Aeolus Reference Manual

Barbara Liskov

Contents

1 Introduction

This document describes the interface that the Aeolus platform provides for users who are implementing applications using Java. The document explains how the Aeolus features are made available by means of a Java library.

The document assumes readers are already familiar with Aeolus (see [2]); however we provide a brief overview here. Aeolus provides a DIFC (distributed information flow control) model in which tags are used to compartmentalize data. Each object (file, thread) has two labels, a secrecy label and an integrity label, where a label is a set of tags. Information is allowed to flow from a source to a target only if the target's labels are *at least as constrained as* those of the source (i.e., the target's secrecy label contains that of the source, and its integrity label is contained in that of the source). Labels of data objects are immutable but the labels of threads can change. Some changes are safe (adding a tag to a secrecy label, removing a tag from an integrity label) but others are not. The unsafe label manipulations require authority, which resides with principals (PIDs). Each thread runs on behalf of a principal, and a thread can perform the unsafe label manipulations only if its principal has the proper authority: a tag can be removed from a secrecy label (declassification) or added to an integrity label (endorsement) only if its principal is authoritative for that tag.

Figure 1 shows an Aeolus deployment, consisting of a set of nodes that run application code and a special node, called the AS, that stores the authority state (the principals, tags, and information about what principals are authoritative for what tags). Each node within the deployment runs application code as one or more VNs (virtual nodes); in addition some nodes may provide an Aeolus file system. Within a VN are many threads, and these threads can communicate via our shared state mechanism.

Nodes within a deployment use DNS names to identify one another. Some nodes may run an Aeolus file system as well as VNs; in this case the file system is also identified by the DNS name of the node.

VNs can communicate using RPCs, and labels flow on these internal communications. Communication with programs running outside of a deployment can be accomplished through the use of Java sockets. In addition threads can read and write files from external file systems and make use of input/output devices using Java I/O. Labels do not flow on communications that cross the deployment boundary, however, since we cannot vouch for what happens to information outside of a deployment. Instead, we require

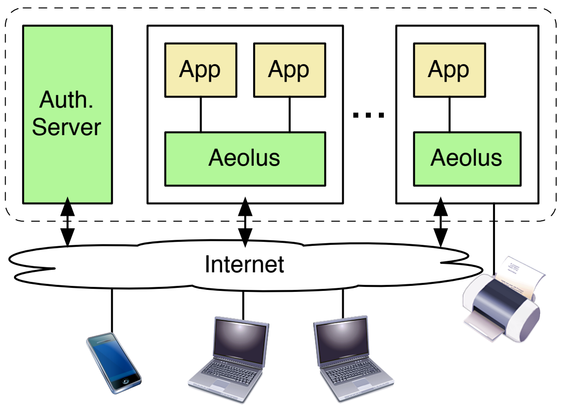

Figure 1: An Aeolus Deployment.

that writes to outside a deployment can be done only if the writer has a null secrecy label, and reads from outside a deployment can be done only if the reader has a null integrity label.

The Aeolus library is defined in edu.mit.csail.aeolus.API. The library provides the various classes that support Aeolus types, e.g., PIDs, tags, and labels. In addition there is a special class, AeolusLib, that provides access to thread state and to some other features of our system. For example, VNs are created through a call to an AeolusLib method. AeolusLib is a class with no constructor and only static methods.

In this document we do not use prefixes for Aeolus type names nor for Aeolus exceptions, e.g., we talk about PID, rather than edu.mit.csail.aeolus.API.PID. We also refer to certain Java types, such as List, without giving the name of the library.

We begin in Section 2 with a discussion of the constraints that Aeolus imposes on code of Java applications that run on the platform. Section 3 provides a brief discussion of the exceptions thrown by our platform. Sections 4 - 6 describe the basic concepts in the Aeolus model (principals, tags, and labels) and Section 7 describes the AeolusLib methods used to access a thread's state. Subsequent sections describe other types provided by Aeolus as well as other methods of AeolusLib.

Further documentation about the Aeolus interface and how to use Aeolus can be found at www.pmg.csail.mit.edu/aeolus/.

2 Code Restrictions

Code that runs on Aeolus needs to satisfy certain constraints to ensure that information leaks do not occur.

As discussed above (and in [2]), an application in Aeolus runs as one or more VNs. All VNs at the same node run in the same process; that process runs the Aeolus platform software on top of the JVM. Within a VN there can be many concurrent threads; threads are created to handle incoming requests, and additionally threads can be created explicitly by forks. Threads running within a VN are allowed to share volatile state through our shared state mechanism.

Our implementation approach provides good performance since we use lightweight threads to carry out concurrent tasks rather than separate OS-level processes. However because all threads within a VN run in the same process, we have to ensure that no information flow errors occur as a result.

We provide this guarantee by limiting what application code can do. Our restrictions are enforced by a combination of load-time checks, code rewriting, and runtime checks. These techniques are applied in a way that does not affect either the JDK nor the Aeolus classes, but they do apply to all code loaded for an application.

We enforce the following restrictions when application code is loaded:

- No static variables. As mentioned all code within a VN runs in the same process and therefore shares the same copy of the code. Static variables would allow threads within at VN to share state without the checking needed to ensure no information flow control violations.

- No use of native code except for approved libraries. At present only the standard libraries are approved; additional libraries can be approved if users request them and they are safe for use within Aeolus. We do not plan to approve native code implemented as part of the application itself.

- No application-defined class loaders (since otherwise we couldn't rely on our class loader to enforce the constraints).

- All subclasses of AeolusShared and AeolusSafe must satisfy the restrictions defined in Sections 10.5 and 11.

This checking is done either when a VN is launched (see Section 8) or later, when code is loaded dynamically. If the checks fail at the time a VN is being launched, the VN will not be created and the launch call will fail with ClassNotFoundException. If the checks fail during dynamic loading, the ClassNotFoundException is raised.

An additional restriction is:

- No use of synchronization except within certain immutable Java types such as String, and within application classes that implement subtypes of AeolusShared (see Section 10).

We prohibit synchronization because it can be used to violate information flow control. Aeolus threads do share objects, e.g., those in shared state. We ensure that calls to methods of these objects are safe because we check labels for each use. However, if threads could synchronize on these objects, this would create a covert channel.

We do not exclude code containing prohibited uses of synchronization. Instead we rewrite the Java byte codes to remove such uses. In particular, consider the byte codes generated for

```
synchronize (e) { stmt }
```

We rewrite this code to leave only the bytecodes for statement *stmt*. We also rewrite code to remove uses of wait and notify.

Removing synchronization does not cause execution errors because we exclude it only for code of objects that are "safe-to-share" as discussed in Section 11, and for code of objects that cannot be shared; no synchronization is needed for such objects.

Third, we enforce the following restrictions dynamically:

- No introspection since this allows breaking of abstraction barriers and as a result can allow tainted information to become visible without proper tracking.

- When a thread does output, its secrecy label must be null.

- When a thread does input, its integrity label must be null.

- No use of Java threads. All threads must be created through Aeolus, so that we can provide each thread with its own information flow control state and private heap.

We do this checking by using a custom Java SecurityManager, which allows us to interpose our own code on every use of I/O, and on every fork. If a constraint is violated, the attempt fails with a SecurityException.

In addition the Aeolus class loader rewrites the Java byte code to support shared objects and closures, as discussed in Sections 10 and 12.

3 Exceptions

Many calls within Aeolus terminate with exceptions related to the particular task being done. For example, if an RPC cannot be performed because there is a problem communicating with the target of the call, the call terminates with RpcException.

There are two exceptions that are thrown by many calls. InfoFlowControlException is thrown when a call fails because the caller's labels don't satisfy an information flow control constraint, e.g., the caller's secrecy label is larger than that of the file on a file write. AuthorityException is thrown when a thread lacks the authority needed to perform the call, e.g., cannot do a declassification because the caller lacks authority.

All exceptions thrown explicitly by the Aeolus platform are subtypes of AeolusException. AeolusException is a general failure exception that is raised if an unexpected error occurs, e.g., inability to communicate with the AS. To simplify the presentation, most specifications do not mention the possibility of the call terminating with AeolusException; instead this should be assumed as a possibility for most calls.

Aeolus exceptions have a string argument that explains in more detail what happened.

4 Principals

Principals (PIDs) are the online entities that represent users with an interest in security. One principal can *act for* another, meaning it can do all security sensitive operations that the one it acts for can do.

Aeolus maintains a *principal hierarchy (PH)* to record the act-for relations. This is a directed acyclic graph where the nodes are PIDs. A link from principal p1 to principal p2 means that p1's authority has been delegated to p2, and therefore p2 can act for p1. The PH is part of the authority state, which records all information about allowed uses of authority for an Aeolus deployment.

PID provides the following methods and constructors. Calls to PID methods and constructors that cause modifications to the authority state can be called *only* if the secrecy label of the thread making the call is null; this restriction is needed to avoid the possibility of information flow leaks through the AS.

> PID( ). If the thread's secrecy label isn't empty, throws InfoFlowControlException. If the thread's PID is the public PID (which isn't allowed to act for any other PID), throws AuthorityException. Otherwise creates a new principal and returns its PID; the caller's principal acts-for the new principal (i.e., there is a delegation link from the new principal to the caller's principal).
>
> void actFor(PID p). If the thread's secrecy label isn't null throws InfoFlowControlException. If the thread's PID doesn't act for this or if p is the public PID, throws AuthorityException. If the link already exists, does nothing. Otherwise adds a delegation link from this to p, allowing p to act for this, except that if adding the link would cause a cycle in the PH, the call fails with AeolusException.
>
> boolean doesActFor(PID p). Returns true if this acts for p, else returns false. This is a transitive test: this acts for p if there is a path of delegation links from p to this.
>
> void revoke(PID p). If the thread's secrecy label isn't empty, throws InfoFlowControlException. If the thread's PID doesn't act for this, throws AuthorityException. If there is no delegation link from this to p does nothing else removes this link.
>
> static PID getPublicPID( ). This is a static method that returns the public PID.
>
> boolean equals(PID p). Returns true if p is the same PID as this else returns false.

PID is serializable and cloneable.

5 Tags

Tags are used to compartmentalize information. Each tag has an associated *delegation graph* that records which principals have authority for that tag. A delegation graph is directed and acyclic. Each node is a principal; a link from principal $p1$ to principal $p2$ indicates that p1's authority for the tag has been delegated to $p2$. When a tag is created its graph consists of a single node containing the principal of the creating process. A call to the delegation method, if successful, adds a link to the graph; the call will fail if adding the link would cause a cycle to form in the graph. A call to revoke indicates a specific link to remove from the graph.

The delegation graph together with the principal hierarchy determines whether a thread has authority for a tag: the thread *is authoritative for the tag* if its principal acts-for some principal that is a node in the delegation graph of that tag.

There are two kinds of tags: supertags and subtags. If a principal is authoritative for a supertag, it is also authoritative for all subtags of that supertag. The relationship between a supertag and a subtag is determined when the subtag is first created. We provide only a two-level hierarchy: subtags are not allowed to have subtags.

The AeolusTag type provides methods to create and compare tags and to grant and revoke authority for tags. Calls to AeolusTag constructors and to methods that cause changes to the authority state are allowed *only* if the caller's secrecy label is null (to avoid information flow leaks through the AS).

AeolusTag( ). If the thread's secrecy labels isn't empty, throws InfoFlowControlException; if the thread's principal is the public PID (which isn't allowed to be authoritative for anything), throws AuthorityException. Otherwise returns a new top-level tag; the caller's principal is authoritative for the new tag.

AeolusTag(tag t). If the thread's secrecy label isn't empty, throws InfoFlowControlException; if the thread's principal is the public PID, throws AuthorityException; if t isn't a top-level tag, throws AeolusException. Otherwise returns a new tag that is a subtag of t; the caller's principal is authoritative for the new tag.

boolean isSubTag(AeolusTag t). Returns true if t is a subtag of this, else returns false.

boolean isSuperTag(AeolusTag t). Returns true if t is a supertag of this, else returns false.

boolean equals(AeolusTag t). Returns true if t is the same tag as this.

void delegate(PID p1, PID p2). If the caller's secrecy label isn't empty, throws InfoFlowControlException. If the caller doesn't act for p1 or p2 is the public PID, throws AuthorityException. If p1 isn't a node in the delegation graph for this, or if adding the link would create a cycle in the graph, or some other error occurs, throws AeolusException. Otherwise adds a delegation link from p1 to p2 to the graph for this, except that if the link already exists, does nothing.

boolean hasAuthority( ). Returns true if the caller is authoritative for this.

boolean hasAuthority(p). Returns true if p is authoritative for this.

void revoke(PID p1, PID p2). If the caller's secrecy label isn't null, throws InfoFlowControlException. If the caller doesn't act for p1, throws AuthorityException. If there is no link from p1 to p2 in the delegation graph for this does nothing else removes this link.

AeolusTag is serializable and cloneable.

6 Labels

Labels are sets of tags. They are used to record the confidentiality and integrity of threads, shared objects, and files. Labels are provided via the AeolusLabel type, with the following constructors and methods:

AeolusLabel( ). Returns an empty label.

void addTag(AeolusTag t). Adds t to this.

void removeTag(AeolusTag t). Removes t from this.

boolean isSubsetOf(AeolusLabel l). Returns true if this is a subset of l, else returns false.

boolean equals(AeolusLabel l). Returns true if this contains the same tags as l, else returns false.

boolean isEmpty( ). Returns true if this is empty else returns false.

boolean hasTag(AeolusTag t). Returns true if t is a member of this; else returns false.

AeolusLabel union(AeolusLabel l). Returns a new label that contains the tags in this and the tags in l.

AeolusLabel intersection(AeolusLabel l). Returns a new label that contains the tags that are in both this and l.

List<AeolusTag> members( ). Returns the list of tags contained in this.

AeolusLabel clone( ). Returns a new label object that contains the same tags as this.

To understand label operations it is important to understand how labels interact with compound tags:

- A label that contains supertag t is considered to contain all subtags of t.

All the methods described above should be understood in light of this definition, e.g., hasTag(t) will return true if the label contains t or the supertag of t, and if t is a supertag and the label contains one or more subtags of t, then remove(t) removes these subtags.

Furthermore we use a "reduced" representation for labels: if a label contains a supertag, t, it does not physically contain any of t's subtags. Thus if the members method is called on the label, t will be one of the tags in the List, but no subtags of t will be in the list.

Labels are serializable and cloneable.

7 Thread State

Each thread has a security state consisting of its current principal and its secrecy and integrity labels. AeolusLib methods allow it to observe its principal and to obtain copies of its labels. In addition a thread can modify its state, but *only* by calling AeolusLib methods.

AeolusLib provides access to thread state through the following methods:

static PID getPID( ). Returns the current principal of the thread.

static AeolusLabel getSecrecy( ). Returns a copy of the thread's secrecy label.

static AeolusLabel getIntegrity( ). Returns a copy of the thread's integrity label.

static void addSecrecy(AeolusTag t). Adds t to the thread's secrecy label.

static void declassify(AeolusTag t). Removes t from the thread's secrecy label provided the thread is authoritative for t; otherwise throws AuthorityException.

static void removeIntegrity(AeolusTag t). Removes t from the thread's integrity label.

static void endorse(AeolusTag t). Adds t to the thread's integrity label provided the thread is authoritative for t; otherwise throws AuthorityException.

8 Virtual Nodes

All computation in Aeolus occurs within VNs (virtual nodes). VNs are created and destroyed by means of AeolusLib methods; in addition AeolusLib methods are used to register a VN to provide a service (so that it can be used as the target of an RPC), and to deregister a VN for a service.

- static void launch(String hostname, PID p, String appName, String appArg). Launches a VN on the indicated host (the host is identified by its DNS name), with the authority of p, provided the launch succeeds. The launch fails under the following conditions: if the caller doesn't act for p, throws AuthorityException; if the caller's secrecy label isn't empty, throws InfoFlowControlException; if the application class is not found on hostname, throws ClassNotFoundException; if the main method isn't found in the application, throws NoSuchMethodException; if the code doesn't satisfy the code restrictions (see Section 2), throws ClassNotFoundException; if an exception is thrown by the main method, throws InvocationTargetException; if there is a failure in communication with hostname, throws RpcException; if the node isn't a member of the deployment or there is any other failure, throws AeolusException. The thread that runs the main method runs with principal p, and starts running with null labels; when it terminates its labels are merged into those of the caller, whether it completes successfully or throws an exception.

- static void registerService(String serviceName, Class<?> serviceClass). Registers this VN as providing the specified service, with the given interface and name. This allows the service to be called via an RPC; such a call will run with the VN's authority. If there is already a registration for this service at the VN's node, it is superceded by the new registration. To register a service the caller's secrecy label must be null; otherwise throws InfoFlowControlException. If serviceClass or serviceName is null, throws NullPointerException.

- static void shutdown( ). If the thread's secrecy label is non-empty, throws InfoFlowControlException. Otherwise shuts down the VN of the caller's thread and deregisters any services registered for it.

A launch starts up a VN at the designated hostname (the host is identified by its DNS name); this VN will run with the designated principal, p. Launch works as follows. Recall that all VNs at a node run in the same process. Aeolus starts up another VN in this process at hostname and creates a thread running with p's authority within this VN. The thread runs the main method of the application code, with appArg as its argument and empty labels. The caller is delayed until the main method finishes running, and the labels of the main method at this point are sent back to the launcher, where they are merged with the launcher's labels. (A merge is a union for the secrecy labels and an intersection for the integrity labels.)

9 RPCs

RPCs can be made to methods of registered services. To make an RPC, the caller must obtain a proxy object for the service by calling the following AeolusLib method:

static Object getService(String hostName, String serviceName, Class<?> serviceClass). Returns a service stub that implements the interface specified by serviceClass, for use in making calls to the designated service on the designated machine (or this machine if the hostName is null).

The thread can then use the service stub to do RPCs. Typically the caller will first cast the Object to the specified interface to make these calls convenient.

For an RPC using a service stub to succeed, hostName must be the DNS name of a member of the deployment (if hostName is null the call goes to this machine so the check will succeed), some VN at that node must have registered for that service, with the indicated interface, and communication with that node must succeed; otherwise the call terminates with RpcException.

If these conditions are all met, the registration service at hostName forwards the call to the VN that is registered for the indicated service. At the VN, Aeolus creates a new service object, using the default constructor; since this constructor takes no arguments, this means that there is no way the service object can access pre-existing objects in the heap of its VN, except through the use of shared state (discussed in Section 10).

Next Aeolus creates a new thread within the VN and calls the specified method within that thread. This method is provided with copies of the arguments (the arguments are serialized at the caller and deserialized at the callee). The thread runs with the labels of the caller, but the PID of the VN.

If execution of the call terminates normally, the result of the call is sent back to the caller (it is serialized at the callee and deserialized at the caller) and the RPC terminates normally; if the execution of the call terminates with an exception, the RPC terminates with java.lang.Exception. In either case, the callee's labels are merged with those of the caller and the caller continues running with its own PID.

Aeolus provides special semantics for sending AeolusBox objects in RPC messages as described in Section 10.2.

10 Shared State

Threads within a VN can share volatile memory through the use of shared state. Shared state consists of one or more shared objects. Each such object has secrecy and integrity labels; these labels are provided when the object is created and cannot be changed afterwards. The methods of a shared object can be called only if the thread and object labels allow the call.

Shared state objects can be thought of as residing in a special "shared heap". Objects in thread heaps can point to shared objects, and also objects in the shared heap can point to other shared objects. However shared objects cannot point to the heaps of the threads. In addition shared objects are "well encapsulated": it isn't possible to have pointers from outside a shared object to a non-shared object that is inside that shared object. These constraints are enforced by copying arguments and results of calls to methods of shared objects. These are deep copies, but they only go down to shared state objects, since shared objects and thread heaps are allowed to refer to shared objects. Furthermore, copying is avoided for "safe" objects as defined in Section 11.

Aeolus provides three types of shared objects (boxes, shared queues, and shared locks). In addition, users can define new types of shared objects.

The shared heap has a root, which can be used by threads to find objects within shared state. The root is undefined when the VN first starts running. It can be accessed using the following AeolusLib methods:

- static void setRoot(AeolusShared x). Makes x the root of the shared state provided the secrecy label of the thread making the call is null; otherwise throws InfoFlowControlException.
- static AeolusShared getRoot( ). If there is no root (because setRoot has not been called previously at this VN), throws AeolusException. Otherwise returns a pointer to the shared root object.

10.1 AeolusShared

Aeolus provides an abstract class AeolusShared that serves as the root of the hierarchy of shared types. All shared objects are defined by classes that implement subtypes of AeolusShared:

abstract class AeolusShared

AeolusShared provides two methods:

AeolusLabel getSecrecyLabel( ). Returns a copy of the secrecy label of the shared object.

AeolusLabel getIntegrityLabel( ). Returns a copy of the integrity label of the shared object.

Objects belonging to subclasses of AeolusShared can be created only by calling constructors of the subclass. AeolusShared provides two constructors for use within these subclasses:

> protected AeolusShared( ). Creates a new AeolusShared object with copies of the labels of the caller.
>
> protected AeolusShared(AeolusLabel s, AeolusLabel i). Creates a new AeolusShared object with copies of the indicated labels provided the caller's labels are no more constrained than the indicated labels. Otherwise throws InfoFlowControlException.

10.2 Boxes

Boxes are generic containers for arbitrary content:

- final class AeolusBox<T extends Serializable> extends AeolusShared
  implements Serializable

with the following constructors and methods:

> AeolusBox<T>(T content). Creates a new AeolusBox<T> object with copies of the caller's labels, and stores a copy of content in the box.
>
> AeolusBox<T>(AeolusLabel s, AeolusLabel i, T content). Creates a new AeolusBox<T> object with copies of the provided secrecy and integrity labels and containing a copy of content, provided the caller's labels are no more constrained than those of the box; otherwise throws InfoFlowControlException.
>
> T get( ). Returns a copy of the content of the box provided the caller's labels allow the read; otherwise throws InfoFlowControlException.
>
> void put(T x). Copies x into the box provided the callers label's allow the write; otherwise throws InfoFlowControlException.

In addition, AeolusBox inherits the getSecrecyLabel() and getIntegrityLabel() methods from AeolusShared.

When content is moved into a box or from a box, it is copied. This copy goes down to any shared objects that are reachable from the object being copied.

Boxes can be used as arguments and results of RPCs, but this is done in a special way that hides the content: both the box content and the labels are sent in the message, and the box is reconstructed automatically at the recipient. This allows contaminated content to be passed through intermediaries without their becoming contaminated. The content is inside the box and inaccessible until it is copied out of the box (by calling the get method); the copier becomes contaminated at that point.

10.3 Shared Queues

Shared queues are used for communication among the threads within a VN:

> final class AeolusQueue<T> extends AeolusShared

Queues have the following constructors and methods:

> AeolusQueue<T>( ). Creates a new empty AeolusQueue<T> object, with copies of the caller's labels.
>
> AeolusQueue<T>(AeolusLabel s, AeolusLabel i). Creates a new empty AeolusQueue<T> object, with copies of the indicated labels. The caller's labels must be no more constrained than the indicated labels; otherwise throws InfoFlowControlException.
>
> void enqueue(T x). If x is null, throws NullPointerException; if the caller's labels don't allow the write, throws InfoFlowControlException. Otherwise copies x onto the top of the shared queue; the copy is a complete copy down to shared objects.
>
> T dequeue( ). If the caller's labels do not match those of the queue, throws InfoFlowControlException. Otherwise waits for the queue to be non-empty; then removes and returns the oldest entry on the queue.
>
> T dequeueNoWait( ). If the labels of the caller do not match those of the queue, throws InfoFlowControlException. If the queue is empty returns null else removes and returns the oldest entry on the queue.

In addition queues inherit the getSecrecy() and getIntegrity() methods from AeolusShared.

AeolusQueues are not serializable.

10.4 Shared Locks

Shared locks allow threads within a VN to synchronize:

final class AeolusLock extends AeolusShared

AeolusLocks have the following constructors and methods:

AeolusLock<T>( ). Creates a new AeolusLock<T> object with copies the caller's labels. The lock is initially unlocked.

AeolusLock<T>(AeolusLabel s, AeolusLabel i). Creates a new AeolusLock<T> object with copies of the indicated labels provided the caller's labels are no less constrained than the indicated labels. Otherwise throws InfoFlowControlException.

void lock( ). If the caller's labels do not allow the write, throws InfoFlowControlException. Otherwise, waits until the lock is available and then locks it on behalf of the caller.

void unlock( ). If the caller's labels don't allow the write throws InfoFlowControlException. If the lock isn't locked does nothing else unlocks the lock.

boolean tryLock( ). If the caller's labels don't match those of the lock, throws InfoFlowControlException. Otherwise if the lock is available acquires it and returns true else returns false.

In addition locks inherit the getSecrecyLabel() and getIntegrityLabel() methods from AeolusShared.

AeolusLocks are not serializable.

10.5 User-defined Shared Objects

Aeolus allows users to define new types of shared objects. For example a user can define a shared hash table; this is convenient as a way to store session state for currently active sessions.

Each shared object has a secrecy label and an integrity label and method calls to access or modify the state of the object are allowed only when the labels of the caller match those of the object exactly. We use this stringent rule because we cannot know whether the method is a reader or a writer; the rule ensures there is no information flow leak in either case.

The class defining the shared objects provides constructors and methods to handle the access to the state of the shared object. There is no need to worry about the label checking when defining such a class since this is taken care of automatically by the Aeolus platform.

To illustrate what such a class might be like, we show the interface of a simple hash table that provides a map from ints to values of some parameter type. This class has the form:

> final class SharedHT <T> implements AeolusShared

and here are specifications for some of its constructors and methods:

> SharedHT( ). Creates a new empty hash table of type SharedHT<T> with the labels of the caller.
>
> SharedHT(Label s, Label i). Creates a new empty hash table of type SharedHT<T> with the indicated labels, provided the caller's labels are no more constrained than the indicated labels; otherwise throws InfoFlowControlException.
>
> void insert(int key, T value). If the labels of the caller do not match those of this, throws InfoFlowControlException. Otherwise, adds the key-value pair to this.
>
> T getValue(int key). If the labels of the caller do not match those of this, throws InfoFlowControlException. Otherwise returns the value associated with key or throws NotInException if there is no entry for key.
>
> void remove(int key). If the labels of the caller do not match those of this, throws InfoFlowControlException. Otherwise removes the pair associated with key or throws NotInException if there is no entry for key.

In addition the class inherits the getSecrecyLabel() and getIntegrityLabel() methods from AeolusShared.

Shared objects can be running calls from different threads within their VN concurrently. Therefore they must be implemented to manage this concurrency. To do this they can use the Java lock of their object. (As mentioned in Section 2, other uses of Java synchronization are prohibited.) They can also be implemented using AeolusLock or using various classes in the JDK, e.g., SharedHT could be implemented using HashMap.

As mentioned earlier, shared objects cannot contain pointers to the heaps of threads. We guarantee this constraint by copying all arguments of shared object constructors and all arguments and results of calls to methods of shared objects. These copies are deep copies down to any contained shared objects (or safe-to-share objects as explained in Section 11).

Classes that implement shared objects must be final, may not contain inner classes, and all instance variables must be private. We enforce these constraints by load-time checking.

In addition, while a thread is running inside a shared object it is not allowed to change the thread's labels; this constraint ensures that any modifications of the object state will respect the object's labels, without the need to do label checks when these modifications occur. Furthermore the code is not allowed to do RPCs or forks, so that we only need to enforce the no-changing-labels constraint within the thread running the method call. If the thread makes a call that violates these constraints, the call terminates with AeolusException.

We ensure proper checking of information flow control constraints by code rewriting, done by the class loader, which wraps the calls of the user-defined methods. The added code checks the labels. In addition, if the label checks succeed the added code modifies the thread state so that the thread is constrained to not modify its labels or make RPCs or forks while running inside the shared object; these constraints are removed when the method call returns.

11 Safe Types

The previous section discussed the constraints on AeolusShared that are needed to prevent information flow control violations:

1. Shared objects are "well encapsulated": pointers from outside an AeolusShared object do not point to non-shared objects that are inside that shared object.

2. There are no pointers from shared objects to objects that belong to threads.

These constraints are enforced by copying arguments and results on all calls to methods and constructors of classes that implement subtypes of AeolusShared. The copies are deep copies down to any contained shared objects.

However copies are expensive and it is clear in some cases that they aren't needed. In particular if an argument or result is immutable, the copy isn't needed since sharing of an immutable object cannot lead to information flow control violations. It's true that such a pointer might allow one thread to access an immutable object belonging to another thread. But this sharing is safe: it provides no additional communication that wasn't possible by using a copy of the object instead. (There could be a covert channel through synchronization on the object, but as mentioned in Section 2, we remove all such synchroniztion code.)

Our platform avoids copying immutable types in both Aeolus (e.g., PID, AeolusTag) and Java (e.g., int, String). In addition, Aeolus provides a way for user-defined code to implement classes whose objects are safe-to-share without copying. It does this by defining an interface, AeolusSafe, which is used as a way to impose constraints on classes that implement its subtypes. These constraints are as follows:

1. All instance variables must be final.
2. All instance variables must belong to types recognized as being *safe-to-share*.
3. The class must not contain any inner classes.

The restrictions above are based on the notion of *safe-to-share*. This notion is more inclusive than AeolusSafe, since certain Java types are recognized as being safe-to-share. Figure 2 shows the hierarchy of the various types having to do with sharing and safety. The figure mentions AeolusClosure; this type is defined in Section 12. The type AeolusSequence is defined in Section 11.1.

The constraints on AeolusSafe classes are checked by our class loader. The class loader does not restrict the types of the instance variables to subtypes of AeolusSafe but uses the more permissive rule of safe-to-share. Thus an instance variable could be a String. In addition our platform examines calls to methods of AeolusShared classes (and also AeolusClosure classes), and avoids copying arguments and results that are safe-to-share.

An example of a safe class is a list class that adds an element to the list by creating a new list that contains that element plus those that were already in

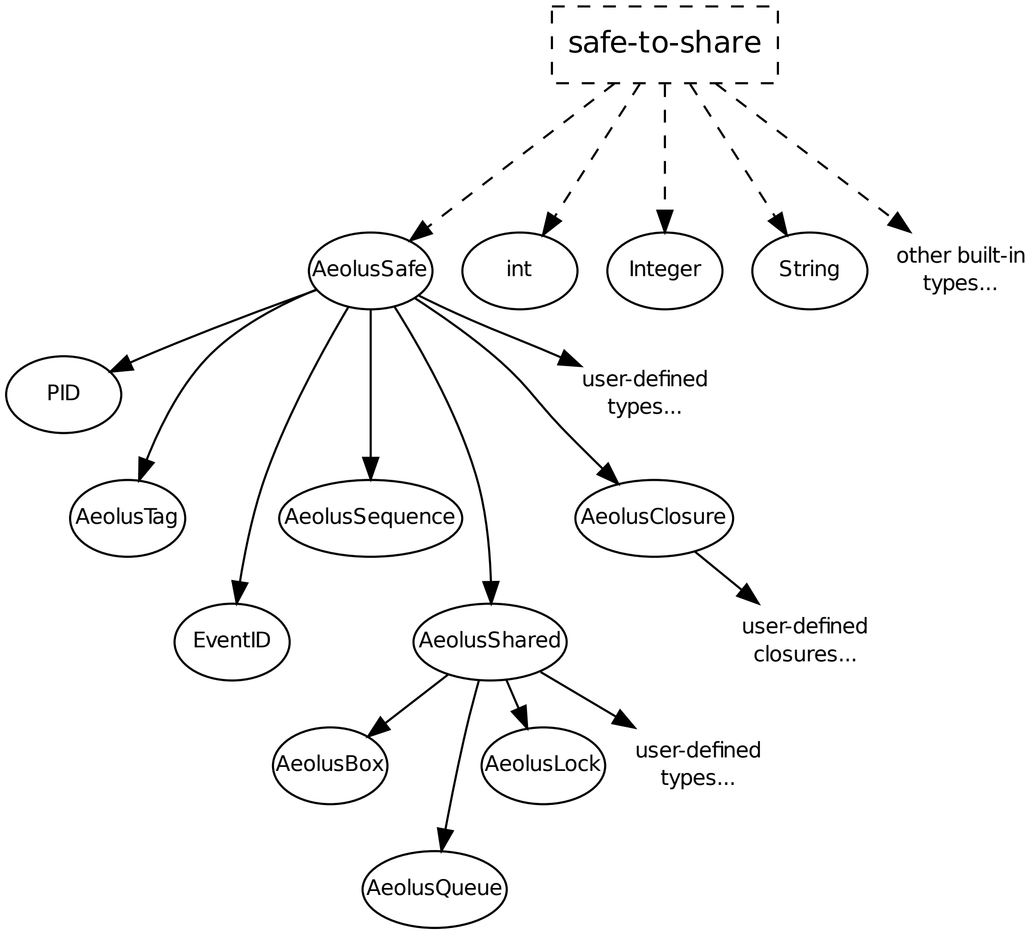

Figure 2: Hierarchy for types that are safe-to-share.

the list. Another example is a session class that maintains session state as a tuple containing the principal and tag for that user, plus a shared object that contains the current state of that session. Such a class can satisfy the rules for AeolusSafe given above provided the instance variables are final, because then any state changes are limited to modifications of the session-state shared object.

A safe-to-share class can be generic. If the class has instance variables whose type depends on a parameter, that parameter must be declared to extend AeolusSafe, since this way those instance variables will be recognized as safe-to-share by our class loader. If that parameter type extends more than one type, AeolusSafe must be listed first, since only the first listed type is retained in the bytecodes.

11.1 Safe Sequences

Aeolus provides an immutable AeolusSequence type that is safe-to-share:

- final class AeolusSequence<T extends AeolusSafe & Serializable>
  extends AeolusSafe implements Serializable

Here is a partial specification of its constructors and methods:

AeolusSequence( ). Returns a new empty sequence.

AeolusSequence(Collection<T> c). Returns a new sequence whose elements are those in c, in the order they occur in c.

T get(int i). If i is a legal index in this, returns the element at that position, else throws IndexOutOfBoundsException.

int size( ). Returns the size of this.

boolean equals (AeolusSequence<T> s). Returns true if the elements of this and s are pairwise equals.

AeolusSequence<T> set(int i, T x). Creates a new sequence that is a copy of this except that the ith element has been replaced with x. If i is not a legal index in this throws IndexOutOfBoundsException.

AeolusSequence<T> add(T x). Returns a new sequence containing all the elements of this, in their original order, followed by x.

AeolusSequence<T> remove(int i). Returns a new sequence containing all the elements of this in their original order except that the ith element has been removed. If i isn't a legal index in this, throws IndexOutOfBoundsException.

AeolusSequence is generic, but since its parameter must be AeolusSafe we are able to ensure that the restriction on the types of instance variables holds. Of course this means that AeolusSequence<String> will not work; however the programmer can implement a SafeString class to wrap the string and AeolusSequence<SafeString> will be accepted by the class loader.

12 Local Closures

Reasoning about the security of applications is easier if the code follows the *principle of least privilege*: running code with as little authority as possible limits the places where data can be leaked, and thus reduces the amount of code that needs to be inspected to ensure security.

Aeolus supports this principle by several mechanisms. VNs (which use their own authority and cannot make use of the caller's authority) are one such mechanism. Local closures, which are discussed in this section, are another, and reduced authority calls, which are discussed in Section 13, are a third mechanism.

Local closures allow code to run with a different principal than that of the caller. The principal is specified when the closure is created; this must be done by a thread that acts-for the principal. Later, methods of the closure can be called by a thread that doesn't have this authority, yet the calls will run with that principal.

Local closures are similar to VNs; they provide a way to bind authority to code so that when the code runs, it runs with that authority, rather than the authority of the caller. However, local closures exist within a single VN.

Local closures must belong to subtypes of AeolusClosure. AeolusClosure is a subtype of AeolusSafe, and therefore the restrictions on such classes apply here, e.g., instance variables must be final and their type must be a safe-to-share type:

```
abstract class AeolusClosure implements AeolusSafe
```

. AeolusClosure has the following constructors and methods:

protected AeolusClosure(PID p). If the caller doesn't act for p, throws AuthorityException. Otherwise binds the new closure object to p so that when it is called it will run with p's authority.

PID getPID( ). Returns the PID bound to this.

The class that implements the local closure defines the constructors and methods that provide the closure behavior. The definer need not worry about causing the method calls to run with the proper PID since this is handled automatically by the Aeolus platform.

Arguments to closure method and constructor calls are copied using deep copies down to any safe-to-share objects.

When a call is made to a method of a local closure, the thread switches to running with the principal of the closure. When the call returns, the thread switches back to the caller's principal and the labels of the closure are merged with those the thread had when it made the call (the merge is a union for the secrecy labels and an intersection for the integrity labels). Thus the closure can remove contamination it added, provided it has the authority, but it cannot be used to remove contamination that already existed in its caller.

An example of a class that implements a closure is a class whose objects encapsulate information about company job offerings; a closure like this would be useful in implementing a job service similar to what is provided by monster.com:

final class CompanyClosure extends AeolusClosure

The class might have the following constructor and method:

CompanyClosure(PID p, Jobs j). If the caller doesn't act for p, throws AuthorityException. Otherwise, creates a new company closure bound to principal p, and containing information about job openings in that company.

JobList evalResume(Resume r). Returns a list of jobs that are matches to the information in r.

Here Jobs might be a subtype of AeolusShared, and the argument j to the constructor would have a secrecy label that includes the tag for that company.

The CompanyClosure method would be called while the calling thread is contaminated with some tags for which the closure doesn't have authority, e.g., the caller's secrecy label would contain the tag for the user whose resume is being evaluated. The closure uses its own state to determine what to do; in doing so the thread running the closure call becomes contaminated with its own tag. Thus in this example the thread will be contaminated both by the user-tag and the company-tag. When it is ready to return, the thread can remove the company-tag using the closure's authority; this way it provides a result that its caller can use.

Since the closure does not have authority for other tags, e.g., the user-tag, the thread running the closure call cannot remove them, and therefore it cannot expose the information in its arguments. The thread could store this information in a shared object or a file, but only if that object's labels are no less constraining than its own labels; thus the thread running a call on the company closure could not store user resume information in a shared object that doesn't contain the user-tag. Furthermore, the thread cannot record this information inside its own object, since this is immutable: a closure class must implement a subtype of AeolusSafe. Therefore the closure call is unable to leak information about the user resume.

Closures are not serializable and cannot be sent in messages.

As with subclasses of AeolusShared, we provide the proper semantics for closure method and constructor calls by wrapping the calls. The wrapping is done by the class loader by rewriting the Java byte code. In addition the class loader inspects the code of the closure class to ensure it satisfies the constraints imposed on classes that implement subtypes of AeolusSafe.

13 Reduced-authority Calls and Forks

This section describes reduced authority calls, a third mechanism that supports the principle of least privilege. Reduced authority calls allow a thread to temporarily reduce its authority. For example, while running a third-party statistics package that evaluates records for all patients in a clinic to obtain epidemiological information, a thread can run with no authority, thus ensuring that the patient data cannot be leaked by the code that does the computation.

A thread can reduce its authority by using the following AeolusLib methods:

static <T> T call(Callable<T> x, PID p). Calls x.invoke() in the caller's thread to run with PID p and the current labels provided the caller acts-for p; otherwise throws AuthorityException. If the call terminates with an exception, throws java.lang.Exception. When the call finishes (either normally or via an exception), switches back to the caller's PID and returns the result (in the case of a normal return).

Forks are done by a similar mechanism; we require the caller to specify the authority explicitly, and often this will be less authority than what the caller has. Forks are accomplished by calling methods of AeolusLib:

static void fork(Runnable x). Forks a new thread in the VN of the caller, to run with the caller's principal and copies of the caller's labels. Copies x into the heap of the new thread and then calls its run() method. This is a full copy down to objects that are safe-to-share.

static void fork(Runnable x, PID p). Forks a new thread in the VN of the caller, to run with PID p and copies of the caller's labels, provided the caller acts-for p; otherwise throws AuthorityException. Copies x into the heap of the new thread and then calls its run() method. This is a full copy down to objects that are safe-to-share.

A fork is executed as follows: a new thread is created, the Runnable object x is copied into the heap of the new thread, and the call to the run() method then happens in the new thread. If x is safe-to-share the cost of the copy can be avoided.

14 Files

Each node within an Aeolus deployment can have a mounted Aeolus file system, and thus an application running in the deployment can use these file systems. A file system is identified by the DNS name of the Aeolus node where it is mounted.

Aeolus files provide an interface based on the Java interface to files. However, files in Aeolus are labeled and uses of files are checked to ensure that the information flow rules are obeyed. In addition we provide only a subset of the methods that are present in the related Java type.

14.1 Label Restrictions

Every file and directory has a secrecy label and an integrity label. These labels are defined when the file or directory is created and they are immutable.

When a file in a file system is first used, e.g., when it is opened for reading or writing, the the pathname of the file has to be interpreted by reading all the directories along the path. To ensure that the labels of the directories allow pathname interpretation without errors, Aeolus imposes the following constraints on labels:

1. Growing confidentiality. The secrecy label of a directory is contained in the secrecy label of any file or directory that is an entry in that directory.
2. Reduced integrity. For all non-root directories, the integrity label of the directory contains the integrity label of any file or directory that is an entry in that directory.

The constraints ensure that the integrity of the pathname is at least as good as that of the file or directory named by that pathname. The constraints also ensure that if a thread's labels allow it to read a file, it will be able to read the directories along the path to that file. For a write, the rules ensure that there are thread labels that both allow the path to be read and the file to be written: the thread's labels must be no less constrained than those of the file's parent directory, and no more constrained than those of the file, but such labels exist since the parent directory's labels are no more constrained than those of the file.

The constraints are enforced by having a special rule for directory modifications. A thread can modify a directory only if its secrecy label matches that of the directory *exactly*; exact match is needed because when a thread modifies a directory, it can read the directory, e.g., if the name selected for the file being inserted is already in use, the thread will learn this. For a non-root directory we also require exact match for the integrity labels, but we do not require this for modifications of the root directory (since the hierarchy constraint implies that the integrity label of the root directory contains all possible labels). Instead when the root directory is modified we require that the integrity label of the thread doing the modification must include that of the file or directory being added or deleted.

14.2 File Access Exceptions

Calls to access files in Aeolus succeed only when certain constraints are satisfied; otherwise the call throws an exception. The constraints and exceptions are as follows:

File system constraint. If the file or some parent directory doesn't exist, throws java.io.FileNotFound exception. If a failure occurs while communicating with hostname, throws AeolusException. If an error occurs while reading or writing the file or some directory, or if an attempt is made to use a file as a directory, or a directory as a file, throws java.io.IOException.

Label constraint. If the thread's labels don't allow the requested operation, throws InfoFlowControlException. For a file or directory read, the thread's labels must be no less constrained than those of the file. For a file write, the thread's labels must be no more constrained than those of the file and no less constrained than those of the parent directory. For a directory write, the labels of the file being added or removed must be no less constrained than those of the directory and the thread's secrecy label must match the directory's secrecy label. In addition, if the modification is to a non-root directory the thread's integrity label must match that of the directory; if the modification is to the root directory, the thread's integrity label must include that of the file or directory being added or removed.

In the following sections we do not mention these exceptions explicitly.

14.3 File Attributes and Name Space Management

Aeolus provides access to certain file attributes and also provides methods to manage the namespace of a file system through the type AeolusFile. AeolusFile has the following constructor:

> AeolusFile(String hostName, String pathName). Identifies the file of interest by providing its pathname and the file system that contains it.

The call to the constructor identifies the file of interest by indicating the DNS name of the host where the file system that contains the file resides, and the pathname of the file within that file system. Howevers the constructor does *not* look up or access the file. Instead access happens when methods of the AeolusFile object returned by the constructor are called, and it is at this point that the pathname and label constraints are checked.

The AeolusFile methods are as follows; in all cases these calls throw exceptions if the file system constraint or the label constraint is violated:

AeolusLabel getSecrecy( ). Returns the secrecy label of the file identified by this, provided the file system constraint is satisfied and the thread's labels allow it to read the parent directory that contains the file.

AeolusLabel getIntegrity( ). Returns the integrity label of the file identified by this, provided the file system constraint is satisfied and the thread's labels allow it to read the parent directory that contains the file.

boolean createNewFile(AeolusLabel s, AeolusLabel i). Creates a new empty file corresponding to the file identified by this, with the indicated labels, provided the file system constraint and the label constraint for a directory modification are satisfied. Returns true if the file is created; if the parent directory already contains a file of the given name returns false.

boolean mkDir(AeolusLabel s, AeolusLabel i). Creates a new empty directory corresponding to the pathname and file system identified by this, with the indicated labels, provided the file system constraint and the label constraint for a directory modification are satisfied. Returns true if the directory is created; if the parent directory already contains a file of the given name returns false.

String[ ] list( ). Lists the names of entries in the file identified by this (as entries in the result array), provided the file system constraint is satisfied, the file exists and is a directory, and the thread can read the directory. The names are relative to the name of the directory; the pathname for the entry can be formed by combining the returned name with the pathname of the directory. Returns null if the file isn't a directory.

boolean delete(). Deletes the file identified by this, provided the file system constraint and the thread label constraint for a directory modification are satisfied. Additionally if the file being removed is a directory, it must be empty; otherwise throws java.io.IOException.

14.4 Input File Streams

Files can be read using the AeolusInputStream type. This type has the following constructor:

AeolusInputStream(String hostname, String filePath). Creates a file stream for reading from the indicated file, provided the file system constraint is satisfied and the thread can read the file.

An input file has a cursor associated with it. When the file stream is created the cursor is at the first byte of the file.

Input file streams provide the following methods; in all cases these methods will throw exceptions if the file system constraint or the label constraint is not satisfied.

int read(byte[ ] buffer). Reads buffer.length bytes of the file, or up to the end of file, starting at the cursor, provided the thread's labels allow the read and the file system constraint is satisfied. Advances the cursor by the number of bytes read and returns a count of the number of bytes read, except that if a previous read encountered an end of file, returns -1 and does not advance the cursor. If the stream is closed throws java.io.IOException.

int read(byte[ ] buffer, int offset, int count). Reads count bytes, or up to the end of file, starting at the cursor, into the buffer at the indicated offset, provided the file labels allow the read and the file system constraint is satisfied. Advances the cursor by the number of bytes read and returns a count of the number of bytes read, except that if a previous read encountered an end of file, returns -1 and does not advance the cursor. If the stream is closed throws java.io.IOException. If the buffer doesn't have enough room to hold the number of bytes read, throws IndexOutOfBoundsException.

void close( ). Closes the file stream provided the file system constraint is satisfied. After this point reads will fail. If the stream is already closed, throws java.io.IOException.

14.5 Output File Streams

Files can be written using AeolusOutputStream. Output file streams also have a cursor. When the stream is created, this cursor either points to the start of the file, or if the file is opened in append mode, it points to the end of the file.

AeolusOutputStream has the following constructor:

AeolusOutputStream open(String hostname, String filePath, boolean appendMode). Creates an output file stream for writing to the indicated file provided the file system constraint is satisfied and the thread can read the parent directory and write the file. If append mode is on, the file cursor is set to the end of the file; otherwise it is set to the start of the file.

Output file streams provide the following methods; in all cases these methods will throw the indicated exceptions if the file system constraint or the label constraint is not satisfied:

void write(byte[ ] buffer). Writes buffer.length bytes from the buffer to the stream starting at the cursor, and advances the cursor by the number of bytes written, provided the thread's labels allow the write and the file system constraint is satisfied. If the stream is already closed, throws java.io.IOException.

void write(byte[ ] buffer, int offset, int count). Writes count bytes to the stream from the buffer starting at the given offset, provided the thread's labels allow the write and the file system constraint is satisfied. If the stream is already closed, throws java.io.IOException. If the buffer doesn't have count bytes starting at offset, throws IndexOutOfBoundsException.

void close( ). Closes the file stream provided the file system constraint is satisfied. After this point the file cannot be written. If the stream is already closed, throws java.io.IOException.

15 Communication across a Deployment Boundary

Threads in VNs can communicate with programs running outside a deployment using Java sockets. In addition they can read and write external files and devices using Java I/O.

However, communication across a deployment boundary is constrained. We cannot vouch for the secrecy of information written to the outside of a deployment; for this reason we allow output only if a thread's secrecy label is empty. Also, we cannot vouch for the integrity of data entered from the outside; for this reason we allow input across a deployment boundary only if the thread's integrity label is empty.

The constraints on external communications are enforced by our custom Java SecurityManager.

16 Auditing

Aeolus provides an auditing subsystem that logs all events having to do with security. For example, each time a thread does a declassify or makes a reduced authority call, this is logged.

Every event has an eventID and it indicates its immediate predecessor events using their IDs. Almost all events have as one of their immediate predecessors the previous event logged in the thread whose current event is being logged. However some events have additional predecessors. For example, when an entry is dequeued from an AeolusQueue, there are two predecessors: the prior event in the thread running the dequeue, and the event recording the enqueue of that entry.

Aeolus allows applications to log application-specific events in the audit trail. This can be done by calling the following methods of AeolusLib:

> static EventID createEvent(String appOp, List<String> appArgs). Adds an event ⟨"appEvent" pred appOp appArgs⟩ to the audit trail as the next event of the calling thread, where pred (the predecessor of this event) is the previous event in the thread of the caller. Returns the EventID of the new event.
>
> static EventID createEvent(String appOp, String appArgs, List<EventID> eList). Adds an event ⟨"appEvent" ⟨pred, eList⟩ appOp appArgs⟩ to the audit trail as the next event of the calling thread. Here pred is the previous event in the caller's thread, and elist is a list of other events that are considered by the application to be predecessors of the event being logged. Returns the EventId of the new event.
>
> static EventID getEventID( ). Returns the EventID of the most recent event logged for the caller.

The appOp should uniquely identify the operation so that it can be distinguished from other application-level operations both in this application and any others running in the deployment. appArgs provides additional application-level information about the event being logged.

EventIDs are serializable and cloneable and can be tested for equality; they have no other methods.

More information about auditing can be found in [4, 1, 3].

17 Setting Up a Deployment

The discussion in the previous sections assumes an Aeolus deployment exists and its membership isn't changing. This section describes how a deployment starts and how membership changes. More detailed information can be found at www.pmg.csail.mit.edu/aeolus/.

When an Aeolus deployment starts up, it consists of a single node on which Aeolus has been installed and the AS (authority state) has been started. This node runs with the root PID (which will act for all other PIDs created by code running within the deployment). The authority state is initially empty, and the audit trail exists and is also empty.

To add a new node to a deployment, the owner of the node must first set up the node to run Aeolus, and must put the code of an initial VN on the node. Then the owner starts Aeolus running on the node.

Aeolus sends a message to the AS informing it of the new node. The AS adds the node to the deployment, creates a "node-root" PID for the new node, and returns this PID to the new node. Then Aeolus starts up process at the new node; as mentioned this process runs the Aeolus code on top of the JVM, and all VNs at the node will run on top of Aeolus in this process.

Next Aeolus starts up a first VN; this VN runs the pre-loaded code with the authority of the node-root PID. The code is first checked by our class loader and the VN is created only if the check succeeds. In this case, Aeolus creates a thread within this process and starts it running the main method. The thread runs with the node-root PID and null labels.

In addition, if the node hosts a file system, the file system is mounted at the node and an Aeolus process is created to handle interactions with the file system; for details see [3].

It's also possible to start up one VN at the AS through a command at the console. When the AS was created, Aeolus also created the process at the AS where VNs can run, but no VNs were created at this point. The console command indicates the class that the single VN will run. This code is checked by the class loader. If the check succeeds, Aeolus creates the new VN and also creates a thread and starts it running the main method. This VN runs with the authority of the root PID, and the thread runs with the root PID and empty labels.

The only way to get more than one VN at a node is to use launch (see Section 8).

The design of Aeolus calls for each node in a deployment to have a public/private key pair that is used to allow encryption of messages exchanged between the nodes in the deployment. However our current implementation does not provide such encryption.

18 Storage at the AS

The authority state is stored in *blocks* at the AS. Each thread has a current block; it can control which block is its current block using the following methods of AeolusLib:

> static void createBlock( ). Creates a new block and makes it the current block of the creating thread, provided the thread's secrecy label is null; otherwise throws InfoFlowControlException.
>
> static void makeCurrentBlock(PID p). If the call returns normally, makes the block that contains principal p be the current block. The thread's secrecy label must be null; otherwise throws InfoFlowControlException. Also, the thread must act for p, else throws AuthorityException.

These calls can fail with AeolusException if there is a problem communicating with the AS.

Newly created principals are stored in the current block and tags are stored in the block of the PID that creates them. Delegations are stored in a single block if the delegator and delegatee principals are in the same block. Otherwise both blocks record information about the delegation.

When a thread starts running, its current block is the one that contains its principal.

Further Reading

[1] A. Blankstein. Analyzing audit trails in the Aeolus security platform. Master's thesis, MIT, Cambridge, MA, USA, June 2011.

[2] W. Cheng, A. Blankstein, J. Cowling, D. Curtis, V. Popic, D. R. K. Ports, D. Schultz, L. Shrira, and B. Liskov. Abstractions for usable information flow control in Aeolus. In submission.

[3] F. P. McKee. A file system design for the Aeolus security platform. Master's thesis, MIT, Cambridge, MA, Sept. 2011.

[4] V. Popic. Audit trails in the Aeolus distributed security platform. Master's thesis, MIT, Cambridge, MA, Sept. 2010.