Summarizing Audit Trails in the Aeouls Security Platform

by

Wissam Jarjoui

S.B., C.S M.I.T., 2011

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2012

Author .....
Department of Electrical Engineering and Computer Science
August 15, 2012

Certified by .....
Barbara H. Liskov
Institute Professor
Thesis Supervisor

Accepted by .....
Christopher J. Terman
Chairman, Masters of Engineering Thesis Committee

Summarizing Audit Trails in the Aeouls Security Platform

by

Wissam Jarjoui

Submitted to the Department of Electrical Engineering and Computer Science
on August 15, 2012, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

For an Aeolus user, analyzing an Aeolus event log can prove to be a daunting task, especially when this log grows to include millions of records. Similarly, storing such an event log can be very costly. The system I present in this thesis provides an interface for the creation of user-defined summaries of the Aeolus audit trails, as well as truncation of the event log, making it easier for to analyze the Aeolus event log and less costly to analyze events of interest. This is done through through the use of what we term a *Summary System* and *Summary Objects*. I present the system in the context of a sample application based on *www.mint.com*. The system is an extension to the Aeolus library, it is written in Java and uses a PostgreSQL database as it's primary database.

Thesis Supervisor: Barbara H. Liskov
Title: Institute Professor

Acknowledgments

First and foremost, I would like to thank my advisor, Prof. Barbara Liskov, for the mentorship she gave me in my two years here at PMG. Her guidance helped me navigateand learn a lot about systems security, and went beyond just this project. I am truely grateful for the understanding, patience and investment she put in while helping me see this project through; I hope that one day I would be able to reflect those traits as a leader.

A big thank you to my colleague, David Schultz, for his tremendous help in this project and my personal learning; from suggesting and brainstorming ideas to improve my project, to explaining technical concepts, and being a strong source of knowledge and inspiration and a good role model in general. David, it has been a pleasure working with you.

I would also like to thank James Cowling, Dan Ports and Barzan Mozafari and everyone in 32-G908 for all the fun conversations and advice they provided. I am very fortunate to have had the chance to hear and learn from their experiences, and be inspired by them.

As this thesis culminates five years for me at MIT, firstly I would like to take this chance to also thank my friends for being their for me, and for offering me help even when I didn't ask. Their presence was a home away from home, and their support guided me in times of uncertainty. Secondly, I would like to thank my family; I would not be where I am today if it were not for them. I thank them for their support and encouragement, for worrying about the things I wouldn't, and for setting the bar high. In one way or another, I will always have something to learn from them.

Finally, I would like to thank MIT for all what it has given me – Anne Hunter, my advisors, my professors and my TAs. I thank Stata and the Infinite for being monuments of my experience here at MIT, and I thank the Charles River, for being the source of calmness in a very busy time. They will all be missed.

Contents

Chapter 1

Introduction

Security of confidential online information, such as medical records and financial data, is a very important problem. Recent research has focused on Cecentralized Information Flow Control (DIFC) as the most promising approach to enable application developers to secure information. DIFC is based on the principle that the system tracks information as it flows through the system, and only allows release if the releaser has sufficient authority. Additionally DIFC provides this ability in a fine-grained way, so that security policies can be tailored to the needs of individual users and organizations.

This thesis extends the security support provided by the Aeolus platform. Aeolus is a platform that combines DIFC and an intuitive security model framework to make it more convenient for developers to build secure applications on a distributed system. Additionally, Aeolus provides automatic auditing of every security related event that occurs while an application runs; also it provides a way for applications to log additional events that are meaningful at the application level. The audit trail is an important component of security to allow discovery of errors that cause security policies to be subverted; the audit trail can also be used to discover attacks.

An issue in any auditing system, including the Aeolus system, is that the audit trails can become extremely large. This is especially true if the system being audited is large, e.g., has millions of users, and long-lived. Therefore a way of re-

ducing the stored information and making the important information more easily accessible is needed.

This thesis addresses this problem. It provides a framework that allows groups of audit events to be summarized: the important content of the group of events is captured in a *summary event*, which is much smaller than the group. Once information has been summarized, the base events that underlie the summary can be moved to archival storage, or even deleted. An additional benefit is that summary events are more accessible than the base events they summarize because they can be defined to explain the information in application-specific terms.

A final point is that the production of the summary events is itself controlled by information flow. This is important because, everything a system does is detailed in the audit log, and therefore, there is a potential for information leaks if the log could be accessed by an unauthorized party.

The next two sections describe the motivation and scope of this system with the help of some real-life examples.

1.1 Motivation for Summarization and Archival

Summarization allows developers to group information that is spread out over the Aeolus audit trails into smaller space. This allows for future archival or truncation of the Aeolus log, as well as sets the stage for writing more performant queries.

Take for example a bank's web administrator who is interested in detecting suspicious activity on customers' bank accounts. One way to do this is to produce a plot of the number of outbound transfers carried through a user's account for each week in the last year. Any spikes in the rate of outbound transactions per week could mean that a user's account has been compromised by an attacker. Finally, the administrator might also wish to produce such a plot at the end of each week.

Producing such information is possible through the current auditing mechanisms available in Aeolus. However, in order to accomplish such a task, the adminsitrator's audit will have to scan all events in the system in the past year, which

could span millions of events, once a week.

There are two points to be noticed in this example:

1. Events in the system cannot be archived or deleted unless they are a year old. In a system where there are hundreds of users, this can be very costly.

2. There is no built-in support for reuse of previous computation - the information needed to produce the plot would have to be gathered from scratch every week.

Summarization solves the first problem by allowing the web administrator to condense the information into *summary events*, reducing the storage requirement to store the necessary information. For example, the web administrator could summarize a user's weekly activity by storing the number of outbound transfers they made.

Summarization solves the second problem by allowing the web administrator to store weekly computations as summaries themselves, and simply reusing these summaries in future weekly audits. For example, the web administrator could store the average number of outboud transfers a user has made per week in the last year and use that as a benchmark for future user activity [1].

1.2 Scope

Our system, in similar fashion to Aeolus, assumes an application developer to be active and aware of the workings of Aeolus as a whole. We do not intend for our system to impose any restrictions on the developer, and hence the developer would have to be careful with the summarization power granted to them through the our system. For example, summaries could be an added source of information leakage if not used correctly.

[1] These summaries could be considered *summaries of summaries* as they further summarize previously-summarized information

Thesis Outline

The remainder of this thesis is organized as follows: Chapter 2 presents an overview of the Aeolus security system. Chapter 3 presents a sample Aeolus application based on the financial management service Mint.com. Chapter 4 describes the summarization models and interfaces. Chapter 5 describes the implementation details of the system. Chapter 6 evaluates the ease of use of the system as well as overall system performance. Chapter 7 discusses related work in streaming databases. Chapter 8 presents some topics for future work and reviews the contributions of this thesis. Appendix A details the summarization and archiving interface and presents some examples.

Chapter 2

Aeolus System Overview

This chapter presents an overview of the Aeolus security platform, on which our summarization system is based, and highlights relevant details such as log collection. More complete descriptions of the Aeolus security platform and it's log collection and analysis can be found in Cheng [3], Popic [5] and Blankstein [1].

2.1 System Architecture

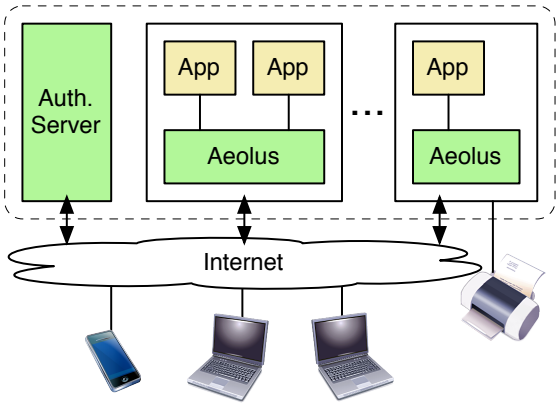

Figure 2-1: High level overview of Aeolus system architecture.

The Aeolus architecture is shown in figure 2-1. The system consists of many nodes, each of which is trusted to enfoce Aeolus's information flow rules.

Aeolus tracks information flow within each system node and between system nodes. Nodes in the system communicate via RPC messages, those messages are

encrypted to protect their secrecy and integrity. Nodes outside the system are considered to be untrusted: information can flow outside of the system if it is uncontaminated, and information arriving from the outside is marked as having no integrity.

Threads in Aeolus run on behalf of *principals*. The ability of a thread to carry out *privileged operations* is determined by the *authority* of the principal it runs on behalf of. Aeolus tracks the authority of principals in the *authority state* stored at the authority server (AS). Principals, privileged operations and authority are described in sections 2.2.1, 2.2.2 and 2.2.3 respectively.

2.2 Information Flow Model

This section describes the basic concepts and rules of the Aeolus security model.

2.2.1 Prinicpals, Tags and Labels

This section describes the basic concepts and rules of the Aeolus security model.

Aeolus employs an intuitive security model to implement information flow control. The model revolves around three key concepts: *principals*, *tags* and *labels* [2]. Principals represent entities in the system that create, modify and share information. Tags represent security categories of information. A principal authoritative for a certain tag can modify or share information categorized by that tag. Labels are sets of tags and are used to determine whether information can flow from a source to a receiver based on the information flow rules described in the following section.

2.2.2 Information Flow Rules

Aeolus allows information to flow from a source S to a destination D only if the following rules are satisfied:

$$\begin{aligned} \text{SECRECY}_S &\subseteq \text{SECRECY}_D \\ \text{INTEGRITY}_S &\supseteq \text{INTEGRITY}_D \end{aligned}$$

Threads in an Aeolus node run with security and integrity labels associated with them. In Aeolus, a thread cannot modify labels of files or shared memory objects, and hence it would have to modify it's own labels in order to read or write data. Certain label manipulations, called *privileged manipulations* are unsafe because they remove constraints on information flow:

1. *Declassfication* Remove a tag from a secrecy label.

2. *Endorsement* Add a tag to an integrity label.

A thread is allowed to carry out privileged label manipulatinos if it is running on behalf of a principal that has authority for the tags affected by these manipulations.

2.2.3 Authority

Authority determines whether a thread can perform privileged label manipulations. Authority starts with tag creation: when a thraed creates a tag, its principal has authority for that tag.

Subsequently, the Aeolus authority state can be modified through *grant*, *act for* or *revoke* operations. Grant operations allow a principal to delegate authority for a particular tag to another principal. Act for operations allow a principal to delegate all of its authority to another principal. Revoke operations remove *act for* and *grant* links between principals. To avoid covert channels, Aeolus only permits threads running with null secrecy labels to modify the authority state.

2.2.4 Compound Tags

Applications frequently have sets of tags that are closely related. In order to simplify the authority structure of the application, Aeolus allows for tags to be grouped upon creation using *compound tags*. For example, in a medical clinic, patient data tags are *subtags* of an *ALL-PATIENT-DATA* tag, a *super tag*.

A principal authoritative for a super tag is also authoritative for all of its subtags. Similarly, a having super tag in a thread's secrecy label is equivalent to having all subtags in its secrecy label. This reduces label size substantially, and makes label manipulations involving paricular groups of tags less expensive.

Figure x shows a sample authority state graph, detailing *act for*, *grant* and *subtag* relationships in the financial services application is described in chapter 3. [describe the graph]

2.3 Programming Model

This section explains the programming abstraction Aeolus provides, and how they support DIFC.

2.3.1 Threads and Vitual Nodes

Virtual nodes, shown as applications in figure y, communicate via RPC and have many threads inside. A Virtual node is given a principal when it is created and all threads run with this principal or some prinicipal it acts for. An RPC is run in its own threads with the vn prinicipal but the labels of the claler; on the return the labels are sent back to the caller where they are merged with those of the caler: a union of the secrecy labels and an intersection of the integrity labels. Then the caller continues with its own principal.

2.3.2 Shared State Objects and Boxes

Within a virtual node, threads can share state via special Aeolus shared objects. Aeolus gives threads access to a *root* shared memory object, which has null labels, which threads can modify to point to other Aeolus shared objects and share them.

Aeolus shared obects have immutable label associated with them, and Aeolus ensures that the rules in 2.2.2 are respected. User's can create their own shared objects, however, Aeolus ensures that label manipulations do not take place inside the object's methods.

2.3.3 Authority Closures and Reduced Authority Calls

Aeolus provides developers with *authority closures*. An authority closure is an object bound to a principal at the moment of creation. Threads can later run the closure's methods with the closure's principal. Authority closures allow threads to allow threads to process confidential information without being exposed to the information itself.

Aeolus also provides a mechanism for threads to reduce their authority, called *reduced authority calls*. To make a reduced authority call, a thread specifies a function and a principal to run it with. The calling thread's principal must act for the principal of the reduced authority call.

[labels merging?]

With those two mechanisms in place, Aeolus makes it possible to ensure that each part of the application runs with only the authority it needs. Aeolus also provides a principal that is not authoritative for any tags, P_{PUBLIC}, for developers to use when they need to ensure that a part of a program cannot leak any information.

2.3.4 Files

Aeolus provides a network file system that enforces DIFC. Similarly to shared memory objects, files have immutable labels and the rules in 2.2.2 apply for information flow in and out of them.

Files are yet another way (RPCs, shared memoery objects) through which Aeolus allows threads to communicate. A complete description of the Aeolus file system API is presented in McKee [4].

2.3.5 Log Collection

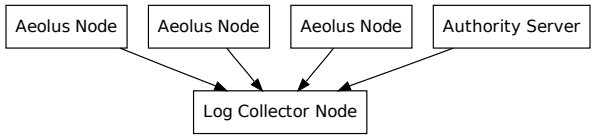

Figure 2-2: Flow of logs to the log collection system.

Aeolus provides automatic auditing of every security related event that occurs while an application runs; also it provides a way for applications to log additional events that are meaningful at the application level.

Aeolus distributes log collection across all virtual nodes in the system. Aeolus virtual nodes send events that occur locally to the log collector as shown in figure 2-2. The log collector stores the events for later processing and analysis, as explained in [1]. Figure y This happens automatically for all calls to the Aeolus runtime, however, Aeolus also provides application-level logging to allow developers to log their own events using the following interface:

```
AeolusLib.createEvent(String desc, List<String> args)
```

This creates a new record in the Aeolus event log. The *desc* and *args* argument are stored as attributes for that record, allowing the developer to query the event log based on the arguments they provided.

Aeolus Event Attributes

Aeolus stores different attributes for different events that take place in the system. Blankstein [1] provides a complete description of those attributes, but here we will highlight the ones more relevant to the useof summarization:

event_counter

The event counter uniquely identifies an event, and provides an ordering for the occurence of the event, i.e. events with a higher eventcounter took place after events with a lower eventcounter.

timestamp

This is the time at which the event was registered by the client virtual node.

secrecy

This is the secrecy label of the thread which caused the event, just before the event took place.

integrity

This is the integrity label of the thread which caused the event, just before the event took place.

opname

This field specifies the type of the event, e.g. a DECLASSIFY event, or a SENDRPC event. Those types are detailed in [1]. Application-level events created with the above events all have one opname.

app_args

This field holds the arguments specified for an application-level event (the second argument in the interface shown above).

running_principal

This field holds the principal of the thread that caused the event.

It is important to note that the *secrecy* and *integrity* attributes are particularly import to prevent information leakage in the system: a thread can only read those event records if the DIFC rules in 2.2.2 are satisfied.

[mention this ideally runs on david's db?]

Chapter 3

Mint

Prior to the development of the summarization system, it was important to develop a sense for what is needed from such a system, and what problems can it solve. To answer these quesitons, I developed an Aeolus application based on the financial management service *mint.com*. The *Mint* application served as a case study and guided the development of the summarization system.

In this chapter I present the Mint application model as well as some examples to how a summarization system could be used for this application.

3.1 Mint Model

Mint.com is an financial management service that provides its users with the ability to monitor their bank accounts across different banks from one place. It also allows users to run transaction analysis tools that will access the information of certain transactions for each bank for the user, and generate a result. In addition, Mint runs aggregate analysis for all user accounts.

For example, Bob could sign up on www.mint.com and add his Bank of America account and his US Bank account for monitoring. This will cause our application to store credentials for both of these banks for Bob. Bob can then request a graph that presents his expenditures on food in the last week. Bob's actions would also affect the general aggregate behavior of the Mint users.

This example gives us a starting idea of how should we build our authority structure. For instance, there should be a *BOB-DATA* tag associated with Bob's information (e.g. expenditure statistics, transactions, etc.) that only a principal running on Bob's behalf can declassify. Note that Bob's password would never be revealed to Bob himself. Similarly, there's a natural grouping of user data tags, and hence a super tag, *ALL-USER-DATA*, could be useful to run the aggregate analysis tools for Mint.

In this section we examine the security model of the application more closely.

3.1.1 Authority State Model

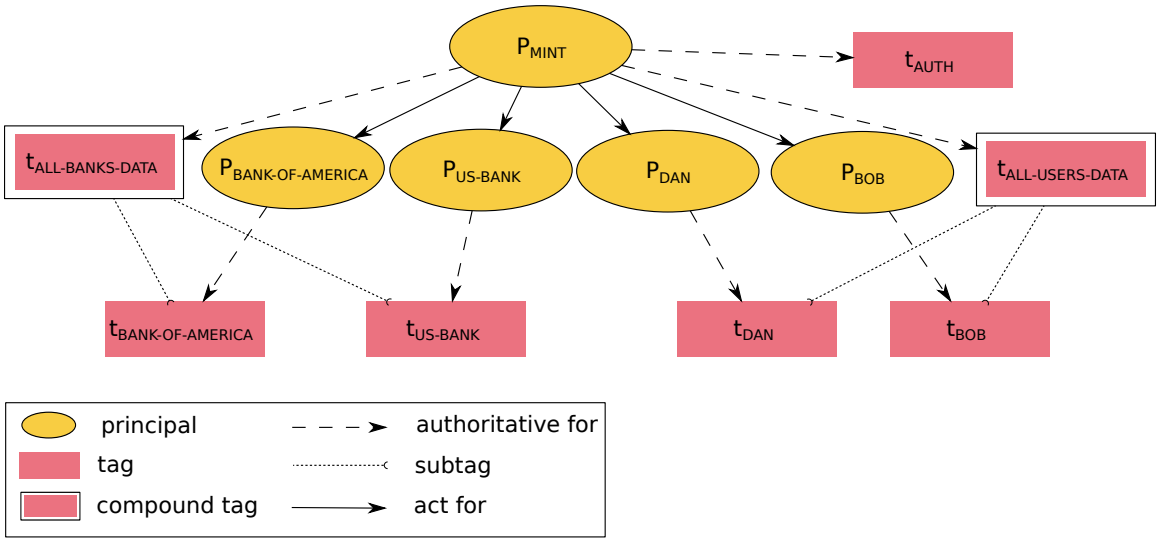

Figure 3-1: Mint Authority State Model. In this example, two banks are registered at the application: Bank of America and US Bank. Users Bob and Dan have both signed up. Principals are denoted $P_{<\text{name}>}$ and tags are denoted $t_{<\text{name}>}$.

In this section we identify the prinicpals running in the system, the tags associated with different data, and the relationships between them. Figure 3-1 shows an overview of the authority state model for Mint.

An intuitive way to list all necessary[1] principals, is to think of the different "clearance levels" in the system: some information should be accessible by a user,

[1]Neccessary by good design. The application could run with just one principal.

some by a bank, and some information should not be accessible at all. In the light of these requirements, the application employs the following principals:

Mint principal

This principal is authoritative for all tags in the system, and is used for aggregate analysis of user data and user authentication.

Bank principals

Each of these principals is authoritative for a particular bank's tag.

User principals

Each of these principals is authoritative for a particular user's tag.

Public principal

Described in ref 2.3.3, this principal is used when no privileged operations are necessary to ensure that no information leaks.

Similarly, to separate data in the system according to it's purpose, (e.g. user password, credentials to download bank transactions, etc..), the application employs the following tags:

User data tags

Each of these tags is associated with data carrying a particular user's information. Reading (or modifying) user information requires declassifying (or endorsing) a user data tag.

Bank data tags

Each of these tags is associated with data carrying a particular bank's information. Reading a user's bank credentials requires declassifying a bank data tag. This tag is important so that only a Bank's closure is able to access a user's bank password.

All-Users-Data tag

This tag is a super tag for all the user tags. Running aggregate analysis on Mint user accounts requires declassifying this tag.

All-Banks-Data tag

This tag is a super tag for all the bank tags. Adding a new bank to Mint requires endorsing this tag.

Authentication tag

This tag is associated with user's Mint passwords. Authenticating a user's mint credentials requires declassifying this tag. This tag is important to prevent release of a user's password to someone who might have gained access to their account.

The following two sections explain how data is stored in the system and which tag operations are carried out in a typical workflow.

3.1.2 Files

Files	Secrecy Label	Integrity Label
mint-dir	{}	{$t_{\text{ALL-USERS-DATA}}$, $t_{\text{ALL-BANKS-DATA}}$}
banks-dir	{}	{$t_{\text{ALL-BANKS-DATA}}$}
Bank of America-info	{}	{$t_{\text{BANK-OF-AMERICA}}$}
US Bank-info	{}	{$t_{\text{US-BANK}}$}
users-dir	{$t_{\text{ALL-USERS-DATA}}$}	{$t_{\text{ALL-USERS-DATA}}$}
Bob	{t_{BOB}}	{t_{BOB}}
bob-info	{t_{BOB}}	{t_{BOB}}
bob-password	{t_{AUTH}}	{t_{BOB}}
bob-bankofamerica-creds	{t_{BOB}, $t_{\text{BANK-OF-AMERICA}}$}	{t_{BOB}}
bob-usbank-creds	{t_{BOB}, $t_{\text{US-BANK}}$}	{t_{BOB}}
Dan	{t_{DAN}}	{t_{DAN}}
dan-info	{t_{DAN}}	{t_{DAN}}
dan-password	{t_{AUTH}}	{t_{DAN}}
dan-usbank-creds	{t_{DAN}, $t_{\text{US-BANK}}$}	{t_{DAN}}

Figure 3-2: The file system hierarchy for Mint. In continuation of the example in figure 3-1, Bob added accounts in both banks to his Mint account, and Dan added a US BANK account to his Mint account.Tags are shown under the secrecy label and integrity label columns, and are denoted $t_{<\text{name}>}$. t_{BOB} and t_{DAN} are both *USER-DATA* tags.

The application uses Aeolus files for persistent storage. Figure 3-2 shows the hierarchy of the Mint files and their labels, described below:

mint-dir

This is the root directory for the Mint application. The secrecy label is empty, the integrity label contains the *ALL-USERS-DATA*, *ALL-BANKS-DATA* and *ALL-USERS-AUTH* tags.

banks-dir

Contains a *bank-info* file for each bank that was registered to the application. This directory has an empty secrecy label, and has the *ALL-BANKS-DATA* tag in its integrity labels. Each of the *bank-info* files in this directory has a *BANK-DATA* tag in both its secrecy and integrity labels.

users-dir

Has the *ALL-USER-DATA* tag in both its secrecy and integrity labels. Contains a subdirectory per user, each subdirectory contains a *USER-DATA* tag in its secrecy label, and a *USER-DATA* tag in its integrity label. Each of these subdirectories contains the following files:

user-info

Contains the user tag for that user. This file has the *USER-DATA* tag in both its secrecy and integrity labels.

user-password

Contains the user's password. This file contains the *AUTH* tag in its secrecy label, and the *USER-DATA* tag in its integrity label.

user-bank-info

Contains user credentials for a particular bank. The *users-dir* directory contains one *user-bank-info* file per bank the user added to their account. This file has the *BANK-DATA* and *USER-DATA* tags in its secrecy label, and the *USER-DATA* tag in its integrity label.

3.1.3 Shared Memory Objects

The application uses Aeolus shared memory objects for storing session state for users and bank closures.

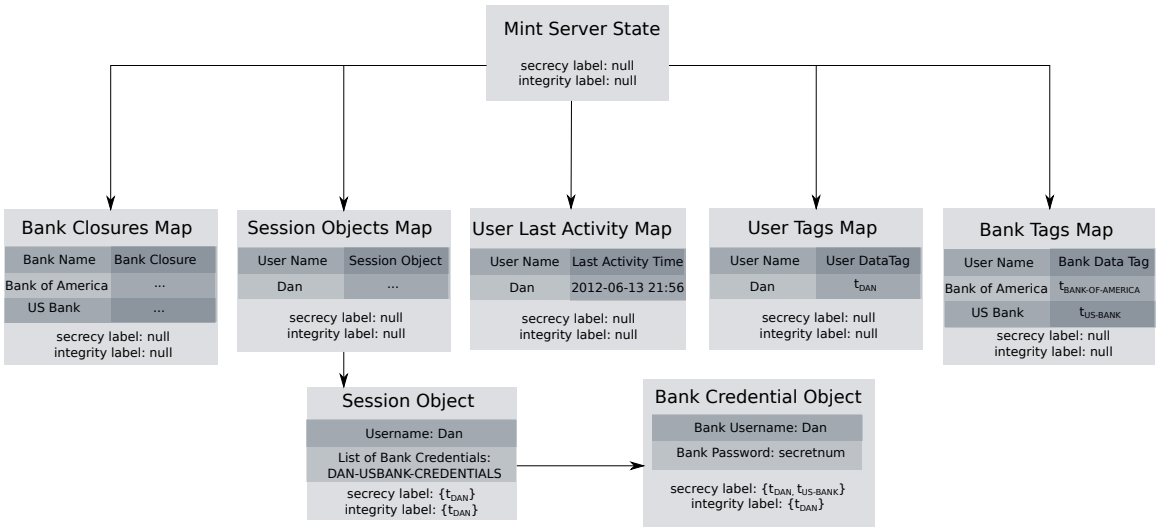

Figure 3-3: Mint shared state objects. This diagram builds on the example presented in Figure 3-2. Bob's sessoin had timed out, and no session object or last activity time is stored for his username.

To allow for quick responsiveness to user requests, the application stores some key information in shared memory. Figure 3-3 diagrams the different shared memory object, their relations what information they store.

A *Mint Server State* object that has null labels is stored as the Aeolus *root* object and contains the following objects:

Bank Closures Map

A bank closure is used to retrieve user information from a bank. The bank closure is authoritative for the tag of the bank it is working for. A bank closure has null labels. This object maps that maps each bank name to a bank closure, and has null labels.

Session Objects Map

This object maps a username to a session object. A *session object* contains session information about a user. It also the list of *bank credential* objects, one for each bank a user registered on their account, as well as the user's PID. A *session object* has the *USER-DATA* tag in both its labels. A *bank credential* object contains the user's username and password for that particular bank.

It has a *BANK-DATA* tag and a *USER-DATA* tag in its secrecy label, and a *USER-DATA* tag in its integrity label.

User Tags Map

The application stores the tag for each user that has an active session, this allows the application server to access session objects with the user tag in their label. User tags are stored as a mapping from username to tag. This object has null labels.

Bank Tags Map

Bank tags are important to store in memory to allow for quick server response time. They are stored as a map from bank name to bank tag. This object has null labels.

User Last Activity Map

The time of a user's last activity is used to identify which users are still active. This information is stored as a mapping from username to last activity time. This object has null labels.

3.2 Implementation

The application server runs on a virtual node, and receives user requests via RPC[2]. Users can sign up, login, logout, register a bank and retrieve statistics. A typical flow of RPC calls is shown in figure d.

At startup, the system registers a number of banks. These are the banks which the users can later add accounts for on their Mint account. The system is now ready to accpet users. Users can register to the Mint service througha *signUp* RPC call:

signUp(username, password)

[2]HTTP requests would be more suitable for this purpose, but the application enforces that the RPC requests have null labels, and hence it is as if they are received from outside the system. The difference then between using RPC or HTTP becomes purely a matter of implementation

Signs up a new user with the username as *username* and password as *password* if *username* is available, associates those credentials with a newly-created user data tag and a newly-created principal ID, then stores all this information on disk under a new user directory, as per the hierarchy described in 3.1.2. The system also stores the newly created tag in the *User Tags Map*. Throws an exception if this *username* is already used.

Users can then log in to be able to modify their account and retreieve their information using the following RPC calls:

login(username, password)

Authenticates a user, and if successful, stores their information in a session object, and updates their last activity time using the shared memory objects described in 3.1.3. The system authenticates the credentials by calling an *authentication closure*, which adds the All User Data tag, verifies the credentials, declassifies and terminates if the *username* and *password* are correct, throws an exception otherwise.

Once a user logs in, they only have to include their *username* in persuing requests to carry out different actions. In each of the following RPC calls, authentication is carried out by checking if the provided *username* has a session that has not timed out yet (there is a last activity time associated with it, and is not older than a certain amount of time, e.g. 15 minutes):

addBank(username, bankName, bankUsername, bankPassword)

Registers the specified bank for the specified username with the given credentials. This request writes a *user-bank-info* file to disk, with the tag of fthe user in it's integrity label, and the tag of the user and that of the bank in it's secrecy label.

downloadTransactions(username)

This action connects to each bank the user has registered for their account, downloads the latest transactions for the user, processes them, and returns

the result. The RPC thread uses the closure of the bank to connect to the bank and declassify the information returned. The thread then adds a user tag to it's secrecy label, and uses a reduced authority call to the public principal to process the information, this ensures that information cannot be leaked while it is being processed [3].

logout(username)

Terminates the user's session, removing their information from shared memory.

removeBank(username, bankName)

Removes the specified bank from a user's account.

3.3 System Security

In the design of the Mint application, we make an assumption that the user is connecting from another Aeolus node. If this was not the case, i.e. the RPC calls are incoming from outside the Aeolus system, sensitive information would be exposed over the network (e.g. the user's Mint credentials), which wouldn't happen for RPCs within the Aeolus system because those are encrypted.

Another problem that comes up due to that assumption is due to the use of a *username* in the implementation of the RPC calls defined above to identify which user invoked those calls. Alice could send a *downloadTransactions("Bob")* request, and because the information is being sent outside the Aeolus system, the information in the reply would have null labels.

Both issues can be resolved by using standard cryptography mechanisms to encrypt and authenticate external communication. For example, encrypting all requests and replies coming in and out of the system and assigning temporary session-ids when users start a session, those session-ids would be used to authenticate future requests in that session.

[3] One can imagine that the information is being processed by a different application in the system, using this technique means our application does not have to trust that application.

[cite authentication schemes here?]

Chapter 4

Auditing and Summarization

The application makes use of the Aeolus audit trails system described in [5] and [1] to monitor system behavior. In particular, we monitor label modification involving a particular user's data tag. Figure x shows a table of which principals declassified Bob's tag, in other words, it shows which user's accessed Bob's information. Figure 4-1 shows a graph of how many user requests caused a label modifications involving a user data tag for every 5-second period since the application launched, in other words, it shows the activity trend of the user, any spikes in the graph mean that the user was abnormaly active in certain periods of time, which could raise suspicion to their account being compromised.

In this section we described how the data for Figure x can be extracted from the event log, as well as the possible uses of summarization for this scenario.

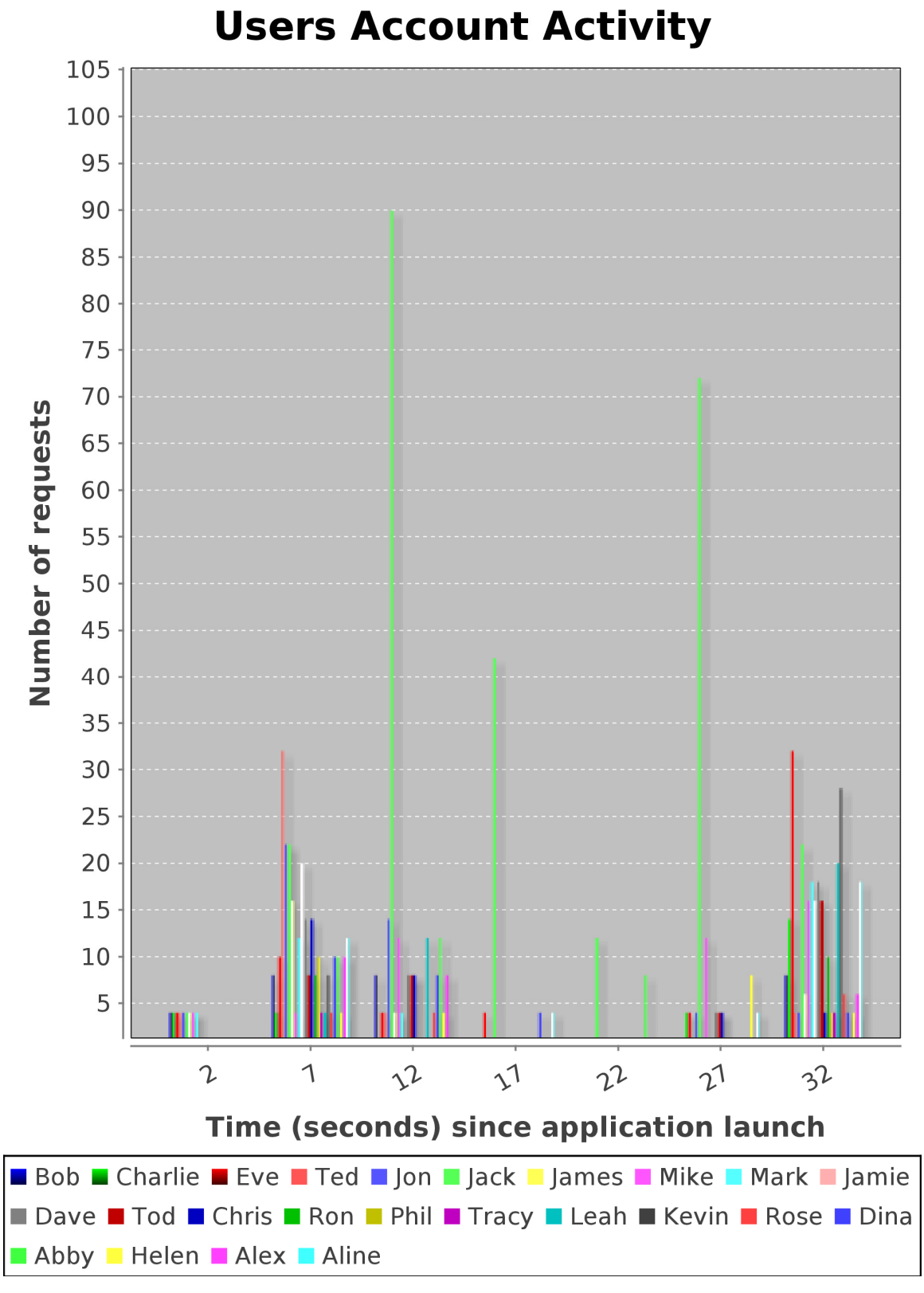

Figure 4-1: This graph shows the activity trend of each user. Notice that the activity for user Jack spikes in the graph at multiple places, this could be the result of an attacker accessing his account and issuing many requests to retreieve all of Jack's financial information in a short amount of time.

4.1 Logging

Aeolus provides a library that allows application developers to add their own application-specific events to the event log. These events are meant to help the developer attach semantic meaning to the event records in the log.

In order to gather the information necessary to produce the graph in figure x, we add application-specific event records to the event log for each different action the user takes.

For example, when a user signs up successfully with our application, we record their username, as well as the tag and principal associated with this user. This done using the following procedure:

```
AeolusLib.createEvent(``mint-signup'', username,
   principal, tag);
```

To obtain a table detailing the username, prinicpal, data tag and other sign-up information of all users, we use the following SQL query based on our application-specific events:[1]:

```
select to_array(app_arg)[0] as pid, to_array(app_arg)[1]
   as username, to_array(app_arg)[2] as tag, timestamp as
   signed_up, event_counter from events where
   op_name='mint-signup'
```

Table 4.1 shows what such a view would look like, with some extra information about time swhen each user signed up. We can now use this table to extract more semantic information about the event log.

For example, we can obtain the information necessary to create the graph in figure y using this query:

```
select e.count, users.username, e.to_timestamp as
   timestamp from (select count(*), tags_modified,
   (to_timestamp(((extract (epoch from
   events.timestamp)/5)::int)*5)) from events
```

[1] *to_array* is a procedure that turns a comma-delimited string into a PSQL array.

pid	username	tag	signed_up	event_counter
6	wissam	[4]	2012-06-13 21:56:36.207	131
9	david	[7]	2012-06-13 21:56:36.207	445
12	dan	[10]	2012-06-13 21:56:37.338	793
13	Bob	[11]	2012-06-13 21:56:37.338	1148
14	Charlie	[12]	2012-06-13 21:56:37.338	1211
15	Eve	[13]	2012-06-13 21:56:37.338	1274
16	Ted	[14]	2012-06-13 21:56:37.338	1337
17	Jon	[15]	2012-06-13 21:56:37.338	1400
18	Jack	[16]	2012-06-13 21:56:37.338	1463
19	James	[17]	2012-06-13 21:56:37.338	1526
20	Mike	[18]	2012-06-13 21:56:37.338	1589
21	Mark	[19]	2012-06-13 21:56:37.338	1652
22	Jamie	[20]	2012-06-13 21:56:38.578	1715
23	Dave	[21]	2012-06-13 21:56:38.578	1778
24	Tod	[22]	2012-06-13 21:56:38.578	1841
25	Chris	[23]	2012-06-13 21:56:38.578	1904
26	Ron	[24]	2012-06-13 21:56:38.578	1967
27	Phil	[25]	2012-06-13 21:56:38.578	2030
28	Tracy	[26]	2012-06-13 21:56:38.578	2093
29	Leah	[27]	2012-06-13 21:56:38.578	2156
30	Kevin	[28]	2012-06-13 21:56:38.578	2219
31	Rose	[29]	2012-06-13 21:56:38.578	2282
32	Dina	[30]	2012-06-13 21:56:38.578	2345
33	Abby	[31]	2012-06-13 21:56:38.578	2408
34	Helen	[32]	2012-06-13 21:56:38.578	2471
35	Alex	[33]	2012-06-13 21:56:38.578	2534
36	Aline	[34]	2012-06-13 21:56:38.578	2597

Users Table

Table 4.1: This is a table detailing the information of users signing up during a simulated run of the Mint application. The *pid* column denotes the user's principal, the *username* column denotes the user's username, the *tag* column denotes the user's data tag, the *signed_up* column denotes the time at which the sign up event was logged to the system, and the *event_counter* column denotes the unique event_counter number of the sign up event.

count	username	timestamp
22	Jack	2012-06-13 21:56:40-04
4	Jack	2012-06-13 21:56:35-04
42	Jack	2012-06-13 21:56:50-04
72	Jack	2012-06-13 21:57:00-04
90	Jack	2012-06-13 21:56:45-04
12	Jack	2012-06-13 21:56:55-04
22	Jack	2012-06-13 21:57:05-04

Account activity for user *Jack*

Table 4.2: This table shows a sample result of the *accesses* view (for simplicity we only show the result for the user Jack). It details the number of user requests causing a label manipulation involving a user data tag for every 5-second period since the application launched. The *count* denotes such number. The *username* column denotes the user who's data tag is in question. The *timestamp* column denotes the start of the 5-second period.

```
tags_modified in (select tag from users) group by
(to_timestamp(((extract (epoch from
events.timestamp)/5)::int)*5)), tags_modified) as e
inner join users on e.tags_modified=users.tag
```

Table 4.2 shows a sample result of this query. This query provides us with a count of how many user requests caused a label modification that involved a user data tag for every 5-second period of time since the application launch, the information necessary to construct the graph in figure y.

4.2 Summarization

Creating summaries could prove to be very useful in auditing the Mint application. For example, if we store the *count* in Figure 4.2 per user session.

In the next chapter, we present a system that allows us to summarize our log and extract the same information from the summary events, and we explain how to reach the same auditing information relying mostly on summary events.

Chapter 5

Summarization Model

The Mint study case highlighted the inherent need for a form of summarization. For example, the query in 4.1 depends on the *users* table, which details the system information related to each user. The *users* table in itself can be thought of as a summary, and queries such as the one in 4.1 depend on that summary for analysis.

In the example mentioned in the previous chapter, we also notice that analyzing the event log can become complicated quickly. For example, query in 4.1, whereas it is fairly readable, it shows that the developer would have to write long SQL queries for different forms of analysis, and that there is no intermediate level of abstraction between what the queries read and what they produce.

The following sections describe how our summarization system allows developers to accomplish three things: use a clear interface to query the event log, reason more easily about events in the past, and improve the performance of their system in terms of runtime and storage costs.

5.1 Querying the Event Log

We provide a query management system that allows developers to name queries and store them. Queries are defined with a placeholder replacing the query arguments, and with a version number. Developers are able to redefine queries, increasing their version number. Developers are later able to run a query by speci-

fying it's name and the query arguments - the system will run the query with that name that has the highest version number. Furthermore, developers can define queries that run over both the event log and the *summaries table*.

5.1.1 Compliance with DIFC

The thread running the query can only retrieve the subset of event records that satisfy the query and that have secrecy and integrity labels that satisfy the DIFC rules defined in section 2.2.2 [1].

We further require that a thread has null secrecy and integrity labels when defining a query. This is important to avoid conflict when the query is run by another thread.

This is a simple but powerful model, which leads to the three main features in our query management system, described in the coming sections.

5.1.2 Usage Simplicity

While auditing the events of an application developers will often have to write similar queries for different purposes. For example, in the Mint application, we might want to identify all writes to disk for each user separately during a particular week. Rather than rewriting many similar queries to obtain these results, we could use one stored query and run it with different arguments.

5.1.3 Versioning

Another important feature that our system provides is the ability to define different versions of queries. This allows for applications that use our system to evolve over time and for developers to change how they audit their applications.

Developers can still use older versions of a particular query to query the event log, this is possible through the use of *Summary Objects* described in section 5.2.

[1] All event records in the event log have a secrecy and an integrity label, as described in [1]

5.1.4 Extensibility - Querying Summaries

Developers are able to query the *summaries table*, described in , in the same way they query the event log. This allows developers to reason about summaries as base events and adds an abstraction layer that gives semantic meaning to the different activities taking place in the system.

5.2 Summaries and Summary Objects

We introduce *Summary Objects* in our system to allow developers to summarize the event log. A Summary Object represents a summary over different event records in the event log.

The notion of a Summary Object relies on the developers ability to define queries through our query management system as described above. To relate Summary Objects to the events they summarize, developers must specify two things when constructing a Summary Object: a query name, and query arguments.

After creating a Summary Object, the developer can store the underlying summary in the *Summaries Table*. The Summaries Table contains the following information for each summary record in it:

Query Version, name and arguments

: The version, name and arguments of the query underlying the Summary Object are used to identify the events summaries by a particular summary record.

Operation name

: This field is specified by the developer, and is used to attach semantic meaning to the summary record.

Secrecy and Integrity Labels

: The secrecy and integrity labels are the same as that of the thread that stores the summary record. This field is necessary to determin who can read this summary record in the future.

Summary info

This field is specified by the developer. It stores the summarized information, for example, the count of how many of the event records represented by this summary record are declassifies of a particular tag.

5.2.1 Summarizing Summaries

It is also possible that a developer might like to summarize summaries they already defined. For example, in the Mint application, after summarizing declassify events for user data tags per user session, we could use those summaries to summarize declassifiy events for user data tags per month.

Our system does not make any strong assumption about the base events while creating a summary, and hence those base events could either be in the event log, or in the Summaries Table itself.

With the use of Summary Objects and our query management system, developers are able to (a) easily query the event log and the summaries table, (b) to use summaries to reason more easily about events in the system, and (c) write more performant code.

5.3 Deleting Records

One of the main purposes of our summarization system is to allow developers to reduce the storage cost that comes with logging all events taking place in an application running on Aeolus.

Given that deleting event records could mean permanent loss of critical information, our system *hides* event records, and summary records, from the developer's access rather than deleting them. This is done by marking the deleted records with a flag, and ensuring that records with that flag set are never returned as a result of running a query.

5.3.1 Archivers

Our system does not fully address the problem of truncating the event log, however it does pave the way for a role of an *archiver*, a program that archives or deletes records marked for deleting based on a certain protocol, for example, if they have been marked for deletion for more than a year.

5.3.2 Undeletable Records

There are some records that are never meant to be deleted. For example, event records in the event log detailing information about the startup of the virtual node, or even sensitive application-specific events.

To address this issue, important Aeolus system events are never marked for deletion, and our system provides an interface for developers to specify a set of events that should never be marked for deletion.

Bibliography

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

[2] Winnie Cheng, Dan R. K. Ports, David Schultz, Victoria Popic, Aaron Blankstein, James Cowling, Dorothy Curtis, Liuba Shrira, and Barbara Liskov. Abstractions for usable information flow control in Aeolus. In *Proceedings of the 2012 USENIX Annual Technical Conference*, Boston, MA, USA, June 2012.

[3] Winnie Wing-Yee Cheng. *Information Flow for Secure Ditributed Applications*. Ph.D., MIT, Cambridge, MA, USA, August 2009. Also as Technical Report MIT-CSAIL-TR-2009-040.

[4] F. Peter McKee. A file system design for the aeolus security platform. Master's thesis, MIT, Cambridge, MA, USA, September 2011.

[5] Victoria Popic. Audit trails in the Aeolus distributed security platform. Master's thesis, MIT, Cambridge, MA, USA, September 2010. Also as Technical Report MIT-CSAIL-TR-2010-048.