possibility of disrupting the entire data service. Moreover, such an approach also simplifies the management and provision of service components.

In this model, the individual isolated service components may need to coordinate and collaborate to provide the service and the various service components may not have access to a common centralized database management system or a key-value store for the purpose of communication. A simple example will help illustrate this case. The e-greetings service is composed of three service components each running in a separate isolated environment. The service sends personalized greeting e-cards to the e-mail addresses of the recipients. Components have only the minimum privileges needed to accomplish their tasks and are deployed under separate unprivileged UIDs. Due to isolation properties, those applications cannot write data objects to a shared storage area of the server OS such as /var directory to simplify their interaction. First component is the HTTP web server that provides the web interface for the service users to fill out the greetings card form where a user can choose a specific card image and specify the recipient e-mail address. Four key attributes of the individual greetings order are written into a separate order text file – greetings message, sender's and recipient's e-mail addresses and the absolute URL to the location of the postcard image file. Second component is the e-mail assembling agent (mailman) that processes the order file to prepare the greetings e-mail message. Third component of the service is the actual Mail Transfer Agent (MTA) that is responsible for sending the individual e-card e-mail message to the greetings recipient. Thus, the mailman component requires controlled access to the order file prepared by the web server component to create the e-mail message and the mail sending component, in turn, requires controlled access to individual mail message object.

In order to address the requirements of such applications, we introduce the notion of a *communicative policy class* that consists of a group of applications (service components) that reside in different isolated environments and need to collaborate and/or coordinate with each other in order to provide a service offering. Our notion of communicative policy class is different from the conventional notion of UNIX groups. In the conventional groups, the privileges as-

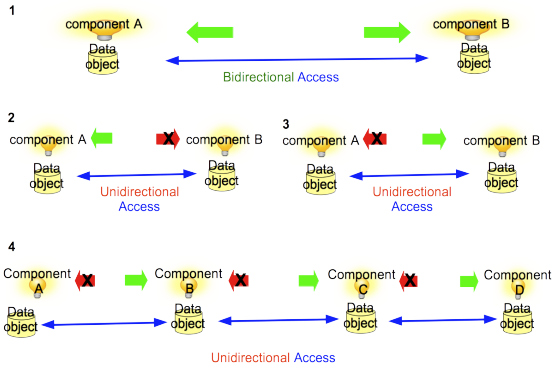

Figure 2: Flow control of isolated service components

signed to a group are applied uniformly to all members of that group. In this case, we allow controlled sharing of private data objects among members of the communicative policy class via object replication. Such a sharing can be very fine-grained and it may be *unidirectional* – an isolated application can request a replica of data object belonging to another isolated application but not the other way around. Some applications may require *bidirectional* access requests where both applications can request replicas of respective data objects. Such types of possible information flow are depicted in Figure 2 where green arrow denotes the granted request for a replicated data object in the direction of an arrow, while red one with a cross signifies the forbidden request. Implementing such rules may be non-trivial as isolated environments are non-traversable due to isolation properties. This necessitates proposing alternative communication constructs.

For instance, we can apply communicative policy class concept to applications deployed in standard chrooted jail that is a variant of isolated (containerized) runtime environment. Jail directory is assigned a distinct UID to utilize existing Linux Discretionary Acces Control (DAC) mechanism. A chroot on UNIX/Linux operating systems is an operation that changes the apparent root directory for the current running process and its children. A program that is run in such a modified environment cannot name (and therefore normally cannot access) files outside the designated directory tree. The usage of communicative policy classes for various sets of "jailed" applications will naturally allow them to interoperate in a seamless manner without a need to leverage or alter traditional DAC scheme of UNIX groups.

The access control policies of a communicative policy class specify how the individual applications in such a class can request a replica of each others' data objects. Note that, the individual applications in the class may have different Linux capabilities –

therefore, the applications may belong to the same communicative policy class but to different capabilities policy classes. Only applications within the same communicative class can communicate and therefore communication across different communicative policy classes is forbidden. Such a regulation is well-suited for multiple data services hosted on a single server instance. The assignment of individual data service to a separate policy class facilitates the fine-grained specification of communication policies between various isolated service components.

The construct of communicative policy class is designed to support the following communication patterns among the applications in a single class.

(i) *coordination* – often applications acting as a single data service do not require direct access to mutual data objects or their replicas but rather need an exchange of messages to perform coordinated invocation or maintain collective service component. For instance, one possible deployment scheme for the described e-greetings data service might add the layer of coordination logic to the interaction between the components. For example, if the web component of the service experiences excessive load from user requests and generates massive amounts of order files in a single directory, it can optionally send the coordination message to the mailman component to indicate that the request interval for the order files has to be decreased from the default 30 seconds to 5 seconds. Optionally, additional coordinative information may be incorporated in the message such as availability of alternative web server for load-balancing purposes. If the mailman component is capable of receiving and processing such coordination messages, it can, in turn, adjust its e-mail message preparation intervals. Consequently, it can coordinate with the mail sending component to indicate faster request times for the assembled e-mail message objects.

Coordination across mutually isolated environments is problematic. However, if applications belong to a single communicative policy class, it enables the exchange of coordination messages without reliance on usual UNIX IPC mechanisms that may be unavailable under security constrained conditions.

(ii) *collaboration* – components acting as a single data service may need to access mutual data or runtime file objects to collaborate and perform joint or codependent measurements or calculations as illustrated in the e-greetings example and the description of the web caching service presented in Section 1. Empowering an application with the ability to obtain a replica of a data object that belongs to another application in the same communicative policy class makes such collaboration possible.

The following high-level (business logic) operations are supported by our communicative policy class model as a component of our LPM: (i) create a communicative policy class (ii) add/remove applications to/from a communicative policy class (iii) show/count applications in a communicative policy class (iv) add/remove explicit permission for an application to request a replica of a filesystem data object(s) to/from a communicative policy class (v) enable/disable application coordination with other application(s) in a communicative policy class.

4 COMMUNICATION ARCHITECTURE

We now discuss the proposed enforcement architecture for communicative policy class model.

4.1 IPC Constraints

In general, applications that need to communicate across machine boundaries use TCP/IP level communication primitives such as sockets. However, that is unnecessary for individual applications located on a single server instance (Minsky et al., 2000). Applications that need to communicate on a modern UNIX-like OS may use UNIX domain sockets or similar constructs. However, socket level communication is usually complex and requires the development and integration of dedicated network server functionality into an application. Modern data service components also prefer information-oriented communication at the level of objects (Cabri et al., 2000). The necessity of adequate authentication primitives to prove application identity may also be non-trivial. Moreover, as illustrated in Section 3, many localized applications may require to communicate across isolated environments but may not need access to the network I/O mechanisms. Thus, more privileges must be conferred to these applications just for the purpose of collaboration or coordination, which violates our principle of least privilege.

Reliance on kernel-space UNIX IPC primitives may also be problematic. First, such an IPC may be unavailable for security reasons in order to avoid potential malicious inter-application exchange on a single server instance that hosts a large number of isolated application services. In other words, IPC may be disabled on the level of OS kernel (Johnson and Troan, 2004). Second, modern applications often require more advanced, higher-level message-oriented communication that is not offered by the legacy byte-level IPC constructs. Third, UNIX IPC is bound to

UID/GID access control associations that does not provide fine-grained control at the level of individual applications (Johnson and Troan, 2004). Therefore kernel-space IPC mechanisms do not offer regulated way of inter-application interaction.

The usage of system-wide user-space IPC frameworks such as D-Bus (Havoc Pennington, 2016) may also be problematic. D-Bus is the IPC and Remote Procedure Call (RPC) mechanism that primarily allows communication between GUI desktop applications (such as within KDE desktop environment) concurrently running on the same machine. D-Bus offers a relatively high-level message-oriented communication between applications on the same machine. However, it is not designed to transmit data objects such as logs. Although it is a widely accepted standard for desktop applications, D-Bus may not fit the requirements of modern server-based data services. In fact, the main design objective of D-Bus is not message passing but rather process lifecycle tracking and service discovery (Havoc Pennington, 2016). Moreover, applications have to connect to D-Bus daemon using UNIX domain sockets or TCP sockets. Before the flow of messages begins, two applications must also authenticate themselves which adds extra complexity layer to the communication. However, the more pressing problem is the possibility that user-space D-Bus daemon in line with kernel-space IPC may be disabled on the server node for security reasons. Moreover, system-wide communication resources such as global UNIX domain socket for the D-Bus daemon may be inaccessible for applications running in isolated environments.

4.2 Tuple Space Paradigm

In order to address the complexities introduced in 4.1, we propose an alternative approach that can be classified as a special case of *generative communication* paradigm introduced by Linda programming model (Gelernter, 1985). In this approach, processes communicate *indirectly* by placing tuples in a *tuple space*, from which other processes can read or remove them. Tuples do not have an address but rather are accessible by matching on content therefore being a type of content-addressable associative memory (Minsky et al., 2000). This programming model allows decoupled interaction between processes separated in time and space: communicating processes need not know each other's identity, nor have a dedicated connection established between them (Vitek et al., 2003).

The lack of any protection mechanism in the basic model (Vitek et al., 2003; Minsky et al., 2000) makes the single global shared tuple space unsuitable for interaction and coordination among untrusted components. There is also the danger of possible tuple collisions – as the number of tuples that belongs to a large set of divergent applications in a tuple space increases, there is an increasing chance of accidental matching of a tuple that was requested by another application. Moreover, the traditional in-memory implementation of tuple space makes it unsuitable in our current work due to a wide array of possible security attacks and memory utilization overheads. Therefore, we adapt the tuple space model that will satisfy our requirements for secure and reliable communication between applications within a single communicative policy class. Note that, in this adaptation the content-based nature of retrieval from a tuple space will necessitate content-based access control approaches (Minsky et al., 2000).

Another problem identified with the RAM-based tuple spaces is that it is suitable mainly for a single application with multiple threads that share the same memory address space. In such a simplified deployment scenario, a global tuple space is easily accessible by consumer and producer threads within a single application. However in the context of our current work we deal with separate data service applications that do not share the same address space in memory which makes such a solution unsuitable (Cabri et al., 2000). For instance, two isolated service components written in Java cannot access mutual tuple spaces because each component is deployed in a separate Java Virtual Machine (JVM) instance.

We propose a tuple space calculus that is compliant with the base model introduced in (Gelernter, 1985) but is applied on dedicated tuple spaces of individual applications instead of a global space. Our *tuple space calculus* comprises the following operations: (i) *create tuple space* operation, (ii) *delete tuple space* operation – deletes tuple space only if it is empty, (iii) *read* operation – returns the value of individual tuple without affecting the contents of a tuple space, (iv) *append* operation – adds a tuple without affecting existing tuples in a tuple space, and (v) *take* operation – returns a tuple while removing it from a tuple space. We adhere to the *immutability* property – tuples are immutable and applications can either append or remove tuples in a tuple space without changing contents of individual tuples.

Tuple space is proposed as a persistent filesystem abstraction with its calculus performed via tuple space library employed by the applications and the LPM. Therefore, this part of the proposed unified framework is not transparent and the applications may need to be modified in order to utilize the tuple space communication. However, in certain cases that may

not be necessary. For instance, if applications require only limited collaboration, such as periodic requests for replicas of data objects (the case for daily logs), a separate data requester application that employs a tuple space library can handle such a task without the need to modify the existing application such as a log analyzer.

An application is allowed to perform all the described operations in its tuple space while LPM is restricted to read and append operations only. Note that the take operation is the only manner in which tuples get deleted from a tuple space because the delete tuple space operation is allowed only on an empty tuple space.

The LPM plays a mediating role in the communication between applications. The communication takes place through two types of tuples: *control tuples* and *content tuples*. Control tuples can carry messages for coordination or requests for sharing. Content tuples are the mechanism by which data gets shared across applications (service components). The LPM periodically checks for control tuples in the tuple spaces for applications registered in its database. We have two different types of communication between a pair of applications. The first case is where the two applications do not share any data but must communicate with each other in order to coordinate activities or computation. The second case is where an application shares its data with another one.

The structure of the tuples is shown in Figure 3. Control tuples are placed by an application into its tuple space for the purpose of coordination or for requesting data from other applications. A control tuple has the following fields: (i) *Source ID* – indicates an absolute path of the application that acts as an application ID of the communication requester. (ii) *Destination ID* – indicates an absolute path of the application that acts as an application ID of the communication recipient. (iii) *Type* – indicates whether it is a collaborative or coordinative communication. (iv) *Message* – contains the collaborative/coordinative information. For collaboration it is the request for an absolute path of data object. Coordination message may be opaque as other entities may be oblivious of this inter-application communication. It may even be encrypted to ensure the security and privacy of inter-application coordination efforts. XML or JSON are possible formats that can be used for the representation of coordination messages. LPM merely shuttles the coordination tuples between respective applications' spaces and is not aware of their semantics. Content tuples are used for sharing data objects across applications and they have the following fields: (i) *Destination ID* – indicates the ID of recipient application

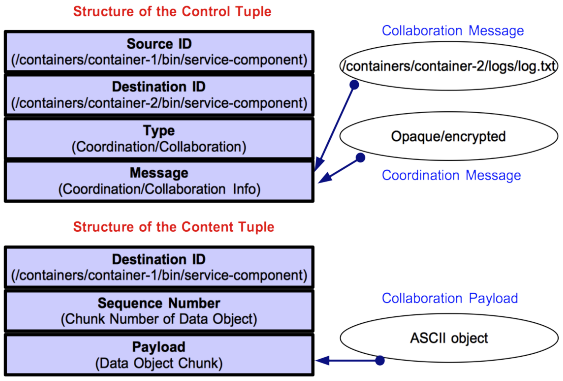

Figure 3: Tuples Structure.

that is an absolute path of an application. (ii) *Sequence Number* – indicates the sequence number of a data object chunk that is transported. ASCII objects in the form of chunks are the primary target of inter-application collaboration. (iii) *Payload* – contains the chunk of a data object. Content tuples are placed by the LPM reference monitor into corresponding tuple space of the requesting application that needs to receive content. Note that content tuples are designed for collaboration only. Coordination is performed exclusively through control tuples.

4.3 Security Aspects

We focus on the confidentiality, integrity, and availability issues with respect to tuple space implementation. Only members of the same communicative policy class can coordinate and/or share data. Extra protection mechanisms are also incorporated for each application's tuple space. The applications in the same isolated environment have a directory structure within the filesystem for their individual tuple spaces. Each application creates its own tuple space in this directory structure. Only the individual application can perform all the operations, namely, *create tuple space*, *delete tuple space*, *read*, *append*, and *take*. The LPM can only perform *read* and *append* operations on the tuple space. Thus, no one other than the application itself can remove anything from its tuple space. Moreover, the confidentiality and integrity are guaranteed by preventing other applications from directly accessing its tuple space. The directory structure in the filesystem allocated to a given application at any point of time is contingent upon the needs of the application, its priority, and its past behavior. Note that, from the confidentiality standpoint a malicious application cannot request a replica of a data object that belongs to another application deployed in the separate isolated runtime environment unless it is registered in the policy database containing communica-

tive policies classes of the LPM and has the appropriate privileges. Removing it from the associated communicative policy class will disable the collaboration with other applications. Unrestricted inter-application communication is avoided through the notion of *trust* between applications that is implicit for components of a single data service. Such components should be logically placed in the same communicative policy class as indicated in Section 3.

To further support isolation properties in containerized runtime environments such as application containers the runtime is expected to host only a single application (service component). For instance, in the isolated runtime environment such as a chroot jail the chroot directory is populated with all required program files, configuration files, device nodes and shared libraries that are required for the successful service component execution. To prevent the compromise of a tuple space, application's shared libraries (if present) have to be properly audited before inclusion in the runtime environment.

Applications may misbehave and cause Denial-of-Service (DOS) attacks by exhausting system resources. Fair scheduling mechanism (Belyaev and Ray, 2015) must be implemented to ensure that the LPM serves all the applications in a fair manner. Moreover, the tuple space library mechanism has to implement schemes that prevent an application from using all the allocated filesystem space in the directory structure of the isolated environment. We anticipate that the LPM must write only a single content tuple at a given time and the application has to take the tuple before a new one is written in its tuple space. Such a strategy avoids overconsumption of filesystem space, alleviates disk access loads, and also serves as an acknowledgement mechanism before the next chunk of the replicated data object is written.

5 IMPLEMENTATION AND DEPLOYMENT

We have implemented the prototype of a model for Linux capabilities management using policy classes. The model is a component of the LPM that supports policy classes abstraction where applications are assigned to various policy classes depending on the categorization criteria. LPM acts as a centralized enforcement point and reference monitor for the applications running on the Linux server instance. The unified framework uses the embedded SQLite database library to store and manage policy classes abstractions and their policies. The usage of embedded database facility eliminates the dependency on a separate database server that is prone to potential availability downtimes and security breaches. The LPM implemented in Java Standard Edition (SE) is deployed at Linux server endpoints. Figure 4 illustrates the components of the LPM. These are described below.

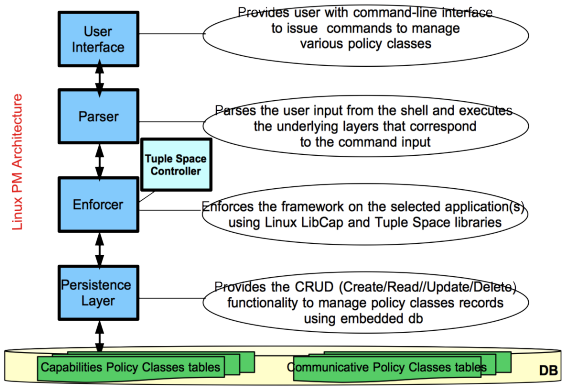

Figure 4: Linux Policy Machine (LPM) Architecture.

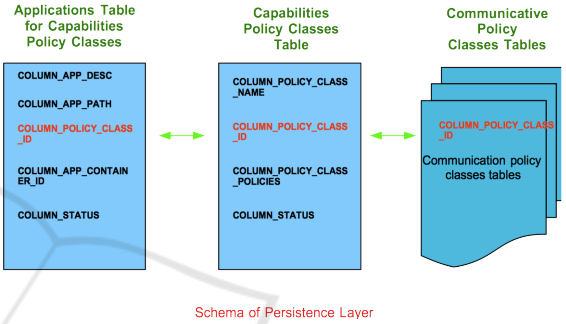

Figure 5: Database Schema of Persistence Layer.

[User Interface Layer:] This layer provides operator with command-line interface (CLI) to issue commands to manage the framework.

[Parser Layer:] This layer parses the user input from the CLI shell and forwards the parsed input to the underlying layers for execution.

[Enforcer Layer:] This layer enforces the capabilities on the given application using Linux LibCap (Linux Programmer's Manual, 2016) library and grants/denies access to OS resources depending on the capabilities policy class associated with the application. The layer also integrates a *tuple space controller* (Minsky et al., 2000) responsible for tuple space operations for the enforcement of collaboration and coordination of applications in a single communicative policy class.

[Persistence Layer:] This layer provides the Create/Read/Update/Delete (CRUD) functionality to manage records using embedded database facilities. The schema of the embedded database for storing framework policies appears in Figure 5.

The following commands illustrate the use of capabilities policy class model:

- *SHOW_CAPABILITIES*
- *SHOW_POLICY_CLASSES*
- *CREATE_POLICY_CLASS P*1 *network_applications_policy_class*
- *ADD_POLICY_CLASS_POLICY P*1 *CAP_NET_BIND_SERVICE*
- *ADD_POLICY_CLASS_POLICY P*1 *CAP_CHOWN*
- *MOVE_APP_TO_POLICY_CLASS /home/containers/web-container-100/bin/httpd P*1
- *REMOVE_POLICY_CLASS_POLICY P*1 *CAP_CHOWN*
- *SHOW_POLICY_CLASS_POLICIES P*1
- *SHOW_POLICY_CLASS_APPS P*1

The deployment of unified framework in the real-world settings requires a thorough performance evaluation. The model for capabilities policy classes does not incur any significant performance overheads for the unified framework. This is because its enforcement is based on the calls to the LibCap library (Linux Programmer's Manual, 2016) that essentially updates the filesystem capabilities metadata information for a process. Such operations do not incur the performance overheads because library mediations do not require extra disk I/O aside from the I/O load of the base system (Badger et al., 1996). There is also no additional memory utilization required aside from the RAM consumption by the LPM itself.

However the situation is quite different for the model of communicative policy classes. The enforcement is based on the anticipated tuple space paradigm that is known to be quite resource intensive (Minsky et al., 2000). The performance overheads for a memory-resident global shared tuple space are well known and include memory consumption overheads, efficiency problems with tuple matching at high speeds and search complexity with a large number of divergent tuples present in a single space continuum (Vitek et al., 2003). Those properties essentially pose a limit on a number of tuple objects in a given tuple space (Vitek et al., 2003; Minsky et al., 2000; Gelernter, 1985). Taking that into consideration as discussed in Section 4, the design of our tuple space implementation is reliant on the alternative strategy of the persistent filesystem-based solution with personal tuple spaces per application. Performance-wise, we rely on OS/hardware filesystem/disk caching (potentially leveraging fast flash-based storage) and the fact that tuple space communication in this work is oriented towards infrequent (semi-daily) replication of medium-sized data objects (not exceeding several Gigabytes) for a finite set of services deployed on a single server instance. Therefore we anticipate to conduct explicit performance benchmarks as part of the implementation evaluation of our tuple space enforcement sub-layer. It includes the benchmarking of various tuple space operations proposed in Section 4 with the measurements of disk/flash I/O, CPU and RAM utilization for the LPM with integrated tuple space controller. We also plan to measure performance degradation with various sizes of data objects replicated through the LPM reference monitor.

The initial prototype is publicly available through GitHub repository (Belyaev, 2016). We have deployed the LPM on the latest Fedora Server distribution with applications located in directories under distinct UIDs. We successfully tested the functionality of policy classes model of managing Linux capabilities as described in Section 2. Specifically, we have created several distinct policy classes for these application categories: (i) general-purpose applications that do not require any specialized capabilities, (ii) system-level sysadmin management utilities involving filesystem and related privileges, and (iii) applications that require access to the OS network stack. For instance HTTP Web and Cache servers have been deployed under unprivileged UIDs in chrooted jails with necessary network access capabilities delegated to their binaries by placing the applications in the network applications policy class using LPM's CLI management interface.

6 RELATED WORK

Traditionally, Linux environments supported DAC which allows read, write, and execute permissions for three categories of users, namely, owners, groups, and all others for managing access to files and directories in the user-space. Another type of supported access control is based on the Mandatory Access Control (MAC) designed to enforce system policies: system administrators specify policies which are checked via run-time hooks inserted into many places in the operating system's kernel. For managing access to system resources, typically superuser privileges are needed. Each file in the system is annotated with a numerical ownership UID. Applications needing access to system resources temporarily acquire the privilege of the superuser. The superuser is assigned UID = 0 – a process executing with this UID can bypass all access control rules. This simple model violates the principle of least privilege.

Researchers have proposed Domain and Type Enforcement (DTE) (Hallyn and Kearns, 2000; Badger et al., 1996) for Linux and UNIX environments. Type enforcement views a system as a collection of active entities (subjects) and a collection of passive entities (objects) (Badger et al., 1996). DTE is designed to

provide MAC to protect a system from subverted superuser processes as the access control is based on enforceable rule sets. The DTE model, unlike the other Linux approaches, avoids the concept of users and only concentrates on applications (Badger et al., 1996). Our work, like DTE, also concentrates on access control requirements of applications and their interaction. We also express policies in a human readable form. However, our LPM is entirely resident in user-space in contrast to DTE that offers kernel level solution. Moreover, we target the access control requirements necessary for the manageable deployment of large numbers of localized isolated application services under unprivileged UIDs in isolated environments, such as chrooted jails and application containers. Such environments were outside the scope of DTE.

Security-Enhanced Linux (SELinux) (Loscocco, 2001; Spencer et al., 1999) allows for the specification and enforcement of MAC policies at the kernel level. SELinux uses the Linux Security Modules (LSM) (Wright et al., 2002) hooks in the kernel to implement its policy. The SELinux architecture is based on the Generalized Framework for Access Control (GFAC) proposed by Abrams (Abrams et al., 1990) and LaPadula (LaPadula, 1995) and supports multiple security models. In SELinux the policy server makes access control decisions and the object managers are responsible for enforcing access control decisions. It provides a policy description language for expressing various types of policies. SELinux supports the concepts of roles and users but is not intended for enforcing policies at the level of individual applications. Policy description and configuration in SELinux is non-trivial because of the relationships between multiple models of SELinux and consequently it is a little challenging to use (Xu et al., 2014). Our work complements the efforts of SELinux in that it provides access control for isolated applications in user-space.

The Rule Set Based Access Control (RSBAC) (Ott and Fischer-Hübner, 2001) attempts to bring more advanced access control model to Linux based server systems. RSBAC is an open source security extension for current Linux kernels. The kernel based patch provides high level of security to the Linux kernel and operating environment. All RSBAC framework components are hard-linked into the custom-built Linux kernel. RSBAC supports divergent security policies implemented as modules in a single framework. However, the framework does not have a mature representation format to provide a unified way of modeling and expressing the policies for all the diverse policy modules that the framework claims to support. This limits its wide-spread adaptability. In contrast to RSBAC, our work provides domain-specific expressive policy formulation framework and is implemented in user-space that allows it to be deployed on any Linux server system.

The Grsecurity package (GrSecurity Developers, 2016) is a composition of Linux kernel patches combined with a small set of control programs. The package aims to harden known vulnerabilities in the Linux system while paying special attention to privilege escalation and root exploits. The set of patches provides protection mechanisms for file systems, executables and networks. It does this by placing additional logic on the Linux kernel and also alters the kernel's own mechanisms to comply with the desired behaviour. Grsecurity does not follow any formal model of security and access control, but emerged as a composition of countermeasures against several known weaknesses, vulnerabilities, or concrete attacks. Consequently, analysis of the security properties of the various mechanisms is non-trivial.

Application-defined decentralized access control (DCAC) for Linux has been recently proposed by Xu et al. (Xu et al., 2014) that allows ordinary users to perform administrative operations enabling isolation and privilege separation for applications. In DCAC applications control their privileges with a mechanism implemented and enforced by the operating system, but without centralized policy enforcement and administration. DCAC is configurable on a per-user basis only (Xu et al., 2014). The objective of DCAC is decentralization with facilitation of data sharing between users in a multi-user environment. Our work is designed for a different deployment domain – provision of access control framework for isolated applications where access control has to be managed and enforced by the centralized user-space reference monitor at the granularity of individual applications using expressive high-level policy language without a need to modify OS kernel.

In the realm of enterprise computing applications running on top of Microsoft Windows Server infrastructure the aim is to provide data services (DSs) to its users. Examples of such services are email, workflow management, and calendar management. NIST Policy Machine (PM) (Ferraiolo et al., 2014) was proposed so that a single access control framework can control and manage the individual capabilities of the different DSs. Each DS operates in its own environment which has its unique rules for specifying and analyzing access control. The PM tries to provide an enterprise operating environment for multi-user base in which policies can be specified and enforced in a uniform manner. The PM follows the attribute-based access control model and can express a wide

range of policies that arise in enterprise applications and also provides the mechanism for enforcing such policies. Our research efforts are similar to NIST PM (Ferraiolo et al., 2014) since it offers the policy management and mediation of data services through a centralized reference monitor. However, our access control goals are different. We do not attempt to model user-level policies as done by NIST PM. Our framework, on the other hand, provides the mechanism exclusively for controlled inter-application collaboration and coordination of localized service components across Linux-based isolated runtime environments that also regulates access to system resources based on the principle of least privilege. Note that, the importance of such a mechanism that is not currently present in NIST PM is acknowledged by its researchers (Ferraiolo et al., 2014).

In the mobile devices environment Android Intents (Chin et al., 2011) offers message passing infrastructure for sandboxed applications; this is similar in objectives to our tuple space communication paradigm proposed for the enforcement of regulated inter-application communication for isolated service components using our model of communicative policy classes. Under the Android security model, each application runs in its own process with a low-privilege user ID (UID), and applications can only access their own files by default. That is similar to our deployment scheme. Our notion of capabilites policy classes is similar to Android permissions that are also based on the principle of least privilege. Permissions are labels, attached to application to declare which sensitive resources it wants to access. However, Android permissions are granted at the user's discretion (Armando et al., 2015). Our server-oriented centralized framework deterministically enforces capabilities and information flow accesses between isolated service components without consent of such components based on the concept of policy classes. Despite their default isolation, Android applications can optionally communicate via message passing. However, communication can become an attack vector since the Intent messages can be vulnerable to passive eavesdropping or active denial of service attacks (Chin et al., 2011). We eliminate such a possibility in our proposed communication architecture due to the virtue of tuple space communication that offers connectionless inter-application communication as discussed in Section 4. Malicious applications cannot infer on or intercept the inter-application traffic of other services deployed on the same server instance because communication is performed via isolated tuple spaces on a filesystem. Moreover, message spoofing is also precluded by our architecture since the enforcement of message passing is conducted via the centralized LPM reference monitor that regulates the delivery of messages according to its policies store.

Our work also bears resemblance to the Law-Governed Interactions (LGI) proposed by Minsky et al. (Minsky et al., 2000; Minsky and Ungureanu, 1998) which allows an open group of distributed active entities to interact with each other under a specified policy called the law of the group. The inter-application communication in our work is proposed in the same manner via the tuple space using the tuple space controller integrated in our centralized LPM reference monitor that has complete control over inter-application interaction (Minsky and Ungureanu, 1998; Cremonini et al., 2000).

7 CONCLUSION AND FUTURE WORK

We have demonstrated how a Linux Policy Machine (LPM) can be developed for the Linux environment that provides access control specification and enforcement for applications running in isolated environments. The beauty of our system lies in providing a uniform business logic interface to manage access control at both the kernel and application levels. We proposed the notion of policy classes to manage policies pertaining to accessing system and application level resources and demonstrated how inter-application communication can take place through tuple spaces. The initial prototype demonstrates the feasibility of our approach.

A lot of work remains to be done. We are in the process of developing the tuple space library and the tuple space controller necessary for the enforcement of communicative policy class model. Also the Parser and Persistence Layers have to be extended to support formulation of policies for communicative model. We also plan to extend this work for distributed settings (Belyaev and Ray, 2015; Singh et al., 2014) where service policies are managed, formulated and updated in a centralized location, and then distributed and enforced at remote LPM nodes.

ACKNOWLEDGEMENTS

This work was supported in part by a grant from NIST under award no. 70NANB15H264 and by grants from NSF under award no. CNS 1619641 and IIP 1540041.

REFERENCES

Abrams, M., Eggers, K., LaPadula, L., and Olson, I. (1990). Generalized Framework for Access Control: An Informal Description. In *Proceedings of NCSC*.

Armando, A., Carbone, R., Costa, G., and Merlo, A. (2015). Android Permissions Unleashed. In *Proceedings of IEEE CSF*, pages 320–333. IEEE.

Badger, L., Sterne, D. F., Sherman, D. L., Walker, K. M., and Haghighat, S. A. (1996). A Domain and Type Enforcement UNIX Prototype. *Computing Systems*, 9(1):47–83.

Belyaev, K. (2016). Linux Policy Machine (LPM) – Managing the Application-Level OS Resource Control in the Linux Environment. https://github.com/kirillbelyaev/tinypm/tree/LPM. accessed 12-March-2016.

Belyaev, K. and Ray, I. (2015). Towards Efficient Dissemination and Filtering of XML Data Streams. In *Proceedings of IEEE DASC*.

Cabri, G., Leonardi, L., and Zambonelli, F. (2000). XML Dataspaces for Mobile Agent Coordination. In *Proceedings of ACM SAC*, pages 181–188. ACM.

Chin, E., Felt, A. P., Greenwood, K., and Wagner, D. (2011). Analyzing inter-application communication in Android. In *Proceedings of ACM MobiSys*, pages 239–252. ACM.

Cremonini, M., Omicini, A., and Zambonelli, F. (2000). Coordination and access control in open distributed agent systems: The TuCSoN approach. In *Coordination Languages and Models*, pages 99–114. Springer.

Docker Developers (2016). What is Docker? https://www.docker.com/what-docker/. accessed 12-March-2016.

Ferraiolo, D., Gavrila, S., and Jansen, W. (2014). On the Unification of Access Control and Data Services. In *Proceedings of IEEE IRI*, pages 450–457. IEEE.

Gelernter, D. (1985). Generative Communication in Linda. *ACM TOPLAS*, 7(1):80–112.

GrSecurity Developers (2016). What is GrSecurity? https://grsecurity.net. accessed 12-March-2016.

Hallyn, S. and Kearns, P. (2000). Domain and Type Enforcement for Linux. In *Proceedings of ALS*, pages 247–260.

Hallyn, S. E. and Morgan, A. G. (2008). Linux Capabilities: Making them Work. In *Proceedings of OLS*, page 163.

Havoc Pennington, Red Hat, I. (2016). D-Bus Specification. https://dbus.freedesktop.org/doc/dbus-specification.html. accessed 12-March-2016.

Johnson, M. K. and Troan, E. W. (2004). *Linux Application Development*. Addison-Wesley Professional.

Krohn, M., Yip, A., Brodsky, M., Cliffer, N., Kaashoek, M. F., Kohler, E., and Morris, R. (2007). Information flow control for standard OS abstractions. *ACM SIGOPS OSR*, 41(6):321–334.

LaPadula, L. (1995). Rule-Set Modeling of Trusted Computer System. In M., A., S., J., and H., P., editors, *Information Security: An Integrated Collection of Essays*. IEEE Computer Society Press.

Linux Developers (2016). Linux Programmer's Manual. http://man7.org/linux/man-pages/man7/capabilities.7.html. accessed 12-March-2016.

Linux Programmer's Manual (2016). LIBCAP Manual. http://man7.org/linux/man-pages/man3/libcap.3.html. accessed 12-March-2016.

Loscocco, P. (2001). Integrating Flexible Support for Security Policies into the Linux Operating System. In *Proceedings of USENIX ATC, FREENIX Track*, page 29.

Manual, L. P. (2016). Kernel Namespaces. http://man7.org/linux/man-pages/man7/namespaces.7.html. accessed 12-March-2016.

Minsky, N. H., Minsky, Y. M., and Ungureanu, V. (2000). Making Tuple Spaces Safe for Heterogeneous Distributed Systems. In *Proceedings of ACM SAC*, pages 218–226.

Minsky, N. H. and Ungureanu, V. (1998). Unified support for heterogeneous security policies in distributed systems. In *Proc. of USENIX SS*, pages 131–142.

n–Logic Ltd. (2016). n-Logic Web Caching Service Provider. http://n-logic.weebly.com/. accessed 12-March-2016.

Ott, A. and Fischer-Hübner, S. (2001). The Rule Set Based Access Control (RSBAC) Framework for Linux. In *Proceedings of ILK*.

Singh, J., Bacon, J., and Eyers, D. (2014). Policy enforcement within emerging distributed, event-based systems. In *Proceedings of ACM DEBS*, pages 246–255. ACM.

Spencer, R., Smalley, S., Loscocco, P., Hibler, M., Andersen, D., and Lepreau, J. (1999). The Flask Security Architecture: System Support for Diverse Security Policies. In *Proceedings of USENIX SS*.

Vitek, J., Bryce, C., and Oriol, M. (2003). Coordinating Processes with Secure Spaces. *Science of Computer Programming*, 46(1):163–193.

Wright, C., Cowan, C., Smalley, S., Morris, J., and Kroah-Hartman, G. (2002). Linux Security Modules: General security support for the Linux kernel. In *Proceedings of USENIX SS*, pages 17–31.

Xu, Y., Dunn, A. M., Hofmann, O. S., Lee, M. Z., Mehdi, S. A., and Witchel, E. (2014). Application-Defined Decentralized Access Control. In *Proceedings of USENIX ATC*, pages 395–408.