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

Distributed Information Flow Control (DIFC) is a technique to confine the flow of information in a distributed system. This technique has an advantage over traditional access control as it separates the ability to read information from the ability to release that information. 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.

Aeolus enforces the information to flow according to the application's security model. However, it would still be possible to for an attacker to tamper with the system and access confidential information, due, for example, to the application having an inadequate security model or bug in it's implementation. Detection of information leakage is hence very important to stop further abuse of application, as well as to identify offenders.

Aeolus allows application developers to track information flow by logging information about the processes running in the application, as well as the operations they carry out. This provides developers with a complete history of the application's activity and information flow through the system since startup. This log is called the *event log*.

The event log is a causal graph: each "event" that took place in the system has predecessors, and hence developers can construct an event graph and analyze it to identify causes of information leakage. However, logging all process operations

for a distributed system can produce a very large event log very quickly. Analyzing and storing such an event log can prove to be both daunting and costly for an application developer, and calls for providing an Aeolus application developer with easier access to information about events of interest.

In this thesis I present a system built on top of the Aeolus security platform that allows users to create *summaries* of the event log. These summaries are application-specific, and left up to the developer to define. What our system provides is a simple framework for the developer to reason about creation, storage and modification of these summaries, as well as archiving of the original event log.

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

Chapter 2 presents an overview of the Aeolsu security system. Chapter 3 presents a sample Aeolus application based on Mint.com.

Chapter 4 describes the summarization and archiving 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.

1.1 Motivation for summarization and archival

The Aeolus platform logs all events that take place in the system. Despite the fact that these events can tell you everything that happened in the system, they do not carry any semantic meaning. Allowing application developers to define a semantic meaning for events, or groups of events, helps in analyzing the history of the system.

Take for example a bank's web administrator who is interested in auditing outbound transfers from customers' web accounts. A straighfoward way to do this is to plot the number of outbound transfers carried through a user's account for each week in the last year. Any spikes in the rate outboud transactions per week could mean that a user's account has been compromised by an attacker. A sample graph is presented in figure x.x.

Producing such information is absolutely possible through the current auditing mechanisms available in Aeolus. However, in order to accomplish such a task, the adminsitrator will not only have to scan all events in the system, but will also, most likely, have to do this frequently to keep his audit records up to date.

Summarization solves these problems by allowing developers to focus on scanning and analyzing only events of interest. Creating and storing *event summaries* also paves the way for archival: now the web administrator in the example above can choose to archive events that more than a year old - if they are *sufficiently summarized* (perhaps footnote a definition of event summaries + sufficiently summarized here).

1.2 Scope

As security platform developers, tackling the problem of summarization and archival raises a number of questions: what usage scenarios are we trying to satisfy? How do we generalize our model to satisfy those scenarios? What guarantees do we provide to our developers? And how do we maintain those guarantees?

I will start by stating that our system, similar to the Aeolus security platform itself, 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 power granted to them through the our system.

Those are the basic assumptions we made while building the system, and most design decisions stem from them. While reading the following chapters, the reader

should expect to gain a better understanding of the definitions of the questions posed above, the design problems they pose, and how we answer them.

Chapter 2

Aeolus System Overview

This chapter presents an overview of the Aeolus security platform, on which my 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 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 authorities 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 help in the control of information flow.

2.1.1 Authority State

Aeolus allows developers to modify the *authority state* using operations such as *delegate* and *actfor*. This allows one principal to delegate authority for a tag it is authoritative for to another principal, or allow another principal to run on it's behalf.

2.1.2 Information Flow Rules

As information flows through the system, it must obey two rules:

$$\begin{aligned} \text{SECRECY}_{source} &\subseteq \text{SECRECY}_{receiver} \\ \text{INTEGRITY}_{source} &\supseteq \text{INTEGRITY}_{receiver} \end{aligned}$$

Aeolus enforces these rules at all times.

2.1.3 Processes

Processes in the system run on behalf of principals, and have security and integrity labels associated with them. In Aeolus, a process 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.

2.1.4 Priviledged Operations

Certain label modifications are *priviledged operations*: removing a tag from the secrecy label and adding a tag to the integrity label. If a process is running on behalf of a principal authoritative for a certain tag, then that process can carry priviledged operations on that tag.

2.1.5 Information Flow in and out of the System

Aeolus process can only write to output outside the system boundaries if it has empty labels. This restriction combined with the information flow rules mentioned above ensures that process can only share or modify information if they are running on behalf of a principal authoritative for tags categorizing that information.

Aeolus virtual nodes can use RPC to communicate with one another, and Aeolus threads can communicate using shared state mechanisms. Aeolus enforces that

both of these communication routes abide by the rules in 2.1.2

2.2 Log Collection

Aeolus nodes are responsible for logging any events that occur locally, and sending them to the *authority server* for analysis and storage.

All calls to Aeolus runtime generate events. For example, calls that modify the authority state, or write to disk, will cause events to be logged to the system. The *audit trails* system descripbed in [5] and [1] gurantees causal relationships between events.

The format of the event log will be described in Chapter 4.

2.2.1 A Semantic Event Log

Each event record

Chapter 3

Mint

Prior to the development of the summarization system, it was important to develop a sense for what is really need from such a system, and what problems can it really 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 it's users with the ability to monitor their bank accounts across different banks from one place. It also allows Mint users to run transaction analysis tools that will access the information of certain transactions for each bank for the user, and generate a result.

For example, Bob could sign up on www.mint.com and add his Bank of America account and his US Bank account for monitoring. Bob can then request a graph that presents his expenditures on food in the last week.

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

3.1.1 Authority State Model

In this section we identify the prinicpals running in the system, the tags associated with different data, and the relationships between them.

A very 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, 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.

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

This prinicpal is not authoritative for any tags [2].

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 tags

Each of these tags is associated with data carrying a particular user's information.

Bank tags

Each of these tags is associated with data carrying a particular bank's information.

[1] Neccessary by good design. The application could run with just one principal
[2] It is used when no privileged operations are necessary to ensure that no information leaks

All-Users-Data tag

This tag is a super tag for all the user tags.

All-Banks-Data tag

This tag is a super tag for all the bank tags.

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

3.1.2 Files and Shared Memory Objects

The application uses Aeolus' file system API, for persistent storage, and uses Aeolus shared memory objects for storing session state for users and bank closures. A complete description of the Aeolus file system API is presented by McKee [4].

Files

The application stores it's information under a root directory called *mint-dir*. The are two direct subdirectories of $mint - dir$: *banks-dir* and *users-dir*.

$banks - dir$ contains a file for each bank the application supports, the filename is derived from the bank name. $users - dir$ contains a subdirectory per user, each of these subdirectories contains two files for the user called *user-info* and *user-password* as well as another file per bank the user has added, called *user-bank-info*.

$user - info$ contains the user tag for that user. $user - password$ contains the user's password. $user - bank_name - info$ contains the credentials for the user to authenticate with a specific bank. Figure x shows a sample directory tree, and table x shows the file labels.

Shared Memory

To allow for quick responsiveness to user requests, the application stores some key information in shared memory:

Bank closures

Bank closures are objects that allow the application to connect to the bank and retrieve user information. Storing them in memory allows for a quicker response if a user's request requires connecting to a bank. It is not very costly to store them in memory because the number of banks supported by the server is limited.

Session Objects

Session objects contain session information about a user. They are stored in a map that maps a session-id to a session object. The session object contains the list of the bank credentials a user has registered, as well as the user's PID.

User tags

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.

Bank tags

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.

User idle time

User idle time identifies users that are still active. This information is stored as a mapping form username to last activity time.

3.2 Implementation

The application server runs on a virtual node, and receives user requests via RPC[3]. Users can sign up, login, logout, register a bank, retreieve statistics as well as *attack other users*.

[3]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

User requests spawn a thread at the server that runs as the Mint principal. The thread first loads the server state from memory, then the thread authenticates the user, carries out the requested user action and returns. This requires label manipulation, reading from and writing to disk, and accessing shared memory storage. The following section describes the authentication mechanism as well as each user action in more detail.

3.2.1 Authentication and User Actions

Users sign up to the service by sending a *signUp* RPC to the server. The server then creates a PID and a tag for that user, and stores this information, along with user credentials, on disk as described in seciton 3.1.2.

A user of the Mint application starts a session by logging in. The system authenticates the credentials by calling an authentication closure, which adds the All User Data tag, verifies the credentials, declassifies and returns the result of the verification.

Once a user logs in, they use a session id to authenticate their future requests. To authenticate session ids, the system accesses the shared memory objects described above to verify that the user's session has not timed out.

If authentication is successful, the thread carries on and executes the user's request. We now examine the different user actions the application supports.

signUp(username, password)

Signs up a new user with the username as *username* and password as *password* if *username* is available, and associates those credentials with a tag and a principal ID. The label modifications happen as described above.

login(username, password)

Authenticates a user, and if successful, places their information in shared memory. Label manipulation happens as described above. Following requests from this user only need to include the username.

logout(username)

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

addBank(username, bankName, bankCredentials)

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 the user in it's integrity label, and the tag of the user and that of the bank in it's secrecy label.

removeBank(username, bankName)

Removes the specified bank from a user's account.

downloadTransactions(username)

This action connects to each bank the user has registered on their account, downloads the latest transactions, processes them and returns the result. The RPC thread uses a closure call that runs with the PID 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 public principal to process the information, this ensures that information cannot be leaked while it is being processed[4]. After that the thread declassifies and returns. Figure x shows a code sample.

attack(attacker, victim)

Attacks the uer specified by *victim*. This request aims to simulate a bug in the appliction that mistakingly delegates the victim's data tag to the attacker's principal. This user action is useful for testing and auditing purposes, as explained later in this chapter.

Figures: closure code sample, authority state structure, file system tree + labels, auditing + summarization

[4]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. Only the code for the Bank closure needs to be trusted.

3.3 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 3-1 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.

3.3.1 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);
```

In order to obtain a table detailing the username, prinicpal, data tag and other sign-up information of all users, we create a view based on our application-specific events using the following SQL query[5]:

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

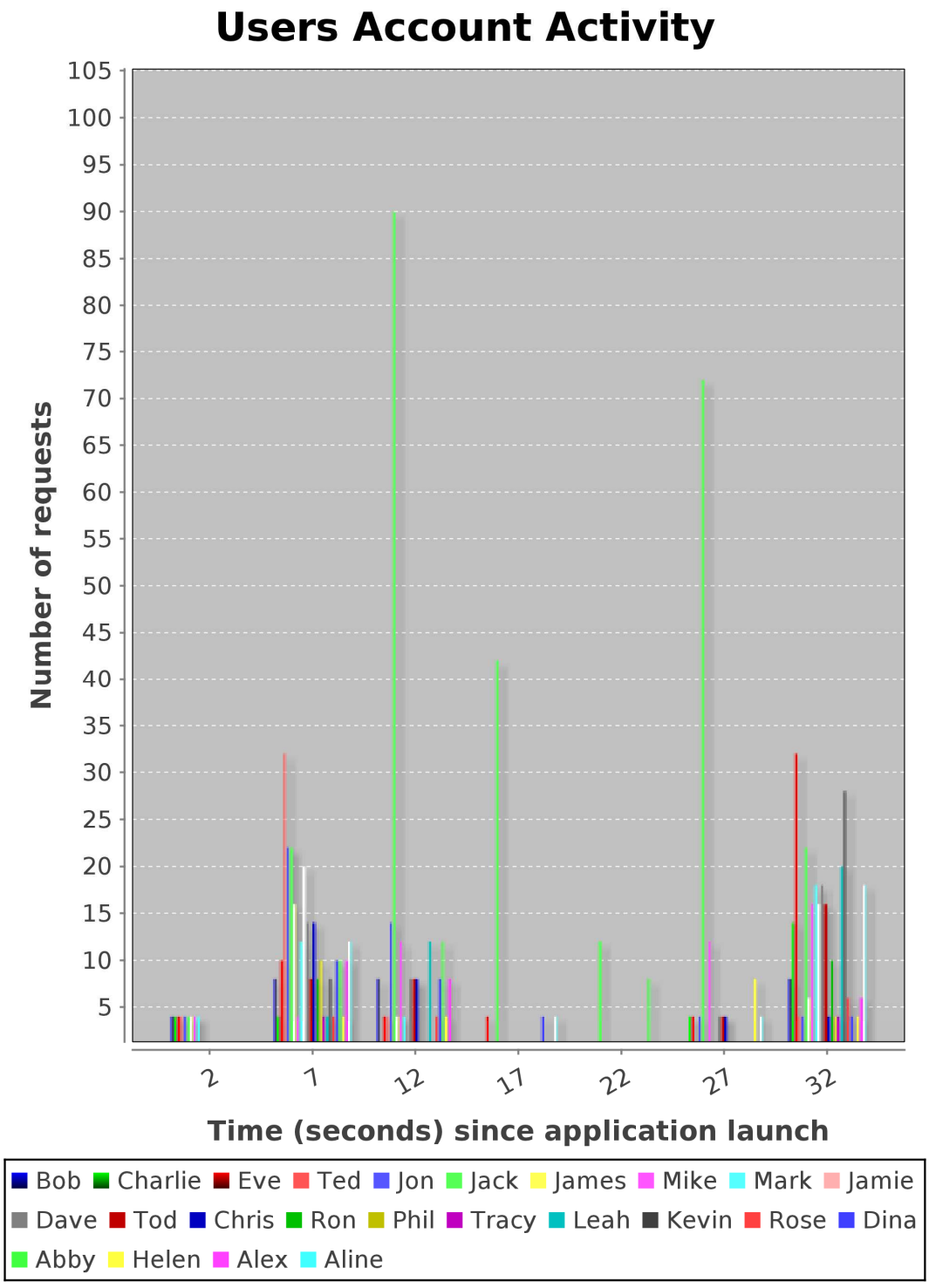

Figure 3-1: This graph shows the activity trend of each user. Notice that the activity for user Jack spikes in the graph at multiple places, this could be the result of an attacker accessing his account and issuing many requests to retreieve all of Jack's financial information in a short amount of time.

```
create view users as 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 3.1 shows what such a view would look like, with some extra information about time of when 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:

```
create view accesses as 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
   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 3.2 shows a sample result of this query. This query provides us with a count of how many user requests 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.

3.3.2 Study Case for Summarization

One of the main purposes of building this application was to evaluate how developers would go about auditing the events in the system, and what can be done to make that task easier. In the next chapter we describe the issues we identified and how our summarization system resolves them.

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 3.1: 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 3.2: 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.

Chapter 4

Summarization Model

The Mint study case highlighted the inherent need for a form of summarization. For example, the query in 3.3.1 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 3.3.1 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 3.3.1, 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.

4.1 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 specifying 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*.

4.1.1 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 2.1.2 [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.

4.1.2 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.

4.1.3 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 4.2.

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

4.1.4 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.

4.2 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.

4.2.1 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.

4.3 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.

4.3.1 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.

4.3.2 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.