1-1-2015

# Evaluation Of An Architectural-Level Approach For Finding Security Vulnerabilities

Mohammad Anamul Haque
*Wayne State University,*

# EVALUATION OF AN ARCHITECTURAL-LEVEL APPROACH FOR FINDING SECURITY VULNERABILITIES

by

**MOHAMMAD ANAMUL HAQUE**

**THESIS**

Submitted to the Graduate School

of Wayne State University,

Detroit, Michigan

in partial fulfillment of the requirements

for the degree of

**MASTER OF SCIENCE**

2015

MAJOR: COMPUTER SCIENCE

Approved By:

______________________________

Advisor Date

# DEDICATION

*To my Family and Friends*

# ACKNOWLEDGEMENTS

This thesis is the summary of my work at SEVERE (SoftwarE Visualization and Evolution REsearch) lab of Wayne State University, Detroit, MI. I would like to thank and express my gratitude to all who supported me during my study.

First, I would like to thank and express my gratitude to my advisor Dr. Marwan Abi-Antoun who offered me a great opportunity to work under his supervision when I needed it most. Without his constant guidance and feedback, it would not have been possible to accomplish this goal. I thank him for his patience and positive criticism that helped me to understand the fundamentals of the research and shape my thinking of new ideas. His constant encouragement inspired me to complete this work with as much perfection as I can.

I am grateful to all the members of the thesis committee who happily accepted my request putting aside their busy schedules. I am thankful to Dr. Vaclav Rajlich and Dr. Alexander Kotov for their valuable comments, suggestions and feedback. Disclaimer: Any remaining faults in this thesis are mine alone.

I would also like to thank all of my friends at Wayne State University whom I met while pursuing my study. I thank to all of the members of the SEVERE group. In particular, I thank Sumukhi Chandrashekar who read my thesis and suggested many changes; Mohammad Ebrahim Khalaj, with whom I discussed new ideas; Yibin Wang, for reading my drafts and providing me with important feedback. I would like to extend my gratitude to my friends from other labs and departments: Faria Mahnaz, Rajiur Rahman Raju, Abdullah-Ibn Mafiz and Tarique Hasan Khan who always were very supportive.

I would like to extend my gratitude to the members of the Department of Computer Science for their outstanding cooperation, providing me with the financial means of my study. I would also like to thank all the professors whose courses I have taken;

I was very lucky to learn from their teaching. I would also like to thank other faculty members who supported me.

Last but not the least, I especially thank my mother, my father and my sisters without whose inspiration and moral support I could have never reached to this accomplishment. I also want to extend my gratitude to all of my friends and other relatives who always helped me with mental support during my study.

# TABLE OF CONTENTS

# LIST OF TABLES

## LIST OF FIGURES

# CHAPTER 1: INTRODUCTION

This chapter introduces the problem addressed in this thesis, the solution to the problem, the thesis statement, and the subject systems that are used in the evaluation.

## 1.1 Problem Description

A wide number of software tools are available that look for coding bugs such as a hard-coded password in a system .[1] However, tools that look for architectural flaws (e.g., storing unencrypted password) of the system are less mature. Such a tool requires a thorough evaluation. The underlying analysis of the tool can be dynamic or it can be static. Dynamic analysis runs on specific inputs whereas static analysis exploits all possible paths in the program. Although static analysis explores all possible paths, the tool may report more than the actual numbers of flaws in the system (false positives). Moreover, an analysis tool also needs to be scalable to large subject systems. Thus, such a tool requires a rigorous evaluation on large subject systems of a variety of domains, such as Android and the Web.

### 1.1.1 Coding Bugs and Architectural Flaws

Architectural flaws account for 50% of the security vulnerabilities .[2] Finding an architectural flaw requires the tool to analyze a high-level representation of the system rather than the code level. On the other hand, to find a coding bug, it only requires to analyze one method of one class at a time and does not require to track the transitive dataflow communication between objects. For instance, to find a hard-coded IP address in the code, the analysis needs to scan through all the hard-coded strings and match with the signature of the IP address. However, it is not sufficient to explore a code statement that writes confidential information into a file and define the statement as an architectural flaw. The security properties of the dataflow object (e.g., the information) and the file object are also important to consider. An architect assigns security properties to the source, the destination and the dataflow objects to query the high-level representation of the system in order to find the flaw.

### 1.1.2 Security Vulnerability and Architectural Flaw

By definition, in computer security, a vulnerability is referred to a weakness that allows an attacker to compromise the system's security infrastructure [1]. The exploitation of a security vulnerability requires to pass through more than one process of the system, at each of which the exploitation can be stopped.[3] Some architectural flaws can be significant enough to potentially compromise the system security. One of the solutions for finding such an architectural flaw is Architectural Risk Analysis (ARA). ARA generates a forest-level view of the system that contains runtime components of a system and connections between them .[4] The architects use the view to assign security properties and check constraints. However, ARA is manual and can miss some potential components and communication edges. Another proposed solution is to extract a high-level diagram of the system called an Object Graph (OGraph). An OGraph represents the abstract runtime structure of the system as a graph in which a node is defined as an abstract object and an edge represents the communication between two objects. Architects find the vulnerabilities by assigning security properties to the components (objects and edges) and writing security constraints .[5]

## 1.2 Analysis Tools

A few analysis tools that find security vulnerabilities are Coverity ,[6] HP Fortify ,[7] Cigital Secure Assist [8] etc. In this thesis, I evaluate a static analysis tool called Scoria. Scoria is a Java-based analysis tool that finds the security vulnerabilities that are architectural flaws of a system. Scoria generates a global sound hierarchical object

[1] http://en.wikipedia.org/wiki/Vulnerability_(computing)

graph (OGraph). A sound abstract OGraph represents all the objects and dataflow edges that may exist at runtime. A system architect finds the vulnerabilities by querying the OGraph. Scoria was evaluated on a number of test cases. Some of these test cases are Android-based small applications adopted from the DroidBench [9] benchmark and the rest of the test cases are devised. DroidBench is a micro-benchmark that contains small test cases.

## 1.3 Thesis Statement

Scoria finds security vulnerabilities that are architectural flaws in medium to large subject systems from different application domains and does not generate many false positives.

### 1.3.1 Hypotheses

To support the thesis statement, I propose two hypotheses:

**H1.** Scoria finds security vulnerabilities that are architectural flaws in medium to large subject systems from different application domains.

**H2.** Scoria does not generate many false positives.

## 1.4 Systems to Evaluate

I select a number of criteria to choose the subject systems: the type of the system (in order to support the hypothesis H1), the code size (KLOC), and the object-oriented properties of the system. Scoria was evaluated on a system named UPMA (Universal Password Manager for Android) which is around 4 KLOC.[10] I choose three subject systems two of which are around 6 KLOC and one is around 25 KLOC. I choose two Android applications: Muspy and Ermete SMS and one web application (WebGoat) (Table  1.1). While WebGoat is a deliberately vulnerable application,

**Table 1.1:** Statistics of the subject systems

| Subject | App. Type | KLOC (approx.) | Packages | Classes | Interfaces | Methods |
|---|---|---|---|---|---|---|
| Muspy | Android | 6 | 14 | 115 | 1 | 376 |
| Ermete SMS | Android | 6 | 9 | 48 | 0 | 198 |
| WebGoat | Web | 25 | 22 | 191 | 2 | 1509 |

the other two systems are also potentially vulnerable applications. Both of these Android systems require user authentication, maintain user profiles and save confidential information into local databases. All of the systems were implemented using object-oriented programming techniques. Table 1.1 shows some metrics about the subject systems and how they are consistent with the selection criteria.

## 1.5 Results of the Subject Systems and Contribution

I evaluate Scoria over three subject systems by annotating the code, typechecking the annotations, extracting the OGraphs, refining the annotations and writing constraints. The results show that there are interesting security vulnerabilities in Muspy and Ermete SMS that point to the common architectural flaws in Android development. The results also show that Scoria detects security vulnerabilities in larger systems. I discuss the precision of the analysis results with respect to the generated false positives.

## 1.6 Outline

The rest of the thesis is organized as follows: Chapter 2 gives some background on available approaches and the underlying Scoria approach. Chapter 3 explains the detailed subject system Muspy, the annotation process, the extraction of the OGraph and the constraints writing. Chapter 4 describes Ermete SMS, the annotation process, the extraction of the OGraph and the constraints. Chapter 5 explains the architecture of WebGoat, the annotation process, the OGraph extraction, the refinement of the annotations and the constraints. Chapter 6 discusses and validates the thesis statement by evaluating the results. Chapter 7 discusses related work that evaluated software analysis tools. Chapter 8 discusses the contribution of the thesis along with possible future work.

# CHAPTER 2: BACKGROUND

This chapter gives some background on the underlying analysis type, i.e., the static analysis, some advantages of static analysis over dynamic analysis, and the process of applying the Scoria approach.

## 2.1 Static Analysis

To resolve potential security vulnerabilities, two types of analysis have been developed; static analysis and dynamic analysis. A static analysis is performed without actually executing the programs. Although dynamic analysis can be successful by precisely identifying the system behavior on specific inputs, there are certain advantages of static analysis over dynamic analysis.

### 2.1.1 Static vs Dynamic Analysis

A dynamic analysis is dependent on the program execution and deals with the runtime values. It may also require loading of special libraries and recompilation of the code. A dynamic analysis can detect the dynamic dependencies of a program. It can extract a dynamic hierarchical object graph where a child object can be found at the descendant level of its parent .[10] A dynamic object graph may have thousands of nodes; however, it requires extensive graph summarization to obtain an abstract graph.

Moreover, dynamic analysis cannot guarantee the identification of the vulnerabilities since all the input scenarios cannot be tested. On the contrary, a static analysis is sound in the sense that it considers all the program execution paths and thus, it is independent of the input.

Additionally, the exploitation of security vulnerabilities is not necessarily causes by a malicious input. An attacker can exploit vulnerability by providing a valid input. This is because an architectural flaw may exist due to a dataflow from a source object to an untrusted destination which may not be detected by dynamic analysis. For example, consider a web application that uses a hidden input field that stores monetary values temporarily. Since the input field takes numeric values as input, tampering the field will not be recognized by the dynamic analysis. A static analysis, on the other hand, can find this vulnerability by analyzing the security properties of the input object and the dataflow object that represents the monetary values.

### 2.1.2 Hierarchical Object Graph

The static analysis tool, Scoria extracts a hierarchical abstract object graph (OGraph) based on the annotations of the code and the abstract runtime structure of the system. The elements of an OGraph are OObjects (abstract objects), ODomains (conceptual group of objects) and OEdges, the dataflow communication edges between the objects. An ODomain can contain multiple OObjects and an OObject in turn can have zero or more domains. An OGraph is sound in the aspect that it shows all possible communication edges between two runtime objects. Soundness also guarantees that one particular runtime object cannot have two different representatives in the OGraph.

### 2.1.3 Extracting information from the Object Graph

Using the OGraph, a security architect can visualize the domains (conceptual grouping of objects), abstract objects, hierarchies of the objects and dataflow communication edges.[10] The architect can reason about the transitive dataflow information from a source to a destination through object hierarchy and object reachability and can reason about a security vulnerability issue. There may also be a direct dataflow edge between a source and a destination object for which an architect assigns security properties to the connecting objects and edges to check if confidential information flows to the untrusted destination.

## 2.2 The Scoria Approach

In this thesis I evaluate if Scoria can find security vulnerabilities.

The Scoria approach consists of a number of steps. These steps are annotate the code, extracting the *SecGraph*, refine the annotations, write constraints and trace suspicious edges. In the following sections I will discuss each one in turn.

### 2.2.1 Annotate the code

The first step of the Scoria approach is to annotate the source code. The annotations reflect the design intent of the architect. The architect annotates each class, variable, field and method return type. After adding annotations, the architect typechecks the annotations using a typechecker[1]. The typecheker shows a number of warnings and their priorities. The architect needs to address the warnings by the order of the priorities. A portion of the annotated code of the CourSys application is shown in Fig. 2.1. CourSys is a samll java application that manages students, courses and registration of courses of a schooling system.

### 2.2.2 Extracting the OGraph

Scoria extracts the OGraph from the annotated code and creates the *SecGraph*. *SecGraph* is a wrapper of the OGraph that allows the architects to assign security properties to the objects and the edges in order to query the graph. Depending on the size of the system, the size of the OGraph may be very large. For a small subject system (e.g., an Android application in DroidBench benchmark), the generated

[1] The typechecker is currently integrated with eclipse, so all the work has to be done in IDE

```
class Main<"user", "logic", "data","owned"> {
       Data<data <owned>> objData = null;
    Logic<logic <data, owned>> objLogic = null;
    Client<user<logic, owned>> objClient = null;
       public void run(){
       objData = new Data(studentFile, courseFile);
       objLogic = new Logic(objData);
       ILogic<logic> logic = objLogic;
       objClient = new Client(logic);
       objClient.execute();
    }
       public static void main(String<lent[shared]> args[]) {
       String<shared> arg0 = args[0];
              String<shared> arg1 = args[1];
       try {
              Main<lent> system = new Main(arg0, arg1);
           system.run();
       }
       catch (Exception<lent> e) {
           System.out.println("Unexpected exception");
       }
    }
}
```

**Figure 2.1:** Annotated main class of CourSys application

OGraph is usually small and easily interpretable. For large subject systems, however, it may be difficult to follow the dataflow communication edges especially in the case of object hierarchy and indirect communication. Hence, I extract and look at the Ownership Object Graph (OOG) of the annotated system. OOG is the hierarchical representation of the abstract runtime structure that provides a collapsed view of the system in which objects of a domain are nested into the parent object that is the owner of the domain. OGraph is the internal representation of the OOG. As an example, the OGraph and the OOG of the CourSys application are shown in Fig. 2.2 and Fig. 2.3 respectively. In this example, `objLogic:Logic` has `log:Logging` and `lock:RWLock` objects in its owning domain (Fig. 2.2), however, in the OOG, these two objects are collapsed in the `objLogic:Logic` (Fig. 2.3).

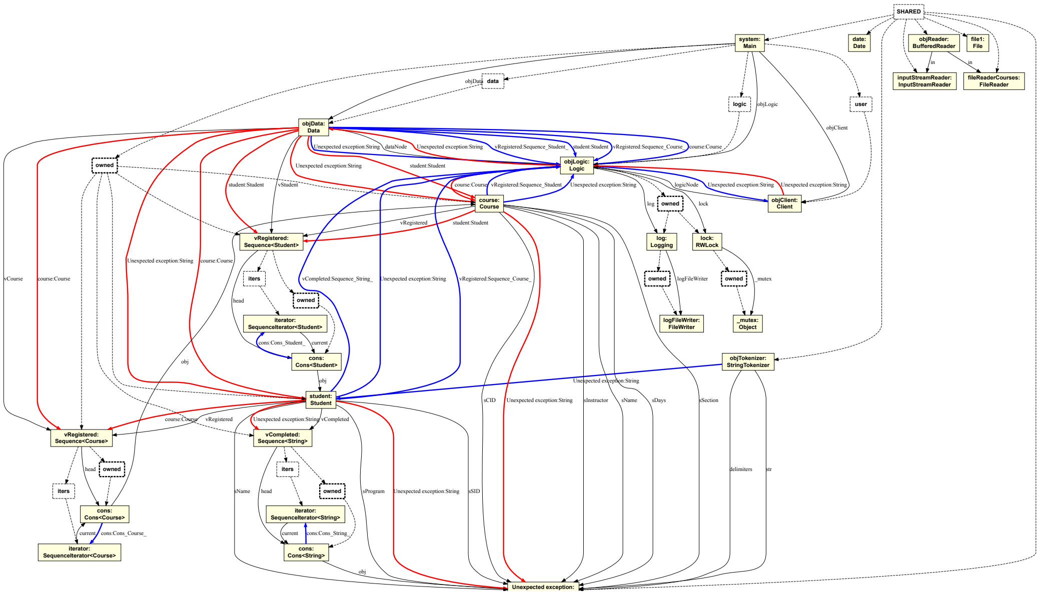

SHARED
system: Main
date: Date
objReader: BufferedReader
file1: File
inputStreamReader: InputStreamReader
fileReaderCourses: FileReader
data
logic
user
objData: Data
objLogic: Logic
objClient: Client
owned
course: Course
vRegistered: Sequence<Student>
iters
iterator: SequenceIterator<Student>
cons: Cons<Student>
student: Student
log: Logging
lock: RWLock
logFileWriter: FileWriter
_mutex: Object
objTokenizer: StringTokenizer
vRegistered: Sequence<Course>
cons: Cons<Course>
iterator: SequenceIterator<Course>
vCompleted: Sequence<String>
iterator: SequenceIterator<String>
cons: Cons<String>
Unexpected exception:

**Figure 2.2:** OGraph of the Coursys application

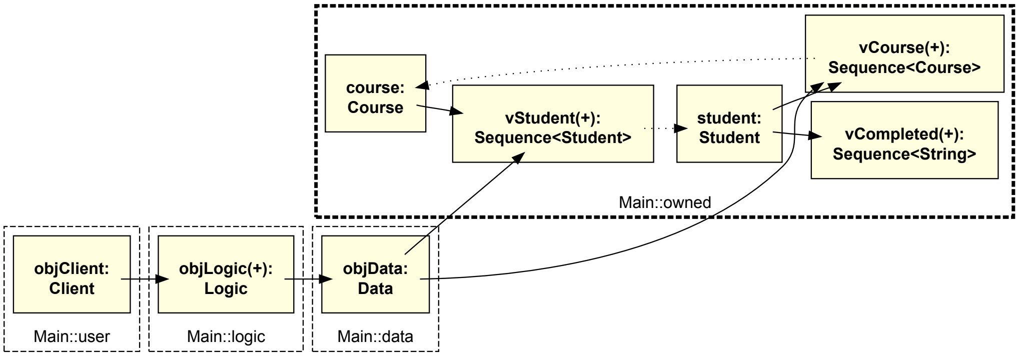


**Figure 2.3:** OOG of the Coursys application (nested boxes)

### 2.2.3 Refine the Annotations

An architect may need to refine the annotations for two types of scenarios: a) keep the annotations consistent with the design intent and code and b) prevent objects from getting merged by the analysis. The architect can modify the annotations to keep the analysis consistent with the design intent and the code. For instance, object `o:O` may be initially annotated in domain `U`. However, it can be found that `o:O` is an important object that contain confidential information or a wrapper of an important object. As a result, the architect modifies the annotation to domain `D` that may usually contain the important objects. The architect can check the OOG, refine the annotations and can re-extract the OOG and iterate the same process until the OOG reflects the design intent. Second, some of the vulnerabilities may not be found due to excessive merging (abstraction), and it is required to separate a target instance of a particular type and annotate it in a distinct domain to prevent merging. Thus, the separated object with relevant edges will appear in the OGraph and will be available for the architect to assign security properties while writing constraints.

### 2.2.4 Write Constraints

Scoria has five different types of queries that help the architect to minimize the effort of writing constraints to find vulnerabilities; Object Provenance, Object Transitivity, Object Hierarchy, Object Reachability and Indirect Communication. These features are depicted in Fig. 2.4. The images are repeated here to make this document more self-contained .[10]

**Object Provenance** is a query that returns a set of dataflow edges if same object flows from two different set of sources to two different set of destinations. For instance, if object `o:O` flows from `a:A` to `b:B` and it also flows from `c:C` to `d:D`, the analysis will report two suspicious dataflow edges regarding the same flow object `o:O` (Fig. 2.4 (a)).

**Object Transitivity** is the communication that reflects the transitive flow of a particular object in a communication path between two objects where the destination object may be untrusted. For example, `o:O` can flow transitively from `a:A` to `c:C` in which multiple objects can be found in the communication path between `a:A` and `c:C` (Fig. 2.4 (b)).

**Object Hierarchy** is defined as the descendants and ancestors of an object `o:O` where confidential information can be found in a descendant of `o:O`. However, instead of passing the descendant object, `o:O` is passed from a source to an untrusted destination.

Thus, the destination has an access to the information (Fig. 2.4 (c)).

**Object Reachability** is defined as a path that exists between a flow object of one path to a predefined object. For instance, flow object `c:C` of the communicating objects `a:A` and `b:B` is reachable to the object `o:O`, if there is a path exists between `c:C` to `o:O` (Fig. 2.4 (d)).

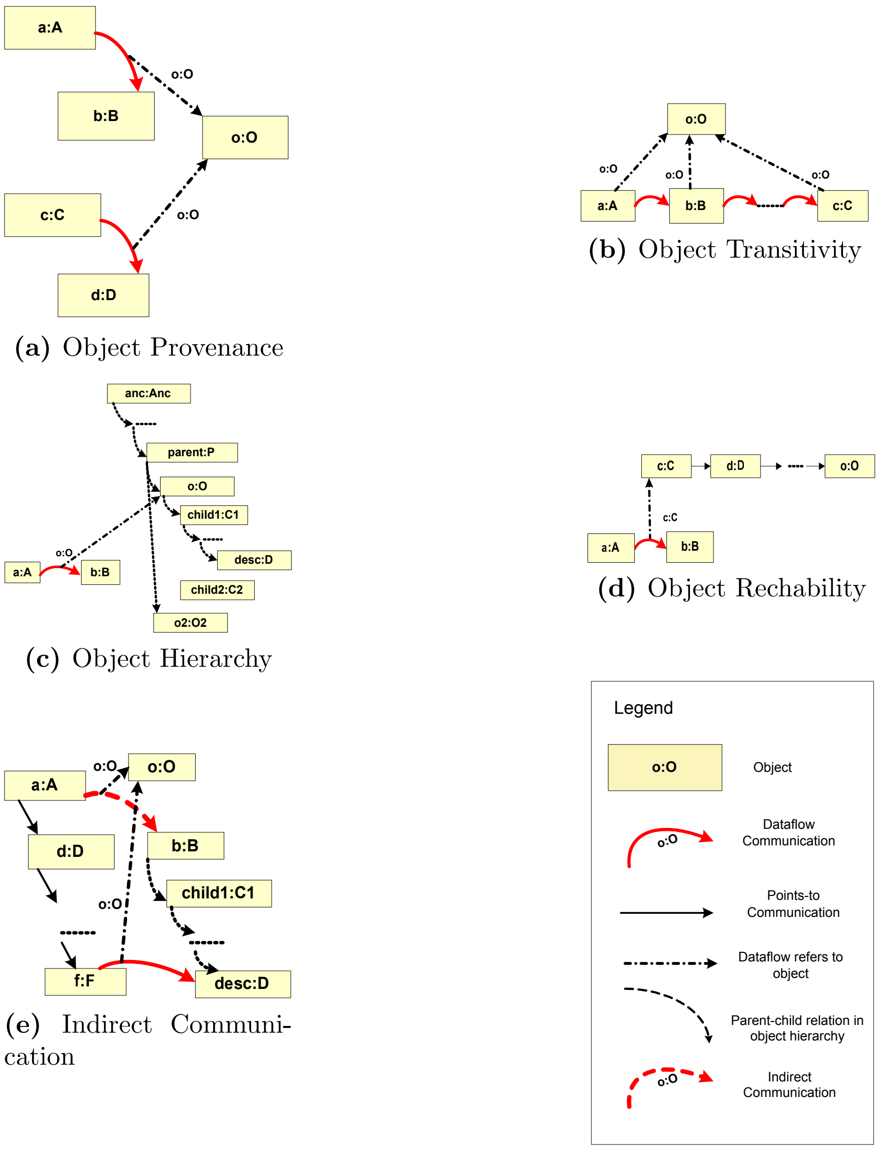
**Indirect Communication** refers to an indirect path that exists from a source `a:A` to a destination `b:B` if a dataflow or creation edge exists from a descendant of `a:A` to a descendant of `b:B` (Fig. 2.4 (e)).

### 2.2.5 Trace Suspicious Edges

The architect implements the constraints as test cases using the features of Scoria. Running the test cases shows a number of suspicious communicating edges in the console window (if any) that includes the corresponding lines of code of method invocations or field writes. Thus, the architect can trace the lines of code responsible for the edges. The analysis may warn about some irrelevant suspicious edges (false positives) alongside the true edges.

### 2.2.6 Compute AF-Index

The security architect computes the Architectural Flaw Index (AF-index) of each constraint that determines if a vulnerability is an architectural flaw or a coding bug. The AF-index classifies security vulnerabilities along a continuum ranging from coding bugs to architectural flaws.[11] The AF-index attempts to classify each vulnerability as an architectural flaw. In this thesis, the AF-index ranges from 1 to 10. Each Scoria feature has a weight and the AF-index of a vulnerability is the weighted sum of the values of all the features used in a constraint. Since object provenance and indirect communication are difficult to reason about, a higher value 3 is assigned to these



**(a)** Object Provenance

**(b)** Object Transitivity

**(c)** Object Hierarchy

**(d)** Object Rechability

**(e)** Indirect Communication

**Figure 2.4:** Scoria Features

features and a value of 2 is assigned to the object transitivity and object hierarchy. Security properties are assigned a value of 1. The higher the weighted sum of the used features in a constraint, the more likely a vulnerability is an architectural flaw.

The following chapters discuss the subject systems, the detailed process of applying the Scoria approach over the systems and the corresponding results of the analysis.

# CHAPTER 3: MUSPY

The chapter gives an overview of the first subject system, its functionality, its architecture and constraints that find vulnerabilities in the system.

## 3.1 Muspy Overview

Muspy is a free Android application that notifies the users who expressed interest when their favorite artists release new albums[1]. Muspy targets music lovers who do not want to miss any album from their favorite artists or learn about new albums after many days of their releases. Moreover, it saves the users' from having to constantly check the official websites of the artists.

Following are the features of Muspy:

1. Sign In and Sign Out: A user can create a new account by providing an email id and a password. Email activation is required after which the user is allowed to sign in to the system. The user can log out from the system by clicking the sign out button and there is no internal caching functionality of the login credentials.

2. Reset Password: In case of forgetting the password, a user can reset her password by providing the email address and Muspy sends a password reset link to the email address.

3. Add and Remove Artists: A user can follow artists by searching for the artists and adding them to the account and can unfollow them by clicking the remove button. The user can also share information of each album release to social media. Moreover, Muspy allows the users to achieve more information regarding

[1] https://muspy.com/

each album release through youtube, Last.FM, Wikipedia, Spotify, Amazon and Discogs.

4. Import Artists: A user can add all or a portion of top artists (25%, 50%, 75% and 100%) from their Last.FM profiles by inserting their Last.FM profiles' user names.

5. Notifications: A user gets an automatic email notification for each new album release. Each new released album is also added to the list of albums of each favorite artist of a user.

## 3.2 System Architecture

Muspy is 6 KLOC, so it is a mid-size Android system. I study the architecture of the system reading the available documentation, browsing the code and by drawing the dataflow diagram (DFD) by hand. I measure some code statistics (shown in Table 1.1) of Muspy by running a tool named Metrics[2].

### 3.2.1 DFD

I inspect the code to understand the architecture and manually draw (since there is no documented DFD) the data flow diagram (DFD) of the system before starting the Scoria analysis for Muspy. I draw two levels of DFDs (DFD Level-1 and DFD Level-2) and manually check the conformance of the OOG with the DFD Level-2 to see if the extracted OOG matches the design intent, and is comparable to the diagrams I drew.

[2] http://metrics2.sourceforge.net/

### DFD Level-1

Muspy has five complex processes, two data storages and the external user. The DFD shows the processes (Fig. 3.1), the data storage, the external user and the corresponding communication edges between the processes.

1. Artist Activity: A complex activity that contains multiple single processes each of which represents an activity relevant to an artist. An external interactor can communicates with the complex process, “Service Manager” via each individual process (Fig. 3.1).

2. Release Activity: “Release activity” shows detail information for each album release of an artist. It communicates with the “Service Manager” and the “external user” (Fig. 3.1).

3. User Account Activity: This complex process consists of all the account related activities; the activities for signin, signup, reset password and user account settings. User account activity communicates with the complex processes: the “Service Manager” and the “Credential Manager”.

4. Service Manager: “Service Manager” consists of a client process, a utility process and one user settings process. It provides HTTP services to other processes to perform the GET and the POST operations. It is also responsible for communicating with the data storage (Fig. 3.1).

5. Credential Manager: This process encrypts and decrypts a user’s password before storing into and retrieving from the data storage. (Fig. 3.1).

6. Unprotected Data: The unprotected data storage stores the insecure information related to artists’ data. Data are pulled and pushed from this storage through HTTP protocols.

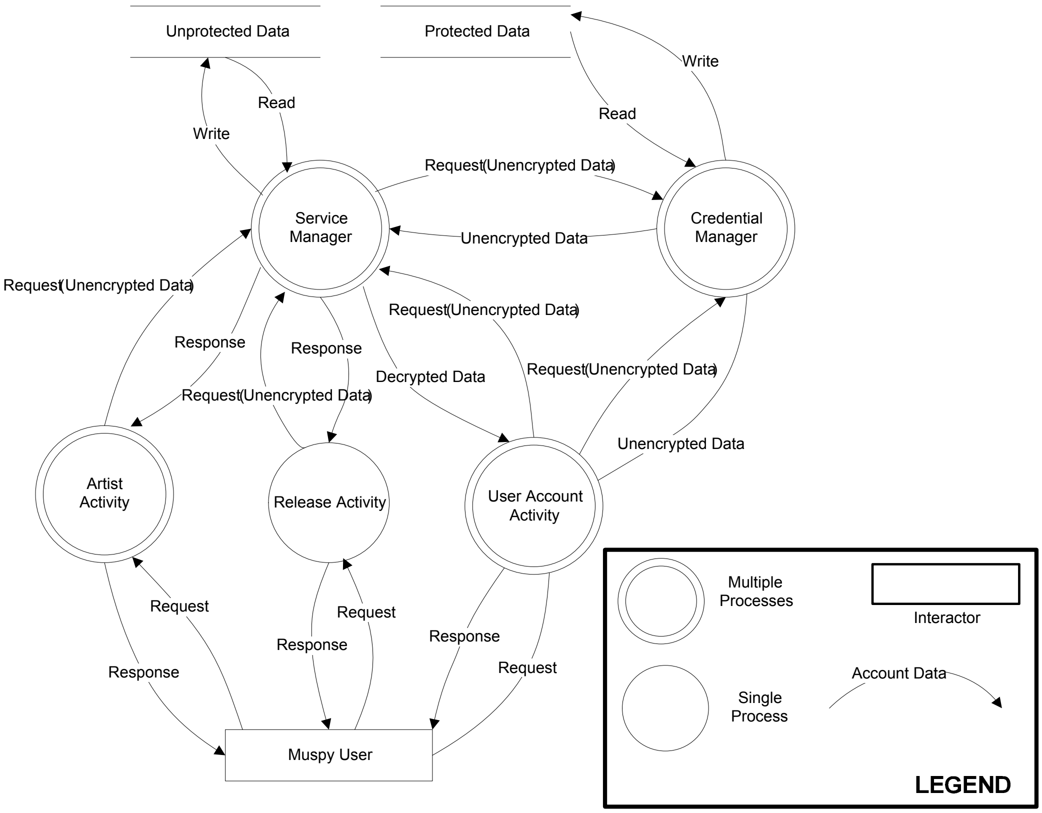


**Figure 3.1:** Muspy DFD Level-1

7. Protected Data: This storage stores sensitive user information. Muspy uses a local database (Android `SharedPreference`) to store the protected data. Protected data are saved and accessed through the "Credential Manager" process (Fig. 3.1).

### DFD Level-2

The DFD level-2 shows the decomposition of several components of the Level-1 DFD. Following are the components of the DFD Level-2:

1. My Artist Activity (Artist Activity): This activity takes the artist name as a parameter and communicates with the "Muspy Client" process to fetch corresponding artist's data (Fig. 3.2). When a user is logged in to the system, this activity populates the favorite artists of the user by default.

2. Search Artist Activity (Artist Activity): This activity passes an artist name as a parameter and retrieves the data through the “Muspy Client” component of the “Service Manager”.

3. Import Last FM activity (Artist Activity): This activity imports the list of artists from a user’s Last.FM profile to Muspy through HTTP protocols.

4. Signup Activity (User Account Activity): This process sends user login credentials to the “Service Manager” through HTTP request and simultaneously sends it to the “Credential Manager” in order to store the data in the local database (Fig. 3.2).

5. Signin Activity (User Account Activity): This activity takes the email and the password and checks them against the locally stored data records before allowing the user to login.

6. Reset Password Activity (User Account Activity): This activity sends the user’s email address to the “Muspy Client” which in turn sends a link to the email address to perform the reset operation (Fig. 3.2).

7. Settings Activity (User Account Activity): “Settings activity” allows the user to change and save account settings and notifications.

8. Muspy Client (Service Manager): “Muspy client” is a service provider process that works as a middleware and convey bidirectional communication with the activity classes (in the UI) and the data storage. For instance, a user searches for a favorite artist by providing an artist name to the “Muspy Client” and the client performs HTTP get operation to retrieve the artists’ information.

9. Utils (Service Manager): This utility class is responsible to retrieve a user credentials stored in local data storage.

10. User Settings (Service Manager): “Muspy Client” gets each user’s settings as an object of the “User Settings” class. It extracts necessary information from the object (e.g., notification settings) to perform relevant operations (e.g., notify the user if sending notification is enabled).

11. Muspy Application (Credential Manager): “Muspy Application” is the top level process of the “Credential Manager” that maintains the dataflow communication between the processes of the “Service Manager”, the user account activity processes and with the local data storage. It also communicates with the “Simple Crypto” process in order to encrypt and decrypt the password.

12. Simple Crypto (Credential Manager): This process encrypts/decrypts the password and receives/pass it from/to the “Muspy Application” (Fig. 3.2).

## 3.3 Annotation Process

The first step of the Scoria Analysis is to annotate the code. I import the Muspy system into Eclipse as an Android application and annotate the code. After adding the annotations, I run the typechecker and address the typechecker warnings.

### 3.3.1 Design Intent

I use existing documentation and inspect the code of Muspy and the code seems to follow the design intent as State-Logic-Display. The intent consists of three architectural tiers. I choose three top level domains to represent these architectural tiers; `UI` (for Display), `LOGIC` (for Logic) and `DATA` (for State). The corresponding formal domain parameters are U, L, and D allow the objects to share state across domains. I use the `owned` domain to annotate strictly encapsulated objects, and annotate static fields and string type variables into the `shared` domain.

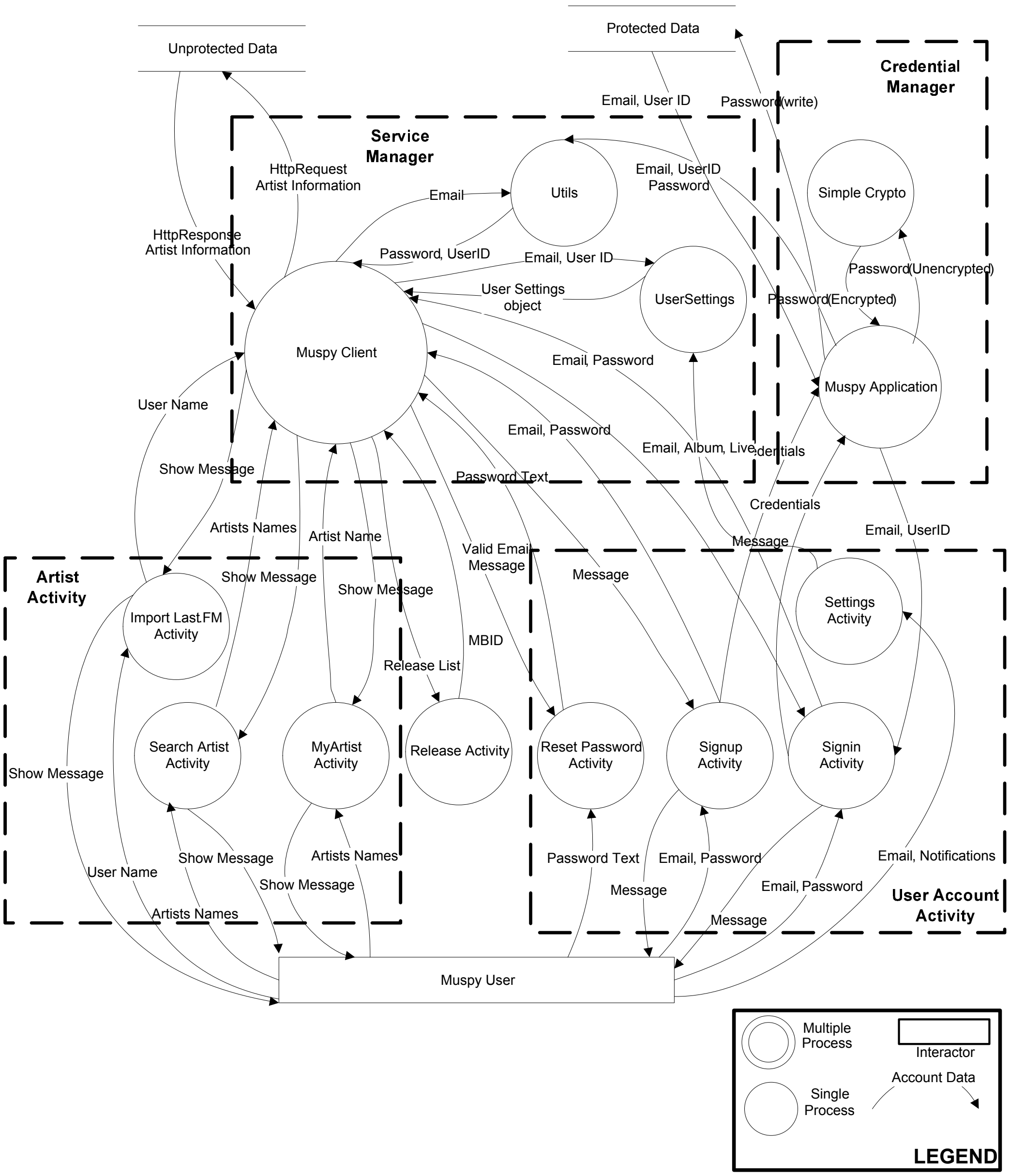


**Figure 3.2:** Muspy DFD Level-2

## 3.3.2 Default Annotations

During the annotation process, I use an existing tool named ArchDefault that partially automates the annotation process considerably and reduces the overhead of adding manual annotations. ArchDefault generates a map that contains a list of the types and an architect needs to assign the required domains and domain parameters to each type. I manually edit the map with corresponding values and properties and

```
public class MuspyApplication<U, L, D> extends Application<U,L,D>
{
        //Added by ANAM
        public MuspyApplication(){super();}
        SimpleCryptoOld<D<D>> SCO = new SimpleCryptoOld();
        SimpleCrypto<D<D>> SC = new SimpleCrypto();
        String<shared> email = null;
}
```

**Figure 3.3:** Code modification: missing constructor

set the properties `takedomainParams` as 'true' and `propagateToSubTypes` as 'true' to propagate the domain parameters across subtypes. I validate the map using the validation functionality of the tool before the default annotations.

### 3.3.3 Root Class and Manual Annotations

Although ArchDefault adds most of the annotations and propagates the domain parameters to the subtypes, the architect may need to modify some of the existing annotations and manually annotate the complex expressions that the tool does not consider. This is because ArchDefault is not a smart inference tool. It may add annotations that do not typecheck. I modify the annotations in order to preserve its consistency with the design intent. I also add annotations that are missed by the default tool especially in case of primitive types and types from the referenced libraries. Based on the priority of the warnings generated by the typechecker, I update the annotations. I minimize the number of warnings to 176 from around 5000 warnings. I modify the code in a few cases e.g., missing public constructor. For example, MuspyApplication had a private constructor that prevents the type for being instantiated (Fig. 3.3).

As the Scoria analysis starts from the entry point of a root class, I manually prepare a root class by instantiating some of the uninstantiated objects. I also declare the three top level domains (`UI`, `LOGIC` and `DATA`) in the root class.

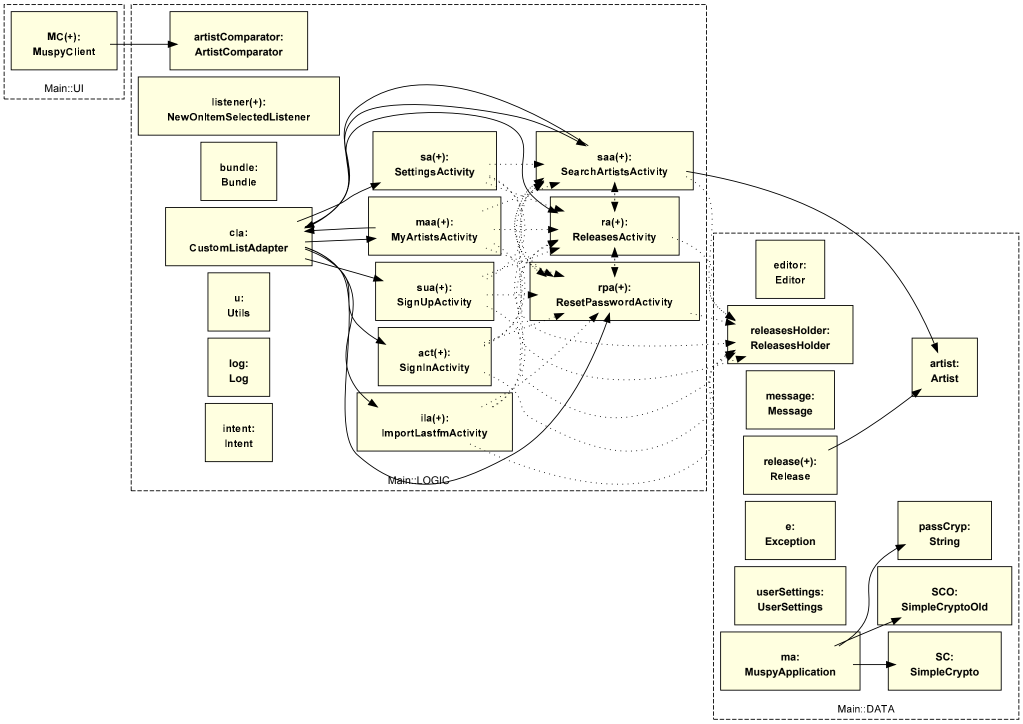


**Figure 3.4:** Muspy OOG (Three top-level domains and collapsed objects)

## 3.4 Object Graph Extraction

I extract the OOG of Muspy. I also extract the OGraph to observe the internal representation of the OOG, and the dataflow communication edges between the objects. I manually study the conformance of the OOG with the extracted DFD Level-2 of the system.

The OGraph of Muspy is large enough to zoom in and follow the dataflow edges between the objects. The expanded view of the OOG is also very large. The collapsed view and the expanded view of one object ("MuspyClient") of the OOG have been shown in (Fig. 3.4), and (Fig. 3.5) respectively.

## 3.5 Conformance with the DFD

In the conformance diagram (Fig. 3.7), a black arrow represents a points to edge, a red arrow shows an export edge, and a blue arrow represents an import

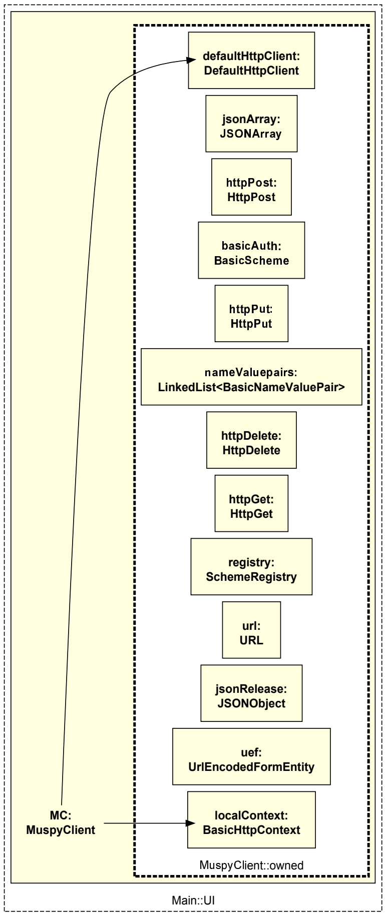


**Figure 3.5:** Muspy OOG expanded view of MuspyClient

edge. The dotted boxes represent three top level domains. The diagram shows the dataflow communication edges between RelseaseActivity and MuspyClient in which ReleaseActivity sends 'MBID' (unique MusicBrainzID of an artist) to the MuspyClient (shown by an export edge) and MuspyClient returns the releases information (shown by an import edge) of an artist that writes a field of the ArtistActivity object. Similarly, there are both an import and an export data communications edges from MuspyApplication to SharedPreference.

## 3.6 Refinement of the Annotations

Since String type objects are annotated in the shared domain by the default tool, some of the strings are significant but get merged in the OGraph with less significant

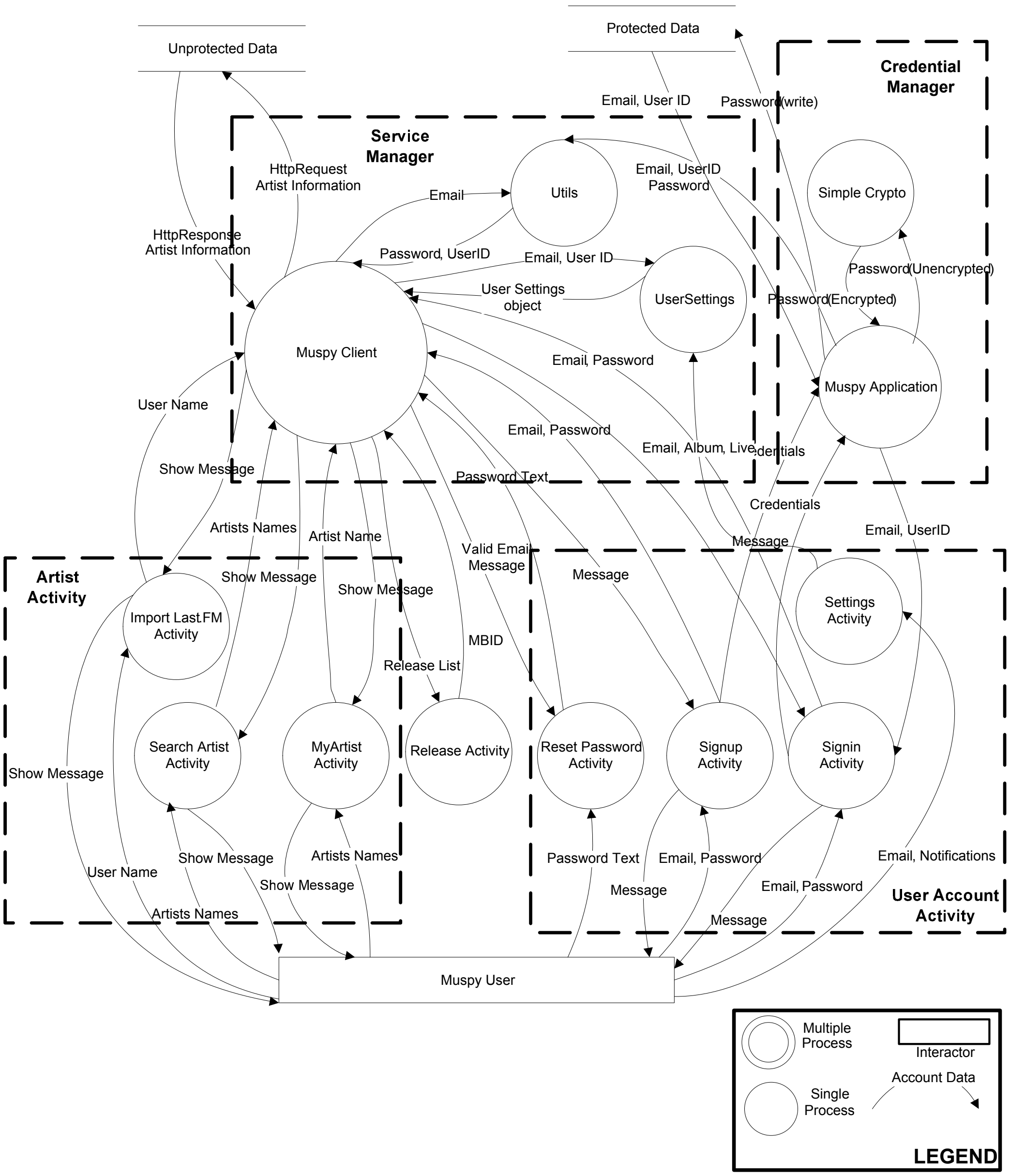


**Figure 3.6:** Muspy DFD (L-2)

objects. As a result, important dataflow edges relevant to a particular String object may be missing. For instance, the 'password' field of type MyspyApplication was initially annotated in the shared domain. I modify the annotation to the DATA domain to keep consistency with the design intent. The intent is that the DATA domain should contain all the significant objects of the system including the login credentials. Moreover, changing the annotation in one statement of the program may lead to annotation warnings elsewhere. I modify the annotation of the "Password"

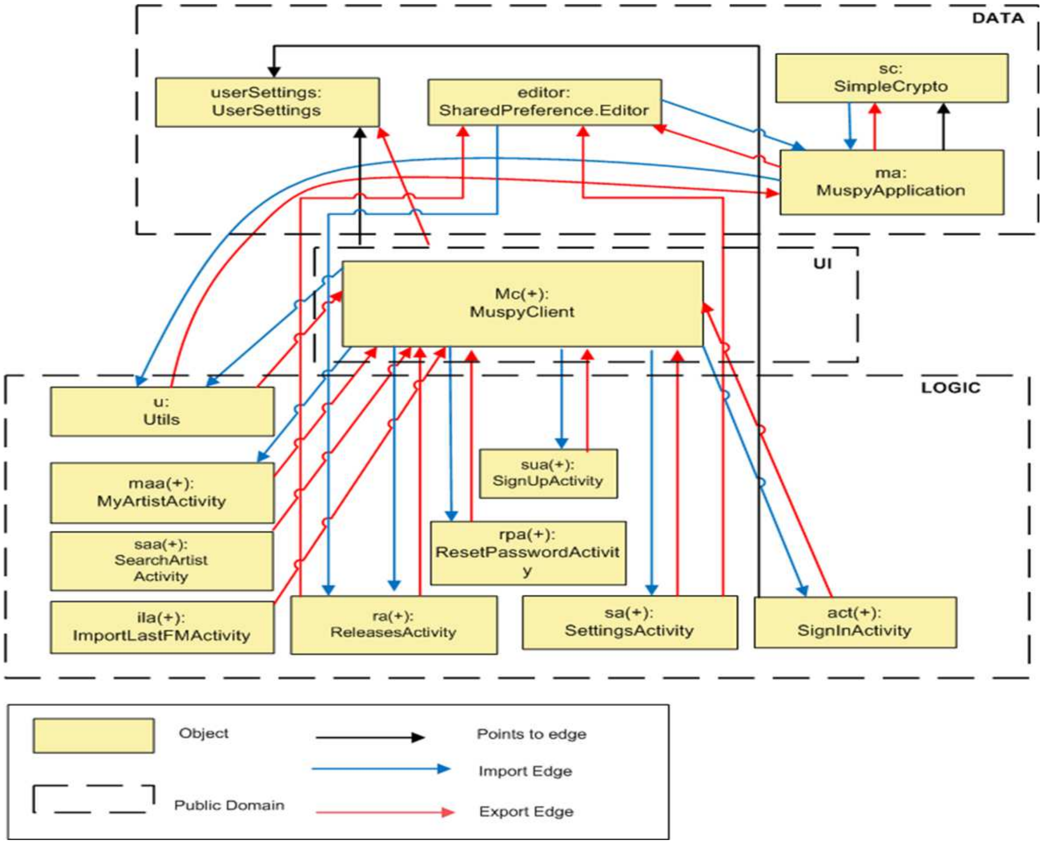


**Figure 3.7:** Muspy OOG Conformance L-2

```
class MuspyApplication<U, L, D> extends Application<U, L, D> {
        String<shared> passCryp = null;
        String<D> passCryp = new String();
        passCryp = null;
        passCryp = SC.encrypt(Constants.SECRET.SEED, pass);
}
class SimpleCrypto<D> {
        String<D> encrypt(String<shared> seed, String<D> cleartext){
          if(cleartext == null)
            return null;
        byte[]<shared> rawKey = getRawKey(seed.getBytes());
        byte[]<shared> result = encrypt(rawKey, cleartext.getBytes());
        return toHex(result);
        }
}
```

**Figure 3.8:** Refinement of annotation: prevention of merging objects

field to the `DATA` domain which is set by a method invocation (`encrypt` method) of the `SimpleCrypto` object (Fig. 3.8). Thus, the return type of the `encrypt` method of `SimpleCrypto` is also required to assign in the `DATA` domain (Fig. 3.8).

## 3.7 Constraints

I implement two constraints for the subject system Muspy. The first constraint evaluates if Scoria can detect an injected vulnerability regarding the Android log file. The second constraint addresses one architectural flaw that is common in Android application development.

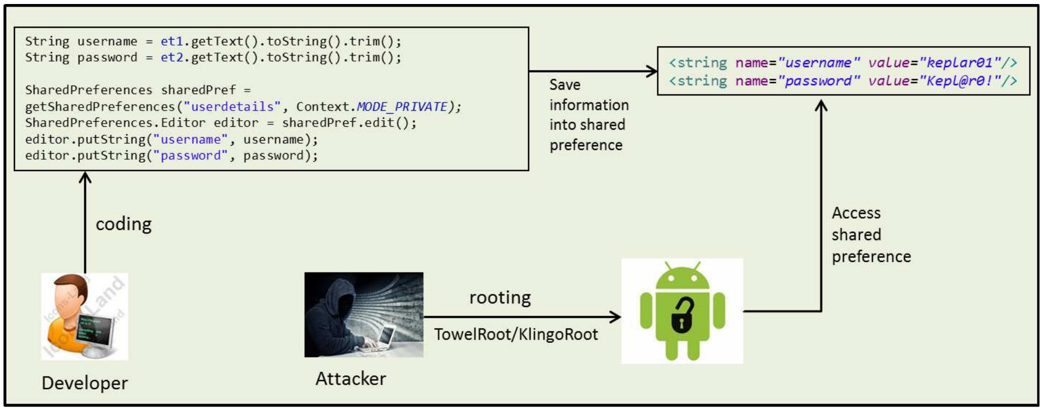
### 3.7.1 Constraint-1: Vulnerability regarding Android SHAREDPREFERENCE

**Android SHAREDPREFERENCE.** Android `SHAREDPREFERENCE` is an interface to store and manipulate data locally of the device. `SHAREDPREFERENCE` is an xml data structure that stores the data in key-value pairs. It allows primitive data types: Boolean, Float, Int, Long and String. An Android developer can easily access an instance of the `SHAREDPREFERENCE` by invoking the `getSharedPreferences` method. Multiple `SHAREDPREFERENCE` objects can exist in a single application.

Device loss is common these days and attackers can get easy access to personal as well as corporate data once they get hold of a device. Corporate data can contain secret information that can lead an attacker to hack into significant resources. The consequences of attacking a lost device can be worse if the device is rooted with permission. There are a number of tools available to enroot a device that can allow an attacker to access all locally stored data (e.g., TowelRoot, KlingoRoot etc.)[3]. Hence, the local storage (`SHAREDPREFERENCE`) is vulnerable to attack. Encrypting the confidential data can be an effective measure to secure the data.

The entire process of the `SHAREDPREFERENCE` attack (of a lost Android device) includes two actors; an Android developer and an Attacker. The developer stores sensitive data (e.g., user name, password) into the `SHAREDPREFERENCE` in an xml

[3] http://www.askvg.com/guide-how-to-root-android-mobile-phones-and-tablets/



```
String username = et1.getText().toString().trim();
String password = et2.getText().toString().trim();

SharedPreferences sharedPref =
getSharedPreferences("userdetails", Context.MODE_PRIVATE);
SharedPreferences.Editor editor = sharedPref.edit();
editor.putString("username", username);
editor.putString("password", password);
```

```
<string name="username" value="keplar01"/>
<string name="password" value="Kepl@r0!"/>
```

**Figure 3.9:** Vulnerability of SHAREDPREFERENCE (attacker accesses the local storage)

format. On the other hand, an attacker can enroot the device (if not rooted), and access the contents of the SHAREDPREFERENCE (Fig. 3.9). In this example, the attacker can access the contents of the "username" and "password" fields that have the values 'kepLar01', and 'kepL@r0!' respectively (Fig. 3.9).

### Implementation

While signing in, a user provides the user name and the password and Muspy stores the encrypted password into SHAREDPREFERENCE. SignHandler (private class of SignInActivity) passes the un-encrypted credentials to the MuspyApplication (Fig. 3.10). Although MuspyApplication encrypts the password, there may be an execution in which SHAREDPREFERENCE can receive an un-encrypted password to store (Catch block, Fig. 3.10).

I draw a partial OGraph showing the objects and the edges related to the vulnerability in order to depict the vulnerability regarding SHAREDPREFERENCE (Fig. 3.11). ma:MuspyApplication imports (blue edge) the password (passCryp:String) from act:SignInActivity and exports (red edge) it to the editor:SharedPreference.Editor (Fig. 3.11). Both ma:MuspyApplication and editor:SharedPreference.Editor are annotated to be in the DATA domain.

```
class SignInActivity<U,L,D> extends AbstractActivity<U, L, D>{
 private class SignHandler<U, L, D> extends Handler<U, L, D>{
 void handleMessage(Message<D> msg) {
 MuspyApplication<D<U,L,D>> muspyApplication = (MuspyApplication)
   getActivity().getApplication();
 String<D> pass = getActivity().passwordEditText.getText().toString();
 String<shared> email = getActivity().emailEditText.getText().toString();
 String<shared> userID = getActivity().userSettings.getId();
 muspyApplication.setCredentials(email, pass, userID);
  }
 }
}
class MuspyApplication<U,L,D> extends Application<U,L,D>{
 void setCredentials(){
        String<D> passCryp = new String();
        Editor<D<U,L,D>> editor = new Editor();
        try {
               if (Build.VERSION.SDK_INT < Constants.API_421) {
                      passCryp = SCO.encrypt(Constants.SECRET.SEED, pass);
                }
               else{
                      passCryp = SC.encrypt(Constants.SECRET.SEED, pass);
               }
        catch (Exception<shared> e){
               passCryp = pass;
        }
        editor.putString(Constants.PREF_PASS, passCryp);
        editor.putString(Constants.PREF_USERID, userID);
        editor.commit();
 }
}
```

**Figure 3.10:** Unencrypted password is stored while executing catch statement)

### Result

I use the object provenance and object transitivity features of Scoria to address this vulnerability and implement the constraint. According to the object provenance, the sensitive object (`passCryp:String`) that flows from `act:SignInActivity` to `ma:MuspyApplication` should not also flow from `ma:MuspyApplication` to `editor:SharedPreference.Editor`. The implementation of the constraint (as a JUnit test) is shown in Fig. 3.12. I run the test case and Scoria finds the suspicious edge with no false positive. Scoria shows the suspicious export edge from `ma:MuspyApplication` to `editor:SharedPreference.Editor` with the flow object

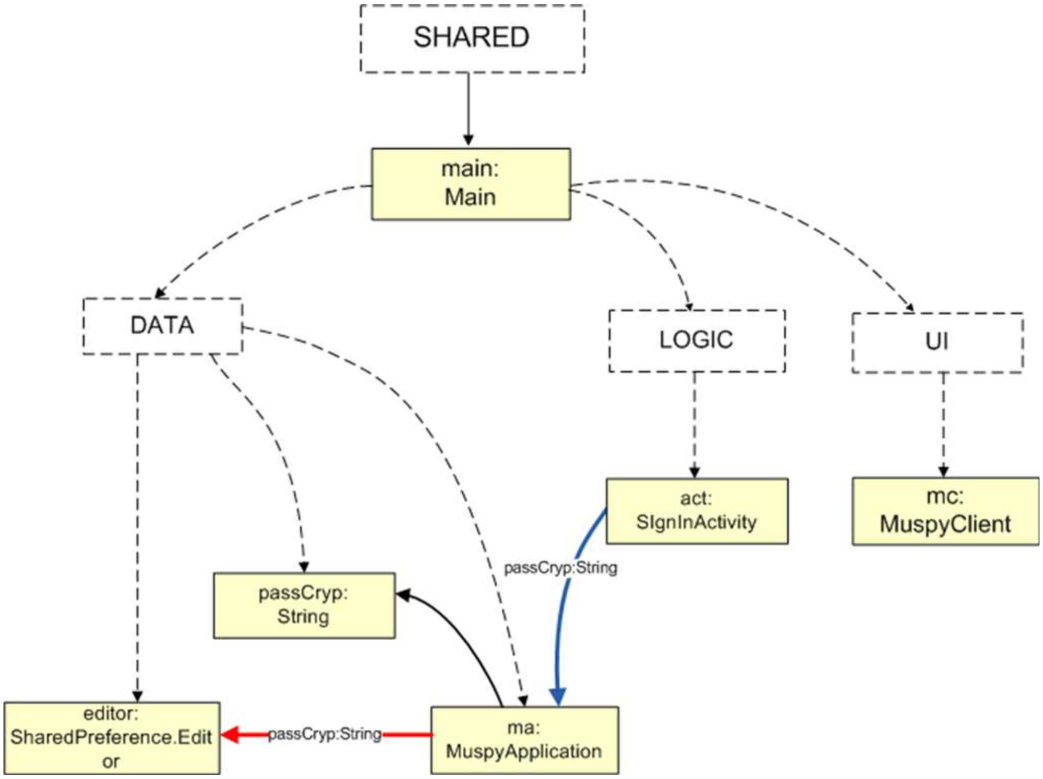


**Figure 3.11:** SHAREDPREFERENCE vulnerability (partial OGraph)

`passCryp:String` due to the method invocation `editor.putString()` (Fig. 3.13).

### 3.7.2 Constraint-2: Vulnerability regarding Android system Log

An Android developer can use the default logging functionality of the android.util package to store the system logs. Although Android log files are not generally visible, the Android logging system allows a developer to access and view the stack traces of the system failures and all system messages that help the developer to debug the errors (using `LogCat`). `LogCat` is a command-line functionality that allows the user to search for logs of various programs. `LogCat` can be used with adb (Android Debug Bridge) command shell (e.g.,[adb] logcat [option] ... [filter-spec] ...,).

Moreover, a number of tools are available to visually look into the contents of the log files (e.g., aLogCat, LogViewer etc.) that make the Android logs more vulnerable.

```
@Test
void CheckPasswordVulnerabilityObjectProvenance() {

Set<IObject> cryptos = secGraph.getObjectsByCond(new InstanceOf(
              "com.danielme.muspyforandroid.activities.SignInActivity.SignHandler"));
Set<IObject> muspyApps = secGraph.getObjectsByCond(new InstanceOf(
              "com.danielme.muspyforandroid.MuspyApplication"));
Set<IObject> logs = secGraph.getObjectsByCond(new InstanceOf(
              "com.danielme.muspyforandroid.Editor"));

Assert.assertFalse(cryptos.isEmpty());
Assert.assertFalse(muspyApps.isEmpty());
Assert.assertFalse(logs.isEmpty());

boolean isSecure = true;
for (IObject cp : cryptos){
 for (IObject mu : muspyApps){
  for (IObject lg : logs){
    Set<DataFlowEdge> sEdges = secGraph.checkObjectProvenance(
         cp, mu, mu, lg);
         scoria.displayWarnings(sEdges);
          isSecure = isSecure && sEdges.isEmpty();
       }
  }
}
if (!isSecure)
 System.out.println("Information disclosure found for unencrypted password");
else
 System.out.println("No information disclosure found for unencrypted password");
}
```

**Figure 3.12:** Mupsy constraint1: JUnit test

```
Sanity Check Visitor
Sanity Check: Null getParent(): O_world
Copying the OGraph into a SecGraph
Suspicious edge : ma:MuspyApplication -> editor:Editor [passCryp:String]
       MuspyApplication setCredentials editor.putString(Constants.PREF_PASS,passCryp)

Information disclosure found for unencrypted password
```

**Figure 3.13:** Mupsy constraint1: output of JUnit test

Additionally, the Android debugging tool named DDMS [4] provides a log file viewer. With a device emulator and a connected Android device, DDMS shows the system logs of the connected device in an Eclipse window.

[4] DDMS stands for Dalvik Debug Monitor Server

Moreover, research shows that the Android log is not a trusted source to store confidential information. Security researchers at IBM reported multiple vulnerabilities when the `Firefox` application on Android writes the directory name of the user's profile in the system log .[12] Although the profile directory name is randomly generated to prevent unwanted directory access, researchers were able to bypass one profile directory name using the brute force method. I write a constraint to verify if Muspy writes any significant information into the Android log and Scoria did not find such a vulnerability. So, I inject a vulnerability to check if Scoria can detect the injected vulnerability regarding the log file.

### Implementation

I use the `checkFlowIntoSink` functionality of Scoria to check if any suspicious edge is found regarding the flow of the confidential information to the untrusted sink (i.e., the Log). To implement this test case, the trust level property of the `android.util.Log` object is defined as low and all the `String` objects of the `DATA` domain are labeled as confidential. The sink and the flow properties are passed to the `getFlowIntoSink` method.

### Result and Injected Vulnerability

The initial results of the constraint show that Muspy does not store any confidential information in the log file. I inject a vulnerability by creating a custom exception class (`InjectException` inherited from the default Java `Exception` class). For storing the sensitive information, I retrieve the password and the email address of a Muspy user and store it in the log file. The annotated version of the `InjectException` class has been shown in (Fig. 3.15). I instantiate the `InjectException` in some places of the code (Fig. 3.15).

Due to the method invocations (`ma.getPass()` and `ma.getEmail()`) in the type

```
@Test
public void checkUntrustedSinkforConfidentialData() {
// Log object is not trustworthy
secGraph.setObjectProperty(TrustLevelType.Low, new InstanceOf(new Type(
"android.util.Log")));
// All strings in domain 'DATA' are confidential
secGraph.setObjectProperty(IsConfidential.True, new IsInDomain("DATA",
new Type(String.class.getName())));
Property[] snkProps = { TrustLevelType.Low };
Property[] flwProps = { IsConfidential.True };
if (secGraph.checkFlowIntoSink(snkProps, flwProps)) {
System.err.println("Information disclosure exists.");
Set<SecEdge> sEdges = secGraph.getFlowIntoSink(snkProps, flwProps);
SecurityAnalysis.getInstance().displayWarnings(sEdges);
scoria.displayWarnings(sEdges);
    } else {
       System.out.println("No information disclosure found.");
    }
}
```

**Figure 3.14:** Test case to verify if sensitive information is passed to Android system log

```
class InjectException<U, L, D> extends Exception<U, L, D> {
       public InjectException() {
       MuspyApplication<D<U, L, D>> ma = new MuspyApplication();
       String<D> pass = ma.getPass();
       String<shared> email = ma.getEmail();
       Log<L> l = new Log();
       l.e("Password: ", pass);
 }
}
class ResetPasswordActivity<U, L, D> {
       void run() {
       try{
         boolean<shared> isValid = resetPassword(this.emailEditText.getText());
       }
       catch (InjectException ex) {
         throw new InjectException();
       }
 }
}
```

**Figure 3.15:** Custom expception (InjectException) class leaks password to log file.

InjectException, IE:InjectException retrieves the password and the email from ma:MuspyApplication. Import edges in the partial OGraph reflect the field writes in IE:InjectException class (Fig. 3.16). Method invocation l.e() creates an export edge from IE:InjectException to l:Log shows a flow of the confidential object

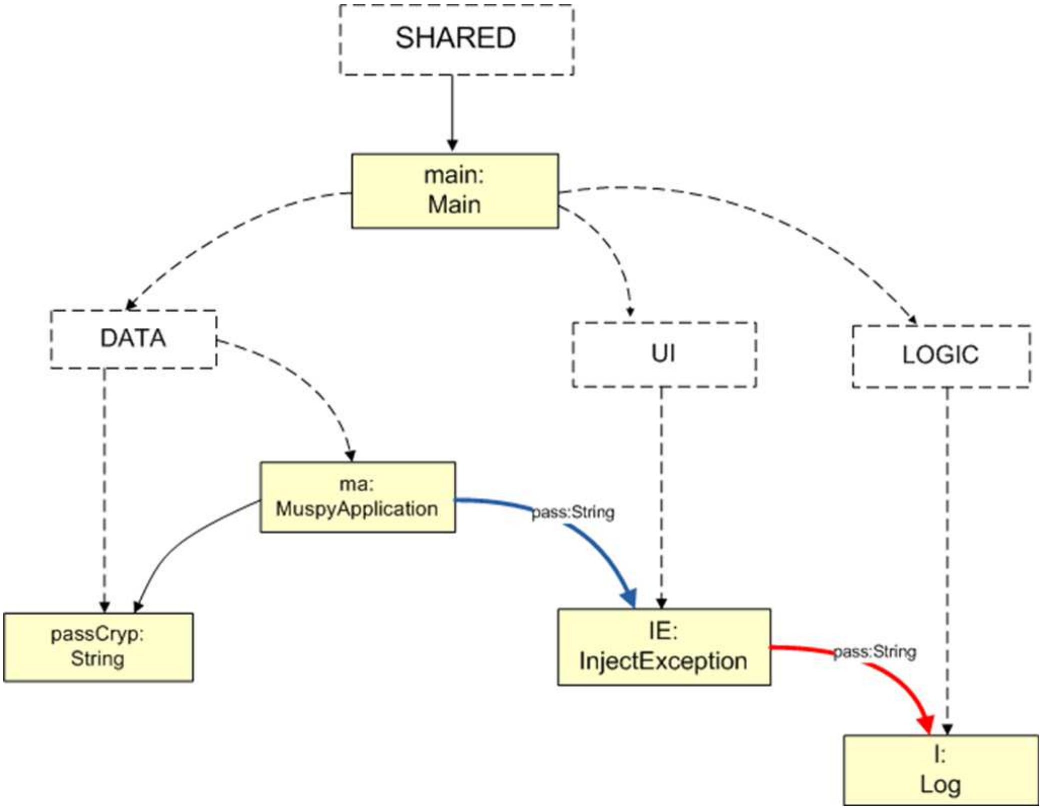


**Figure 3.16:** Injected vulnerability (partial OGraph)

```
Sanity Check Visitor
Sanity Check: Null getParent(): O_world
Copying the OGraph into a SecGraph
Suspicious edge : IE:InjectException -> l:Log [pass:String]
        InjectException InjectException l.e("Password:", pass)

Information disclosure exists
```

**Figure 3.17:** Constraint-2 (JUnit Test Case Output: Suspicious Edge(s))

`pass:String` (Fig. 3.16). I use object transitivity that is implemented using object provenance. After running the constraint, Scoria finds the suspicious edges by checking the transitive dataflow communication (Fig. 3.17).

## 3.8 AF-index of the vulnerabilities

Since both constraints use object transitivity and object provenance, the weighted sum for both of these constraints are 5 in the AF-index. Thus, both of these vulnerabilities rank high on the AF-index scale and are more likely to be architectural flaws.

## 3.9 Conclusion

The Scoria analysis of the subject system Muspy concludes two important points; First, Android `SHAREDPREFERENCE` is not trustworthy data storage to store sensitive unencrypted information (e.g., email, password, user name, etc.). Muspy stores unencrypted password in the `SHAREDPREFERENCE`. Second, Scoria successfully reports the non-existence of any suspicious edge when there is no vulnerability. Besides, it reports the suspicious edge when a vulnerability is injected and confidential information is saved in the log file.

# CHAPTER 4: ERMETE SMS

This chapter gives an overview of the Ermete SMS, its architecture, the vulnerabilities and some Scoria constraints that find some potential vulnerabilities.

## 4.1 Ermete SMS Overview

Ermete SMS is an Android application that allows the user to send free SMS via web protocols. It is a free distribution in Google Market and available to download from the F-Droid[1] repository. Ermete SMS allows only Vodafone and TIM (Telecom Italia Mobile) users to exchange messages with their contacts. Besides, it stores history of messaging including message content for each account.

Ermete SMS has following features:

1. Create and Modify Accounts: A user can create a new account using a user name and a password for the providers Vodafone and TIM. The user can also modify the existing account information.

2. Multiple Accounts: A user can have multiple accounts and she can switch between her accounts.

3. Reset Password: In case of forgetting password, the user can reset the password by providing the email address, and Ermete SMS sends a reset link to the email address.

4. Send Message: A user can send a message to multiple contacts. The messages are stored in a local storage.

5. User Notification: Ermete SMS allows the user to get notification of received messages.

[1] https://f-droid.org/repository/browse/

## 4.2 System Architecture

Ermete SMS is 6 KLOC, so it is a mid-size Android system. I run a metrics tool named Stan[2] in order to investigate the code statistics (shown in Table 1.1) of Ermete SMS.

### 4.2.1 DFD

To understand the system, I inspect the code, run the system and use existing software documentation. I manually draw two DFDs, a level-1 and a level-2, and manually analyze the conformance of the Level-2 DFD with the extracted OOG.

**DFD Level-1**

The Level-1 DFD of Ermete SMS has three major components; complex process, data storage and external user. Level-1 DFD contains three complex processes, two storages and the external user (Fig. 4.1). Ermete SMS uses `SQLiteDB` and `Sharedpreference` as data storages.

1. Account Activities: This complex process contains account related activities: create, modify and update an account information for the telephony providers TIM and Vodafone.

2. Account Manager: The "Account Manager" connects and manages the engine to communicate with the TIM and the Vodafone servers through HTTP protocols.

3. SMS Service: The "SMS service" consists of an account service and processes relevant to messaging and conversation. It also has a notification manager that notifies the users of new incoming and outgoing messages.

[2] http://stan4j.com/

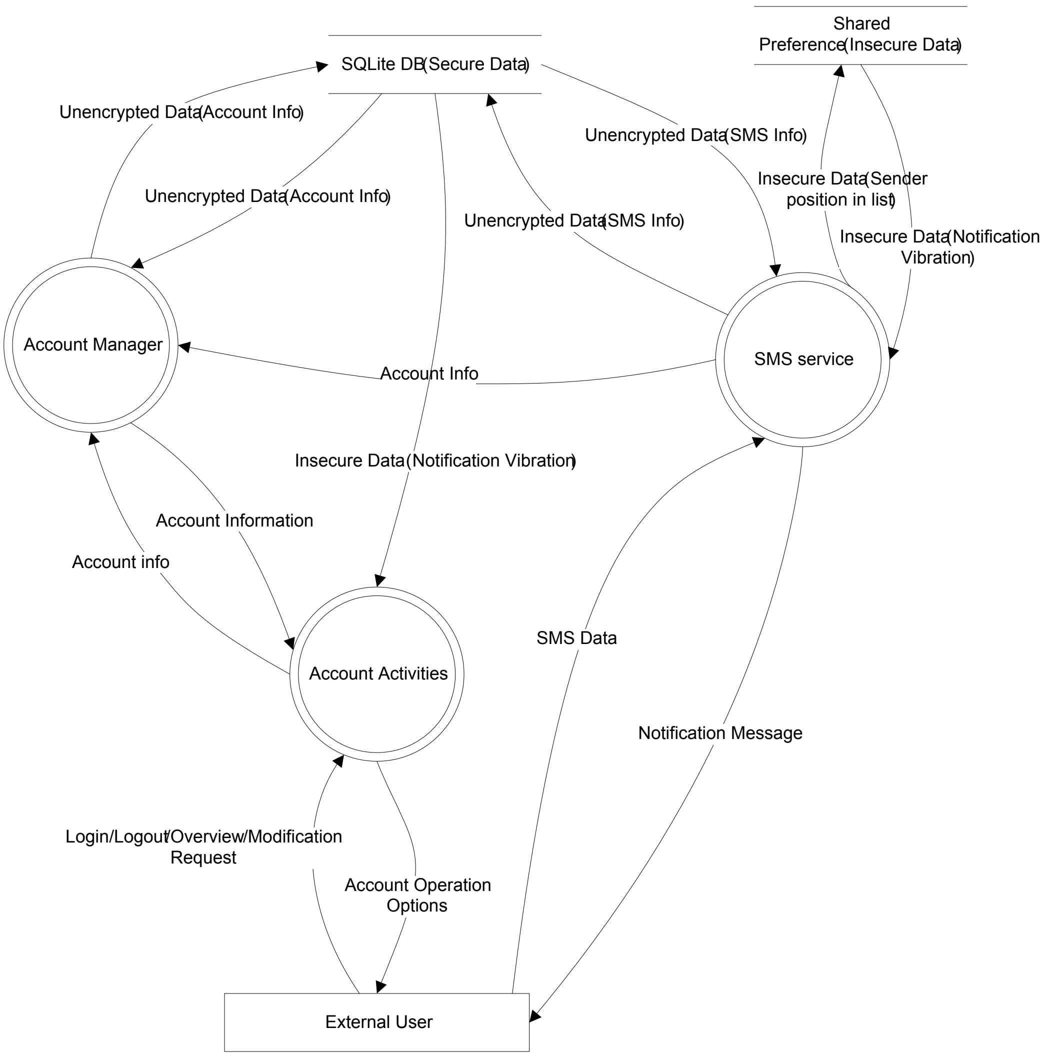

SQLite DB(Secure Data)
Shared Preference(Insecure Data)
Unencrypted Data(Account Info)
Unencrypted Data(Account Info)
Unencrypted Data(SMS Info)
Unencrypted Data(SMS Info)
Insecure Data(Sender position in list)
Insecure Data(Notification Vibration)
Account Manager
SMS service
Account Info
Insecure Data (Notification Vibration)
Account Information
Account info
Account Activities
SMS Data
Notification Message
Login/Logout/Overview/Modification Request
Account Operation Options
External User

**Figure 4.1:** Ermete SMS DFD L-1

### DFD Level-2

The DFD Level-2 shows the decomposition of each complex process of the Level-1 DFD.

1. Account Overview Activity: "Account overview activity" shows the default screen of the system that allows the user to select the account type and redirect the user to either `AccountCreateActivity` or `AccountModifyActivity`.

2. Account Create Activity: It creates a new account object (wrapped with the user name, the password and the provider information) and passes the object

to the `AccountModifyActivity`.

3. Account Modify Activity: This activity gets existing account information from the local storage (`Sharedpreference`) and verifies with the information entered by the user. In order to update the information to the database, it communicates with the `AccountManager` activity. The important dataflow edges are shown as red color in Fig. 4.2.

4. Account Display Activity: This process shows the accounts basic information (including the available contacts) to the user after successful login and modification. It also shows the properties related to the user account. It communicates with the "Account Manager" activity in order to retrieve the list of contacts of a user. The corresponding dataflow edge (in Fig. 4.2) shows that the "Account Manager" sends a list of accounts to the "account display activity".

5. Compose Activity: "Compose Activity" is a part of the "SMS Service" process and it provides the functionality of composing a new message and forwards the composed message to the "Account Service". It is also responsible to store message related information (e.g., index of the message sender position into a list).

6. Account Service: This service process is the middle layer between the "Compose Activity" and the "Conversation Manager" in which the compose activity sends the SMS object to the "Account Service" and it in turn sends the SMS object to the "Conversation Manager". It also sends the account information to the "Account Manager".

7. Conversation Manager: "Conversation Manager" exports the SMS to the senders through HTTP protocol. Besides, it stores the history of messaging to the local SQLiteDB database.

8. Notification Manager: This class notifies the user about new incoming and outgoing messages. It communicates with the "Account Service" process to get the contents of the notification messages.

9. Account Manager: "Account Manager" communicates with the "SMS Service" and with the account activity processes. It sends and receives data from the SQLite database (Fig. 4.2).

10. Account: "Account Manager" sends the account information (e.g., User Name, Password) to the "Account" process and receives the account object in return.

11. Account Connector: "Account Connector" sends the HttpClient object to the account object as a field write of the account that is used later to send the SMS to the receivers.

## 4.3 Annotation Process

I do not use the default annotation tool and annotate the Ermete SMS entirely by hand.

### 4.3.1 Design Intent

Since the software documentation of the Ermete SMS is lacking, it is difficult to get an architectural overview of the system. A few online resources explain the usage of the Ermete SMS from the user point of view, however, most of the resources are available only in Italian. I inspect the code and it seems that the code follows the State-Logic-Display architectural style. So, I represent these tiers using three top level domains; *State* as `DATA` domain, *Logic* as `LOGIC` and *Display* represents the `UI`. Like Muspy, I put uninteresting objects into the `shared` domain and `owned` domain to strictly encapsulate objects into the owning domain of an object.

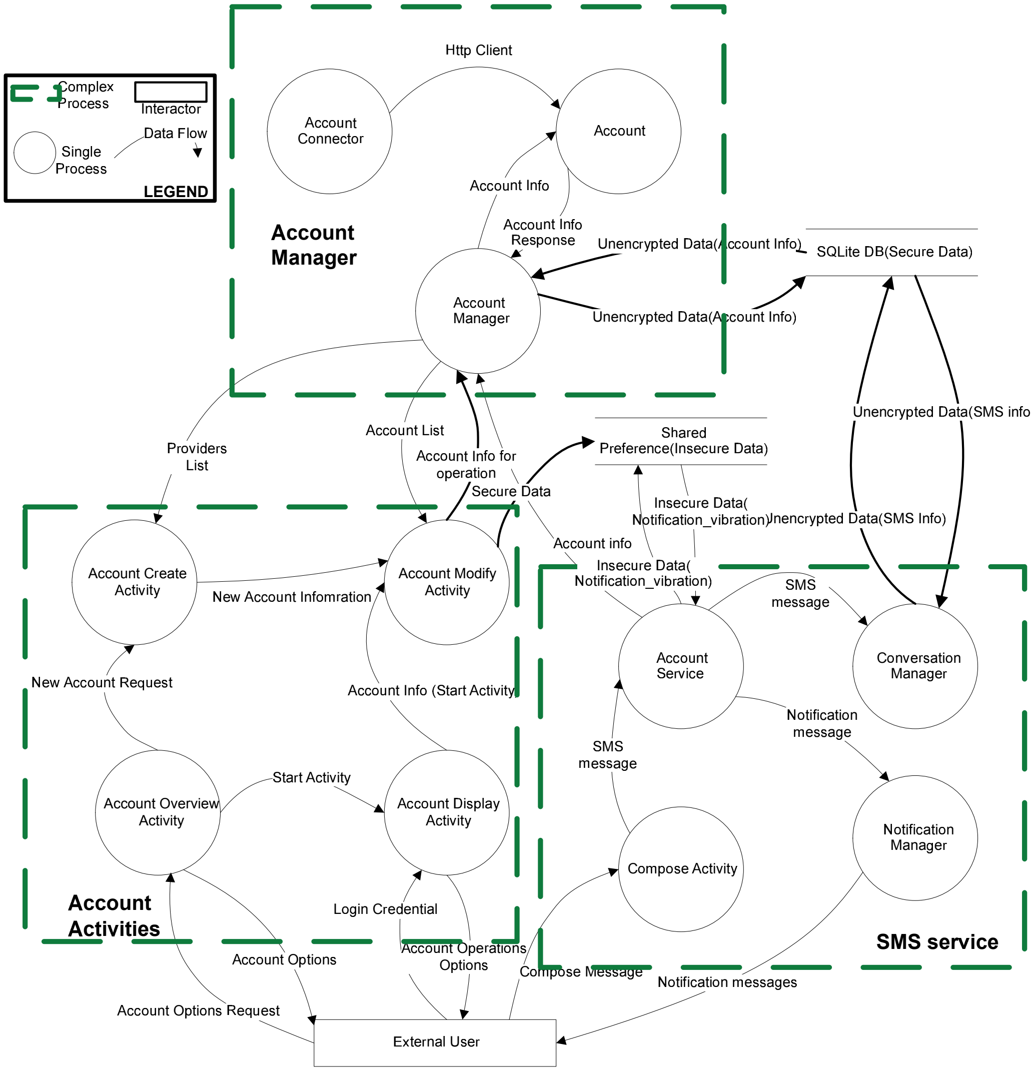


**Figure 4.2:** Ermete SMS DFD L-2

## 4.3.2 Root Class and Manual Annotations

I manually annotate all the types and the subtypes of Ermete SMS with corresponding domains and domain parameters. I create a root class (`Main`) and declare three top level domains. After annotating the code manually, I typecheck the annotations. Typechecker shows a number of warnings with their priorities. I address the warnings according to the priorities. I stop fixing the typechecker warnings when numbers of warnings were reaches around 150. Most of the remaining warnings are related to static variables and fields for which the typechecker always shows warnings. I also instantiate the uninstantiated types in the root class. I use the ArchMetrics

```
SSLSocketFactory<D<U, L, D>> createCustomKeyStoreSSLSocketFactory() {
  try {
     KeyStore<D> keystore = KeyStore.getInstance("BKS");
    //return new CustomKeyStoreSSLSocketFactory(keystore);
    CustomKeyStoreSSLSocketFactory<D<U, L, D>> CSSF = new
    CustomKeyStoreSSLSocketFactory(keystore);
    return CSSF; //Refactored

  } catch (Exception<lent> e) {
    throw new AssertionError(e);
  }
}
```

**Figure 4.3:** Code refactoring: anonymous return object

tool to extract the uninstantiated types.

Moreover, there are few annotation warnings due to anonymous classes of the objects. I refactor the code and convert the anonymous classes into inner classes. As an example in (Fig. 4.3), Moreover, there are few annotation warnings due to anonymous instantiation of the objects. I refactor the code and instantiate them with corresponding variable names. As an example in (Fig. 4.3), the return value of the `createCustomKeyStoreSSLSocketFactory` is anonymous. I create an instance of `CustomKeyStoreSSLSocketFactory` with a variable name "CSSF" and return it.

## 4.4 Object Graph Extraction

I extract the OOG of Ermete SMS and inspect it by expanding the nested boxes of the OOG to verify that it matches the design intent (Fig. 4.4).

I also extract the OGraph that shows the internal representation of the OOG. the top level domains and the corresponding OObjects: `UI` contains the OObjects of the user interface (e.g., `CA:ComposeActivity`, `intent:Intent` etc.), `LOGIC` contains the OObjects that deal with the business logic of the system. The rest of the OObjects are related to the sensitive data (e.g., account, sms) and they belong to the `DATA` domain.

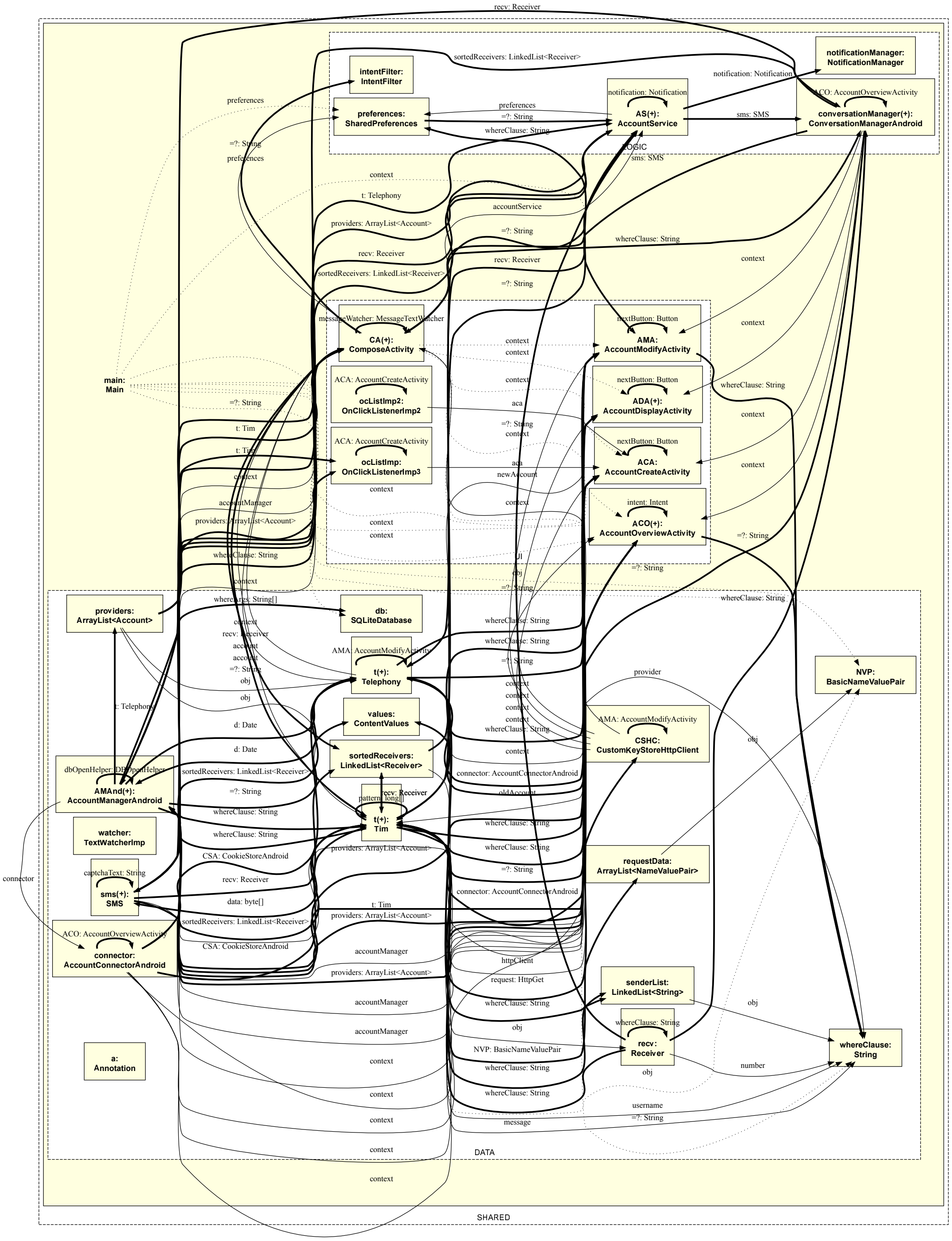

recv: Receiver
sortedReceivers: LinkedList<Receiver>
notificationManager: NotificationManager
notification: Notification
intentFilter: IntentFilter
preferences: SharedPreferences
AS(+): AccountService
sms: SMS
ACO: AccountOverviewActivity
conversationManager(+): ConversationManagerAndroid
LOGIC
messageWatcher: MessageTextWatcher
CA(+): ComposeActivity
nextButton: Button
AMA: AccountModifyActivity
ACA: AccountCreateActivity
ocListImp2: OnClickListenerImp2
ADA(+): AccountDisplayActivity
ocListImp: OnClickListenerImp3
ACA: AccountCreateActivity
intent: Intent
ACO(+): AccountOverviewActivity
UI
main: Main
providers: ArrayList<Account>
db: SQLiteDatabase
NVP: BasicNameValuePair
t(+): Telephony
values: ContentValues
CSHC: CustomKeyStoreHttpClient
dbOpenHelper: DBOpenHelper
AMAnd(+): AccountManagerAndroid
sortedReceivers: LinkedList<Receiver>
t(+): Tim
watcher: TextWatcherImp
captchaText: String
sms(+): SMS
requestData: ArrayList<NameValuePair>
connector: AccountConnectorAndroid
senderList: LinkedList<String>
recv: Receiver
whereClause: String
a: Annotation
DATA
SHARED

**Figure 4.4:** Ownership Object Graph (OOG) of Ermete SMS

## 4.5 Conformance with the DFD

Since, the OGraph is large enough to follow the dataflow edges, I use a tool named ArchDoc that maps the dataflow edges to the corresponding lines of code. The conformance diagram shows the important OObjects, points to edges and the dataflow edges. In the conformance diagram (Fig. 4.5), three top-level container domains are shown by the dotted lines. The boxes represent the objects, the black arrows represent the points to edges, and the blue and red edges represent the import, and export dataflow communications respectively. An example of the conformance can be observed by following the dataflow edges between `AMAnd:AccountManagerAndroid` (subtype of the `AccountManager` type in DFD) and `ADA:AccountDisplayActivity`. `AMAnd` exports a list of accounts to the `ADA`. The field `providers:List<Account>` of `AMAnd` is the flow object of the corresponding communication edge.

## 4.6 Refinement of the Annotations

I refine the annotations because some of the important dataflow edges were missing. The developers of the Ermete SMS use complex expressions that have typechecker warnings, and as a result some important edges are missing from the OGraph. An example is shown in Fig. 4.6. The receiver object is anonymously created (while invoking the method `addReceiver`) and passed to the `sms:SMS` object that causes a missing dataflow edge. I refactor the code by creating a new instance of type `Receiver` separately and pass the object as a parameter to the method invocation.

Moreover, some important objects (especially array of String object) are not initialized with the new keyword and are missed from the OGraph along with the corresponding edges. I modify the code by initializing the object with the `new` keyword (Fig. 4.6).

## 4.7 Constraints

The following sections discuss the implementation of two constraints that find security vulnerabilities of Ermete SMS.

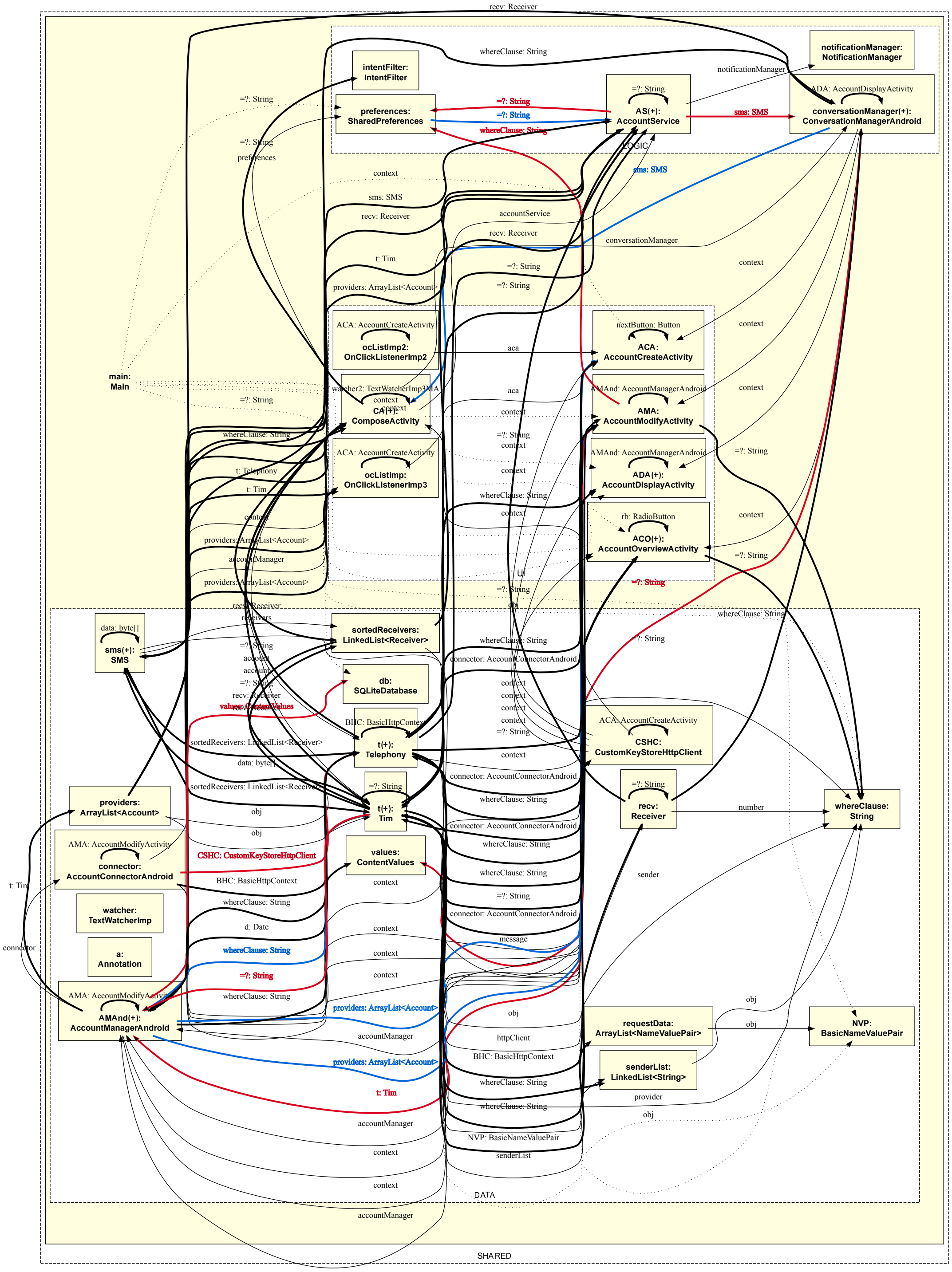

recv: Receiver
whereClause: String
intentFilter: IntentFilter
preferences: SharedPreferences
=?: String
AS(+): AccountService
notificationManager: NotificationManager
notificationManager
ADA: AccountDisplayActivity
conversationManager(+): ConversationManagerAndroid
sms: SMS
LOGIC
context
recv: Receiver
accountService
t: Tim
providers: ArrayList<Account>
conversationManager
ACA: AccountCreateActivity
ocListImp2: OnClickListenerImp2
nextButton: Button
ACA: AccountCreateActivity
aca
watcher2: TextWatcherImp3MA
CA(+): ComposeActivity
AMAnd: AccountManagerAndroid
AMA: AccountModifyActivity
ocListImp: OnClickListenerImp3
ADA(+): AccountDisplayActivity
t: Telephony
rb: RadioButton
ACO(+): AccountOverviewActivity
main: Main
UI
accountManager
data: byte[]
sms(+): SMS
sortedReceivers: LinkedList<Receiver>
db: SQLiteDatabase
values: ContentValues
BHC: BasicHttpContext
t(+): Telephony
CSHC: CustomKeyStoreHttpClient
connector: AccountConnectorAndroid
t(+): Tim
recv: Receiver
number
whereClause: String
providers: ArrayList<Account>
AMA: AccountModifyActivity
connector: AccountConnectorAndroid
CSHC: CustomKeyStoreHttpClient
values: ContentValues
watcher: TextWatcherImp
a: Annotation
AMAnd(+): AccountManagerAndroid
t: Tim
connector
d: Date
sender
message
obj
httpClient
requestData: ArrayList<NameValuePair>
NVP: BasicNameValuePair
senderList: LinkedList<String>
provider
NVP: BasicNameValuePair
senderList
DATA
SHARED

Figure 4.5: OOG conformance diagram with respect to DFD L-2

### 4.7.1 Constraint-1: Vulnerability of exporting confidential information to untrusted destination

`AccountDisplayActivity` shows the basic information of the account to the user interface of the Ermete SMS. It communicates with the `AccountManager` and receives

```
//sms.addReceiver(new Receiver(address));
Receiver<D<D>> r = new Receiver(address);
sms.addReceiver(r);  // Refactored
String[]<shared[shared]> projection;
//String[] projection = { CLASS, LABEL, USERNAME, PASSWORD, SENDER, COUNT, COUNT_DATE };
projection = new String[] {CLASS, LABEL, USERNAME, PASSWORD, SENDER, COUNT,
COUNT_DATE }; // Refactored
```

**Figure 4.6:** Ermete SMS refinement of the annotations

a list of account objects. While it is only responsible to show the account name in a label and logo of the account in an image placeholder, the account object contains the sensitive information (e.g., password, user name etc.) alongside with the account name and the logo.

Architecturally, the display object should not contain confidential information, and I implement a constraint to evaluate if Scoria can detect this vulnerability. The code view (of this vulnerability) shows (Fig. 4.7) that the field `accounts:List<Account>` of `ADA:AccountDisplayActivity` is set by the method invocation `accountManager.getAccounts()` of object `accountManager:AccountManager`. The code view is obtained by tracing back to the code using ArchDoc from the selected edge of interest. The suspicious edge of interest is shown in the console after running the constraint.

### Implementation

I use ArchDoc to hide the nodes that are irrelevant to the constraint. I manually draw a partial OGraph showing the relevant objects and edges of the constraint. The partial OGraph of the constraint shows three top level domains: `UI`, `LOGIC` and `DATA` (Fig. 4.8). `DATA` contains `accountmanager:AccountManager` object that exports `providers:ArrayList<Account>` to `ADA:AccountDisplayActiviy`. Instead of sending the necessary display-only information of the accounts (account name, logo etc.), it sends the entire account object that also contains the password and the username

```
List<Account><D<D<U, L, D>>> accounts = accountManager.getAccounts();
for (Account<D<U, L, D>> account : accounts) {
  TextView<lent> listItemLabel = (TextView)
  listItem.findViewById(R.id.list_item_label);

  TextView<lent> listItemSender = (TextView)
  listItem.findViewById(R.id.list_item_sender);

  ImageView<U<U, L, D>> listItemLogo = (ImageView)
  listItem.findViewById(R.id.list_item_logo);

  String<shared> label = account.getLabel();
  if (label == null || label.equals(""))
    label = getString(R.string.no_label_text);
  listItemLabel.setText(label);
  listItemSender.setText(account.getSender());
```

**Figure 4.7:** Entire account object is exported that contains sensitive information.

of the accounts. An export edge from `accountmanager:AccountManager` to the untrusted destination (`ada:AccountDisplayActivity`) is shown in red (Fig. 4.8). The idea is to evaluate if Scoria analysis can identify this vulnerability (suspicious edge) of the flow of the confidential data.

### Result

I implement the constraint using the object transitivity feature of Scoria. In this constraint, the flow object contains sensitive information and thus, the security property is set as confidential. On the other hand, the sink `ada:AccountDisplayActivity` is assigned as untrusted. I implement the constraint as a test case and run the test case in JUnit. The result shows that Scoria detects the suspicious edge and reports it in the console output of Eclipse (Fig. 4.9). However, Scoria generates a false positive by detecting a communicating edge between `accountmanager:AccountManagerAndroid` to `ama:AccountModifyActivity` as suspicious.

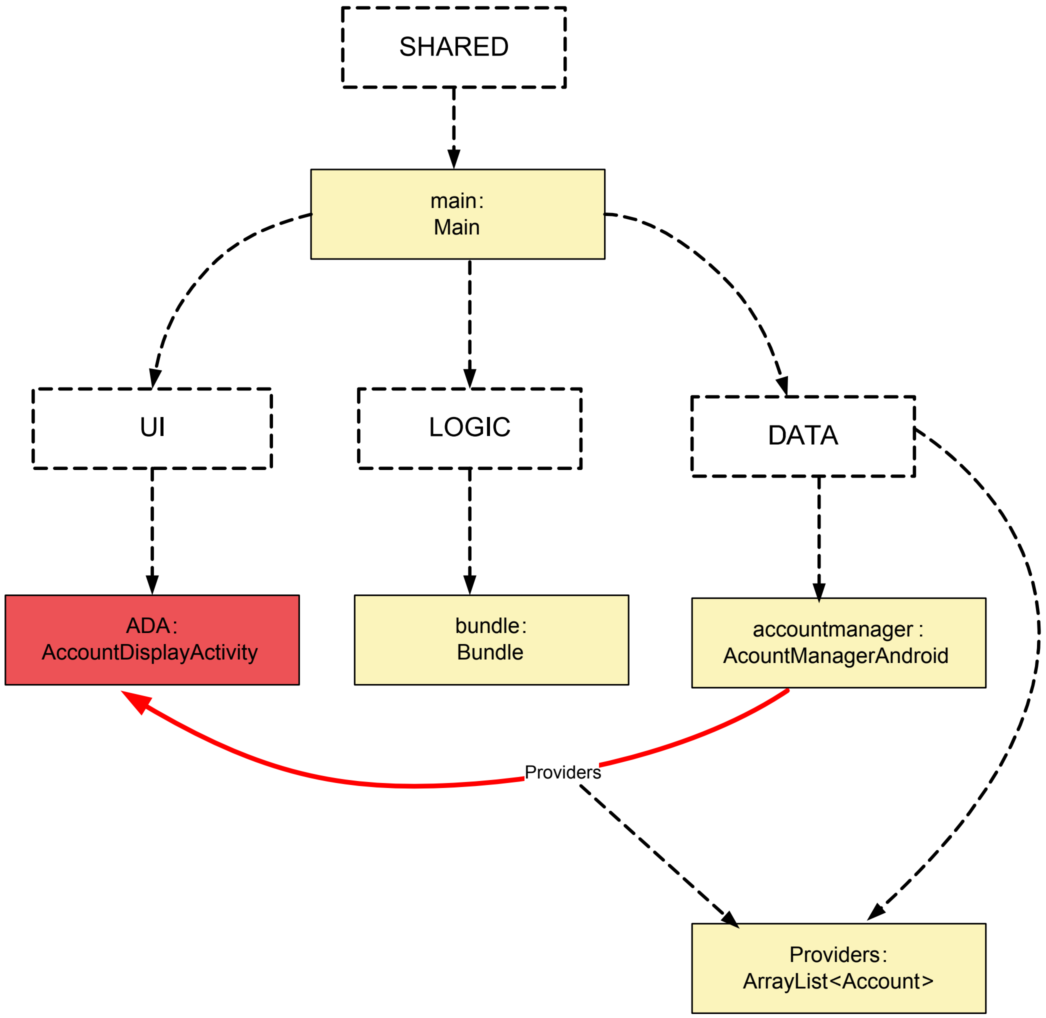


**Figure 4.8:** Vulnerability regarding account object (partial OGraph)

```
Suspicious edge : AMAnd:AccountManagerAndroid -> ADA:AccountDisplayActivity
                               [providers:ArrayList<Account>]
AccountDisplayActivity refreshAccountsList accountManager.getAccounts()

Suspicious edge : AMAnd:AccountManagerAndroid -> AMA:AccountModifyActivity
                               [providers:ArrayList<Account>]
AccountModifyActivity chooseLabel accountManager.getAccounts()
```

**Figure 4.9:** Suspicious edges identified by Scoria.

### 4.7.2 Constraint-2: Vulnerability regarding SQLiteDB

Ermete SMS uses SQLite database to store sensitive information of the user. However, while investigating the possible vulnerabilities regarding the architectural features of Ermete, I learn that storing sensitive information in SQLiteDB is not quite safe (especially without encryption).

**Android SQLiteDB.** SQLite is an open source light-weight SQL database that comes with the default Android distribution. It stores the data in a structured format and supports the regular relational database's features. The database has an extension of .db/.sqlite. The Android package `android.database.sqlite` contains the sqlite class.

By default, SQLiteDB is not accessible from other applications since every Android application has only access to its own private SQLite databases. However, an attacker can gain access to any sqlite database by performing some bypass operations through available Android development features for the developers. Android DDMS allows the attacker to explore the files and folders of an Android device including the databases of each application. Using DDMS, the attacker can also copy a particular file into an external disk. An attacker can further query the sqlite database to fetch information from it. There are a number of online resources available that explain how to hack sqlite database[13] of an Android application. One of the solutions to this problem is to encrypt sensitive data.

I investigate and find that Ermete SMS sends sensitive yet unencrypted data to the SQLite database. The object of type `AccountManager` extracts sensitive information and saves the information inside a wrapper object of type `ContentValues`. Since Sqlite only contains structured data, the `ContentValues` is used to store the data in key-value pairs in which the key represents the table column and the value represents the content of a cell of a row. The corresponding vulnerability is shown in Fig. 4.10. The method `InsertAccountObject` of `AccountModifyActivity` passes the account object to the `accountManager:AccountManagerAndroid`. Later, `insert` method of `AccountManagerAndroid` sets the wrapper object (of `ContentValues`) and pass it to the `db:SQLiteDatabase` object. In this entire path, the confidential information was not encrypted.

```
private void InsertAccountObject(Account<D<U, L, D>> acc){
        AccountModifyActivity<D<U, L, D>> AMA = new AccountModifyActivity();
        accountManager = new AccountManagerAndroid(AMA);
        accountManager.insert(acc);
}

public void insert(Account<D<U, L, D>> newAccount) {
  ContentValues<D> values = new ContentValues();
  values.put(CLASS, newAccount.getClass().getName());
  values.put(LABEL, newAccount.getLabel());
  values.put(USERNAME, newAccount.getUsername());
  values.put(PASSWORD, newAccount.getPassword());

  SQLiteDatabase<D<U, L, D>> db = new SQLiteDatabase();
  db = dbOpenHelper.getWritableDatabase();

  String<shared> NullColumnHack = "";

  db.insert(TABLE_NAME, NullColumnHack, values);
  db.close();
}
```

**Figure 4.10:** Confidential unencrypted information is passed to SQLiteDatabase object.

Before implementing the constraint, I manually verify the vulnerability related to the SQLitedatabase. I check the vulnerability using an Android device, a device emulator and DDMS in Eclipse. I explore the data folders, copy the database of Ermete SMS and query through the tables to extract sensitive information (e.g., User Name, Password, etc.).

### Implementation

I draw the partial OGraph to depict the relevant objects and edges of the constraint. In the OGraph, the top-level domain UI contains the object AMA:AccountModifyActivity, LOGIC contains account:AccountManagerAndroid and DATA contains the database object db:SQLiteDatabase. The confidential information flows from the UI to the DATA domain without going through an encryption object. account:AccountManagerAndroid imports (blue edge) the account object (oldAccount:Account), extracts the information and wraps into the

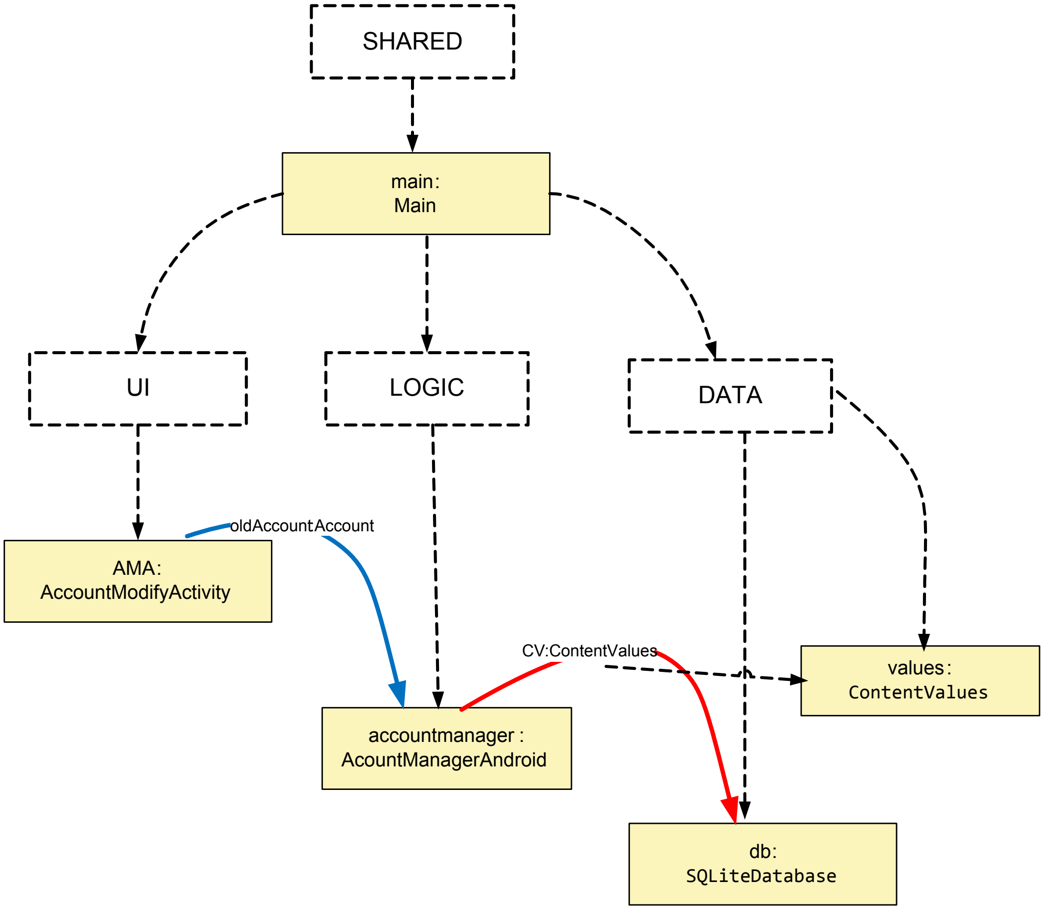


**Figure 4.11:** Vulnerability regarding SQLiteDatabase (partial OGraph)

`cv:ContentValues` object and exports (red object) to the `db:SQLiteDatabase` (Fig. 4.11).

### Result

I implement the constraint using the object transitivity and indirect communication features of Scoria. In object transitivity, all dataflow edges in a path refer to the same flow object. Although, the information of the account object is extracted and wrapped into another object in the middle of the path, the information remains the same (unencrypted) on the entire path. I verify if there is any suspicious edge exists from `AMA:AccountModifyActivity` to `AccountManagerAndroid` and if such a suspicious edge contains a flow object of type `Account`, I further check if any transitive flow exists from AccountManagerAndroid to `SQliteDatabase` that contains a

```
Suspicious edge : AMA:AccountModifyActivity -> AMAnd:AccountManagerAndroid [t:Tim]
AccountModifyActivity InsertAccountObject accountManager.insert(acc)
AccountModifyActivity DeleteAccountObject accountManager.delete(acc)

Suspicious edge : AMAnd:AccountManagerAndroid -> db:SQLiteDatabase [=?:String]
AccountManagerAndroid insert db.insert(TABLE_NAME,NullColumnHack,values)
AccountManagerAndroid delete db.delete(TABLE_NAME,whereClause,whereArgs)
```

**Figure 4.12:** Suspicious edges identified by Scoria.

flow object of type `ContentValues`.

The result shows that Scoria detects the suspicious edges (Fig. 4.12) along with a false positive. Scoria detects an additional communication due to the delete operation. This communication is false positive since it does not store any sensitive information to the SqliteDB.

## 4.8 AF-index of the vulnerabilities

The weighted sum of the Scoria features of Constraint-1 is 4 that include the object transitivity and adding security properties. Constraint-2 has a weighted sum of value 5 that use object transitivity and indirect communication. In both cases, the AF-index values convey that the vulnerabilities are more likely to be architectural flaws than coding bug.

## 4.9 Conclusion

In Ermete SMS, the first constraint can be considered to be a common architectural flaw whereas the second constraint points to an important aspect of the Android software development. Most developers are not concerned about the security issues (regarding destination) while passing sensitive data as an argument. If the `UI` layer is not supposed to show the confidential information, only non-confidential information should be sent. Secondly, SQLitedatabase is not a trusted source to store sensitive unencrypted information. An attacker can get access to the database and query the information using a number of available tools. Scoria detects both of these constraints successfully and reports suspicious edges along with a few false positives.

# CHAPTER 5: WEBGOAT

This chapter gives some overview of the third subject system, its architecture, its vulnerabilities and three constraints.

## 5.1 WebGoat Overview

WebGoat deliberately highlights security vulnerabilities in web applications. It is designed to illustrate the security flaws and provides a platform for the software testers to test an application in the context of Application Security Assessment. WebGoat is developed by Open Web Application Security Project (OWASP) group.

WebGoat contains a number of categories where each category may have multiple lessons. Each lesson may include more than one task where each task is related to a vulnerability. In some cases, multiple lessons create one vulnerability as a whole. A WebGoat user has to demonstrate the understanding of a vulnerability by completing each task.

## 5.2 System Architecture

I run the code statistics of WebGoat (using Metrics) and it shows that the system has 25 KLOC. I inspect the code and study the documentations. WebGoat is a Java-based web application developed using Model-View-Controller (MVC) pattern. In the controller, there is a generic HammerHead controller class that extends from HttpServlet base class which is further extended by other controller classes. MODEL contains the ActionHandler classes and a webSession class. ActionHandler classes are the lesson objects of a category. MODEL also contains the data objects responsible to push and pull data from and to the database. WebSession class is a middle layer class that communicates with both MODEL and VIEW objects. An architectural

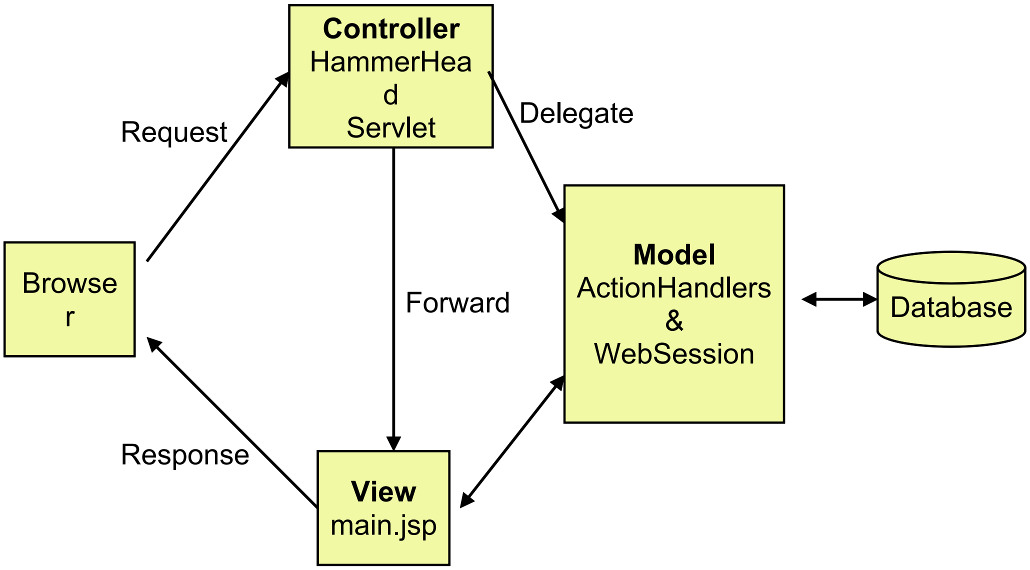


**Figure 5.1:** WebGoat architecture

overview diagram is downloaded from the official website of OWASP (Fig. 5.1).

### 5.2.1 DFD

I draw a simplified version of the DFD to check the conformance with the extracted OOG. Since WebGoat is a big application, in these two diagrams, I emphasize on the objects and the dataflow edges that are part of the constraints. The DFD (Fig. 5.2) shows three architectural tiers: MODEL, VIEW and CONTROLLER. Each object is considered as a single process of the DFD. Following are the components of the DFD:

1. View: This layer consists of two types of objects of the presentation layer such as UI (HTML) objects and view objects that communicates with the controller objects and UI objects. For instance, Input, TD and PRE are the HTML objects and `HiddenFieldTampering`, `LogSpoofing` and `PathBasedAccessControl` are the corresponding view objects.

2. Controller: The Controller layer is composed of the controller objects including the *Websession* that is responsible to maintain the web sessions. Each lesson object in the MODEL layer implements the *createContent* method that takes

the websession object as a parameter and creates the content of the webpage. `ParameterParser` takes the parameters from the HTTP requests, parse the parameters in each method implementation. The `CreateDB` object creates the database, relevant database tables and inserts the default values to the tables in order to initialize the database. `SoapRequest` implements the methods of `LessonAdapter` class including *getFirstName*, *getLastName* and *getLoggedInCount*.

3. Model: This layer consists of the business objects and other data objects (utilities) that communicate with the database. As an example, in the DFD, the `Course` is a model that maintains the learning progress of the user of each lesson. The `Encoding` object encodes and decodes sensitive data. The `DatabaseUtilities` class performs the database related utility actions, i.e., get database connection, make connection etc. The `StringElement` is a wrapper class and used to wrap multiple `String` objects.

## 5.3 Annotation Process

I use the ArchDefault tool in order to add default annotations and reduce the overhead of manually annotating the entire system.

### 5.3.1 Design Intent

The design intent of the WebGoat is State-Logic-Display (can be represented by the `DATA-LOGIC-UI` domains) that is identical to the three tier architecture of the system. State is the MODEL, Logic is the CONTROLLER and Display is the VIEW layer. While adding annotations, alongside three top level domains, I use the `shared` domain to annotate static objects, the `unique` domain to annotate objects of unshared reference and the `lent` to annotate the objects inside of a method scope.

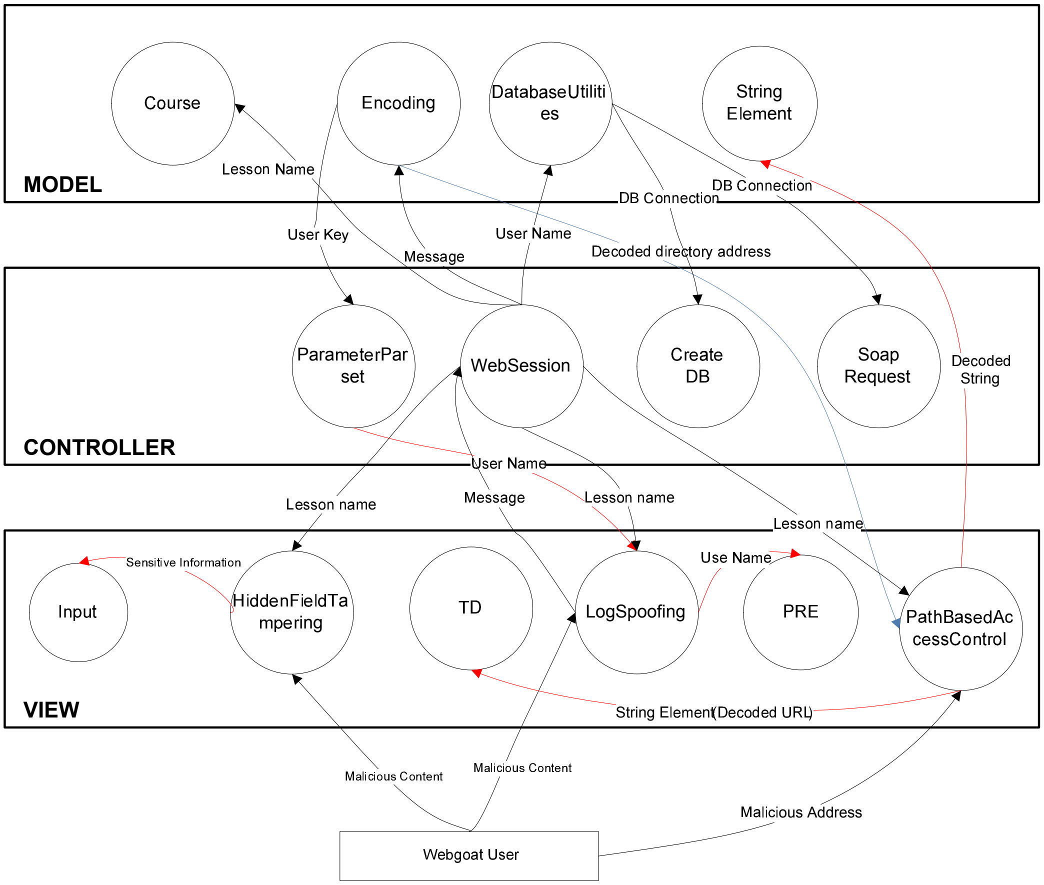


**Figure 5.2:** WebGoat DFD

### 5.3.2 Default Annotations

I use ArchDefault tool to add the default annotations. ArchDefault generates the starter map, and I make necessary changes in the map. I specify the domains, formal domain parameters and unique identifiers for each row in the map. I also set *TakeDomainParams* and *PropagateToSubType* properties as true in order to allow domain parameters to propagate to the subtypes. I validate the map and run the tool to add default annotations. The default tool adds most of the annotations and after typechecking the annotations, the number of warnings are reduced to around 9K and require further manual annotations.

```
class DOMInjection<U, L, D> extends LessonAdapter<U, L, D> {
 static Integer DEFAULT_RANKING<shared> = new Integer(10);
 String<shared> KEY = "key";
 A<shared<U, L, D>> a = new A();
 String<D> key = new String("K1JFWP8BSO8HI52LNPQS8F5L01N");

 Con<D> getCon(String<lent> user, WebgoatContext<L<U, L, D>> context) {
  Con<D> conn = connections.get(user);
  if (conn != null && !conn.isClosed())
  return conn;
  conn = makeConnection(user, context);
  connections.put(user, conn);

              if (dbBuilt.get(user) == null){
                      CreateDB<lent<U, L, D>> createDB = new CreateDB();
                      createDB.makeDB(conn);
                      dbBuilt.put(user, Boolean.TRUE);
              }
              return conn;
       }

Element<U<U, L, D>> createContent(WebSession<L<U,L,D>> s){
       ElementContainer <U<U, L, D>> ec = new ElementContainer();
    String[]<shared[shared]> allowedSelect = new String[]{"foo", "bar"};
 }
```

**Figure 5.3:** WebGoat manual annotations

### 5.3.3 Root Class and Manual Annotations

All the `String` objects are annotated in the `shared` domain by the tool, however, I realize that I need to place some objects of interest into distinct domains. Besides, ArchDefault misses adding annotations for complex expressions (e.g., anonymous object creation inside method invocations). Moreover, typechecker shows warnings for statically initialized variables. To address these issues, I address the warnings by priority. In some cases, I modify the code to add/modify annotations. Fig. 5.3 shows some examples' code changes and modification of default annotations.

The first code snippet shows that I modify the annotation of interesting `String` object ("key") by assigning it to the `DATA` domain although the default tool annotated it in the `Shared` domain. The second code snippet shows the refactoring of the code and adding corresponding annotations of the new instance of *CreateDB* to address

```
@Domains({"UI", "LOGIC", "DATA"})
class WebGoatAnalysisMain {
public void Run() {
 instantiateMissingObjects();
 }
public void instantiateMissingObjects() {
 SqlModifyData<UI<UI, LOGIC, DATA>> sqlModifyData = new SqlModifyData();
 LogSpoofing<UI<UI, LOGIC, DATA>> logSpoofing = new LogSpoofing();
 WebSessionLOGIC<UI, LOGIC, DATA> wSession = new WebSession(new
 WebgoatContext(new HttpServlet() { }), new Servlet().getServletContext());
 ForgotPassword<DATA<UI, LOGIC, DATA>> forgotPassword = new ForgotPassword();
 ParameterParser<LOGIC<UI, LOGIC, DATA>> parameterParser =
  wSession.getParser();
 DatabaseUtilities<DATA<UI, LOGIC, DATA>> dbUtilities = new
  DatabaseUtilities();
 }
static void main(String[]<lent[shared]> args) {
 WebGoatAnalysisMain<shared> WGAMain = new WebGoatAnalysisMain();
 WGAMain.instantiateMissingObjects();
 }
}
```

**Figure 5.4:** WebGoat root class

the typechecker warning of anonymous object creation. The third code snippet shows the modification of the code to address the annotation error due to static initialization of the String[]. I address most of the annotation warnings and reduce the number of warnings from around 9K to around 1K.

I use the ArchMetrics tool to get the list of uninitialized types. I create a root class (WebGoatAnalysisMain), annotate three top-level domains in it and initialize the un-initialized types. A portion of the root class is shown in (Fig. 5.4).

## 5.4 Object Graph Extraction

I extract both the OOG and the OGraph, however, like other systems, both of these graphs are large enough to visualize the entire set of OObjects and follow the dataflow edges. The unexpanded view of the OOG is shown in (Fig. 5.5).

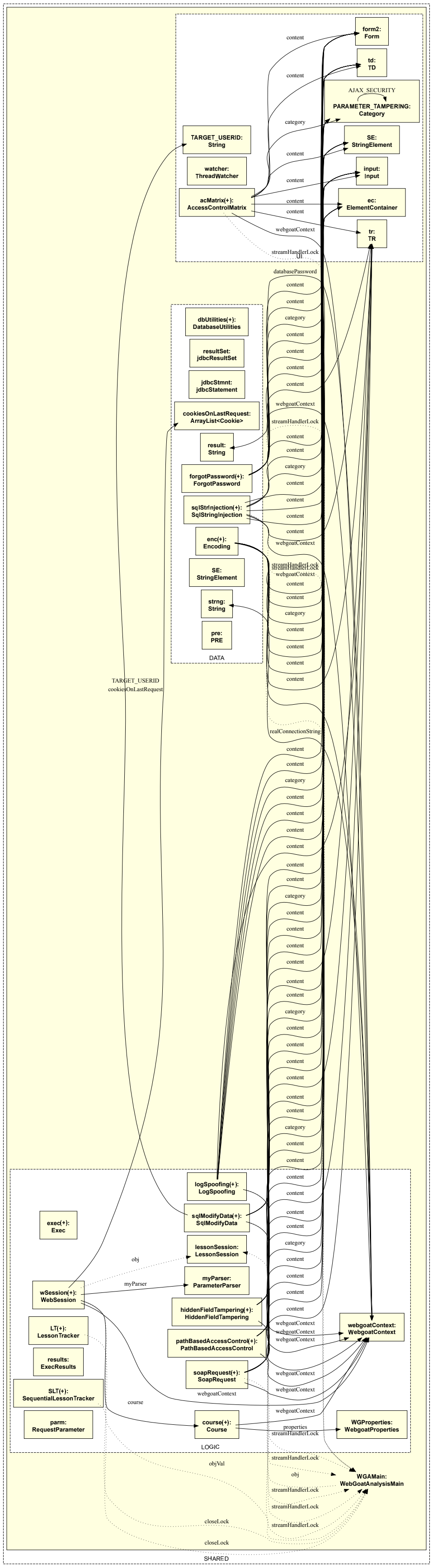


**Figure 5.5:** WebGoat Ownership Object Graph (collapsed view)

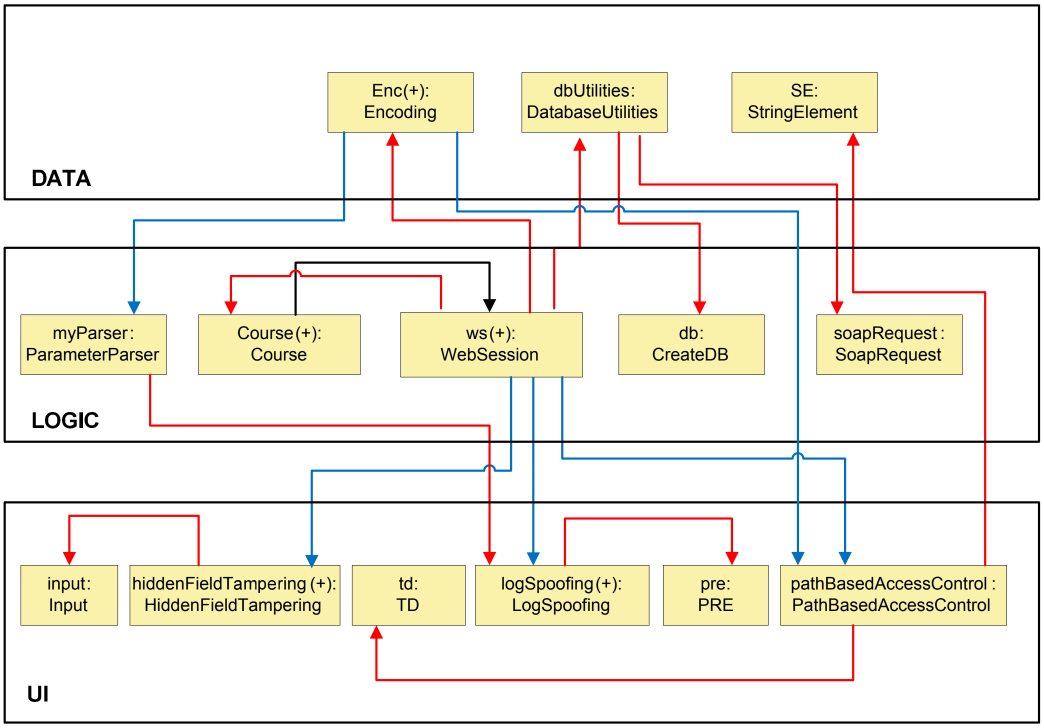

DATA
Enc(+): Encoding
dbUtilities: DatabaseUtilities
SE: StringElement
LOGIC
myParser: ParameterParser
Course(+): Course
ws(+): WebSession
db: CreateDB
soapRequest: SoapRequest
UI
input: Input
hiddenFieldTampering(+): HiddenFieldTampering
td: TD
logSpoofing(+): LogSpoofing
pre: PRE
pathBasedAccessControl: PathBasedAccessControl

**Figure 5.6:** OOG conformance diagram

## 5.5 Conformance with the DFD

The conformance diagram (Fig. 5.6) has three tiers representing the three top level domains: UI, LOGIC and DATA. Each domain contains a number of objects and relevant communication edges: the blue edge represents an import edge, the red edge shows an export edge and the black arrow represents a field reference. To observe an example of the conformance with the DFD (Fig. 5.2), it can be seen that `PathBasedAccessControl` process sends a *decoded URL* object to the *UI* element `TD`. Analogously, there is an export communication edge from `pathBasedAccessControl:PathBasedAccessControl` to `td:TD`.

## 5.6 Refinement of the Annotations

I refine the annotations to reflect my design intent and to prevent the objects of interest from getting merged. An OOG is inconsistent with the design intent if it

```
Element<U<U,L,D>> doStage6(WebSession<L<U,L,D>> s){
 StringElement<U<U,L,D>> SE = new StringElement("not yet");
 return SE;
}
Element<U<U,L,D>> createContent(WebSession<L<U,L,D>> s) {
 String<D> listing = " <p><B>"+getLabelManager().get("CurrentDirectory")+"</B>
  " + enc.urlDecode(dir) +
  "<br><br>"+getLabelManager().get("ChooseFileToView")+"</p>";
 StringElement<D> SE = new StringElement();
 SE.addElement(listing);
 td.addElement(SE);
}
```

**Figure 5.7:** Refinement of annotations (preventing objects from excessive merging)

misses some desired objects and edges due to incorrect annotations. As the type-checker cannot recognize anonymously instantiated objects, those objects are missing in the OGraph. Hence, code refactoring is needed. Examples are shown in Fig. 5.3. In order to prevent some desired objects from being merged, I annotate those objects in distinct domains. For instance, one particular instance of `StringElement` object is annotated in the `DATA` domain while regular `StringElement` objects are not significant and annotated in the `UI` domain (Fig.5.7). This is because `listing:String` contains the sensitive data which is wrapped by `SE:StringElement` and thus, the wrapper object becomes significant.

## 5.7 Constraints:

WebGoat contains around 66 injected vulnerabilities divided into 20 categories. I hand select three vulnerabilities from three categories: Access Control Flaws, Injection Flaws and Parameter Tampering. I implement three constraints of three potential architectural flaws in WebGoat to evaluate if Scoria can detect these vulnerabilities.

### 5.7.1 Constraint-1: Vulnerability regarding path based access control

Many web applications allow the users to access specific files from the disk rather than making a copy of the file in the application's temporary directory. Sometimes, the directory address of the file is stored in a variable or into a hidden field that can be exploited by an attacker to understand the file system and get access to other files.

The lesson `Path Based Access Control` lists a number of files in a dropdown menu to which the user is allowed to access. However, the corresponding disk location is also stored in a variable. A user can copy the address in a browser, get the list of available files in the directory and get access to other files.

To understand this vulnerability from the code implementation of the WebGoat (Fig. 5.8), it can be noticed that `enc:Encoding` object sends the decoded url (`listing:String`) to `PathBasedAccessControl`. Later, `PathBasedAccessControl` wraps the flow object into `SE:StringElement`. Finally, the confidential object `SE:StringElement` is sent to UI element `td:TD` that shows the decoded url to the front end screen.

#### Implementation

Partial OGraph shows the interacting objects of this vulnerability (Fig. 5.9). An import edge from `enc:Encoding` object shows that `pathBasedAccessControl:PathBasedAccessControl` is receiving the decoded url object, `listing:String`. It also shows that `pathBasedAccessControl:PathBasedAccessControl` exports the `SE:StringElement` object to the UI element `td:TD`.

```
class PathBasedAccessControl<U, L, D> extends LessonAdapter<U, L, D>{

Element<U<U,L,D>> createContent(WebSession<L<U,L,D>> s){

        Encoding<unique<U,L,D>> enc = new Encoding();
        TD<U<U,L,D>> td = new TD();
    try
    {
        String<D> dir = s.getContext().getRealPath("/lesson_plans/en");
        String<D> listing = " <p><B>"+getLabelManager().get("CurrentDirectory")+"</B>
        " + enc.urlDecode(dir) + "<br><br>"+getLabelManager().get("ChooseFileToView")+"</p>";
    StringElement<D> SE = new StringElement();
    SE.addElement(listing);
    td.addElement(SE);
      }
   }
 }
```

**Figure 5.8:** `SE:StringElment` is sent to `td:TD`

### Result

I implement the constraint using the object provenance and object transitivity features of Scoria. In this constraint, the flow object contains sensitive information and thus, it is set as confidential. On the other hand, the sink `ada:AccountDisplayActivity` is considered as untrusted. Scoria successfully identifies the suspicious edge and reports it in the console output of eclipse (Fig. 5.10).

## 5.7.2 Constraint-2: Log Spoofing

One of the common vulnerabilities of web applications is the spoofing of the log file. A web application can store each user interaction's success/failure message into the system log. A common example of the logging event is the user authentication system that stores the time of user's signing in and signing out information along with a message. While creating the message, a developer can use the content of the input boxes to dynamically create the message content. Hence, an attacker with keen HTML knowledge can exploit the HTML input box to manipulate the content of the

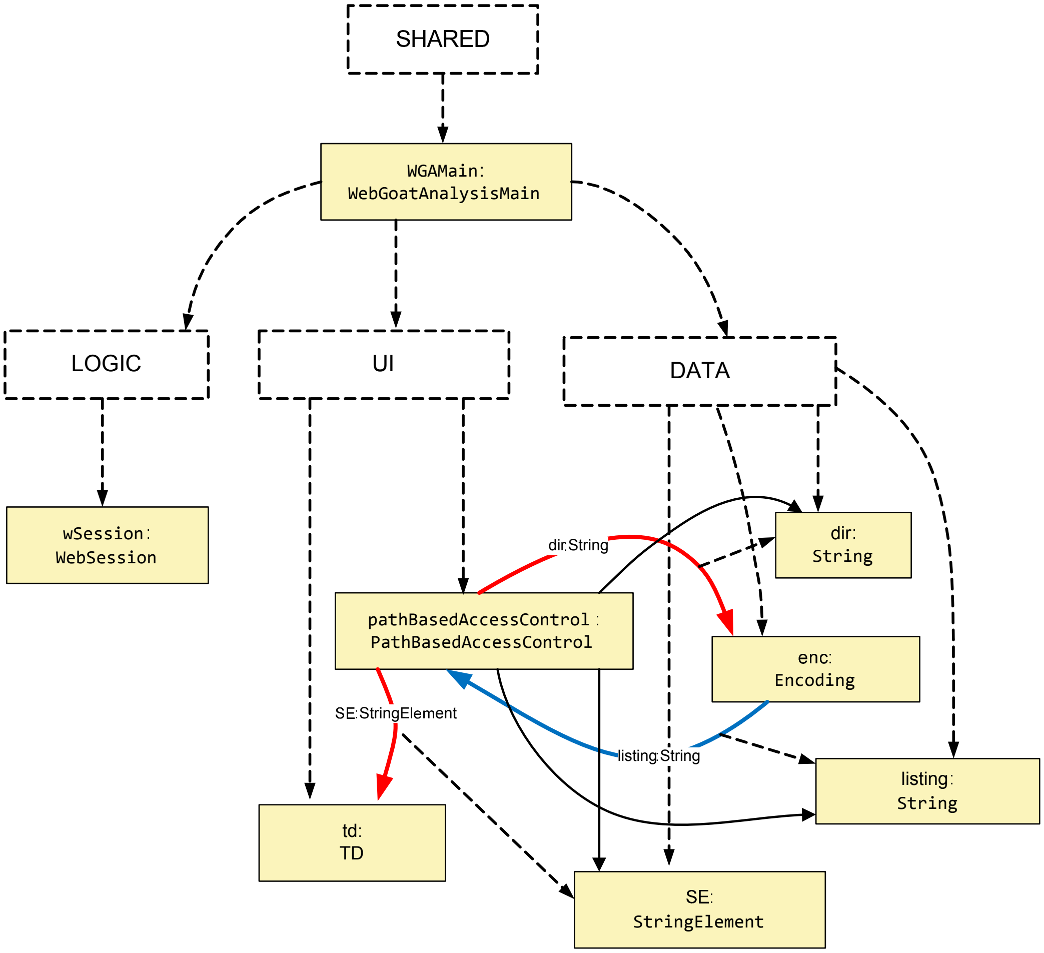


**Figure 5.9:** `pathBasedAccessControl:PathBasedAccessControl` exports `SE:StringElement` to `td:TD`

```
Information disclosure exists.
Suspicious edge : pathBasedAccessControl:PathBasedAccessControl -> td:TD [SE:StringElement]
       PathBasedAccessControl createContent td.addElement(SE)

Suspicious edge : pathBasedAccessControl:PathBasedAccessControl -> td:TD [SE:StringElement]
       PathBasedAccessControl createContent td.addElement(SE)
```

**Figure 5.10:** Suspicious edges identified by Scoria

message and thus, can misguide the system administrator when she checks the logged messages to trace a system failure.

Code inspection shows that `ParameterParser` object sends the content of the "UserName" input field to the `LogSpoofing` object which in turn sends the `inputUserName:String` to an UI element `pre:PRE`. In WebGoat, `PRE` is an UI el-

```
class LogSpoofing<U, L, D> extends LessonAdapter<U, L, D>{

       Element<U<U,L,D>> createContent(@Domain("L<U,L,D>") WebSession<L<U,L,D>> s){
              String<U> inputUsername = s.getParser().getRawParameter(USERNAME, "");

              // Representative of Log. File: 'PRE'
              PRE<D> pre = new PRE(getLabelManager().get("LoginFailedForUserName")+": " +
              inputUsername);
              pre.addElement(inputUsername);
        }
}
```

**Figure 5.11:** `inputUsername:String` is sent to `pre:PRE`

ement that shows the modified logged information and acts as a representative of the log file (Fig. 5.11).

### Implementation

(Fig. 5.12) illustrates the partial OGraph of the Log Spoofing and shows a transitive dataflow of the confidential object `inputUsername:String` from `myParser:ParameterParser` to the destination object `pre:PRE`.

### Result

I implement this constraint using the object transitivity and object provenance features of Scoria and Scoria successfully identifies the suspicious edges due to the import and export of the confidential object `inputUsername:String` (Fig. 5.13).

## 5.7.3 Constraint-3: Vulnerability regarding hidden field tampering

The use of a hidden field for storing sensitive information temporarily is a common practice in web development. However, attackers can be intelligent enough to quickly verify if an web application uses a hidden field to store any information. Once such

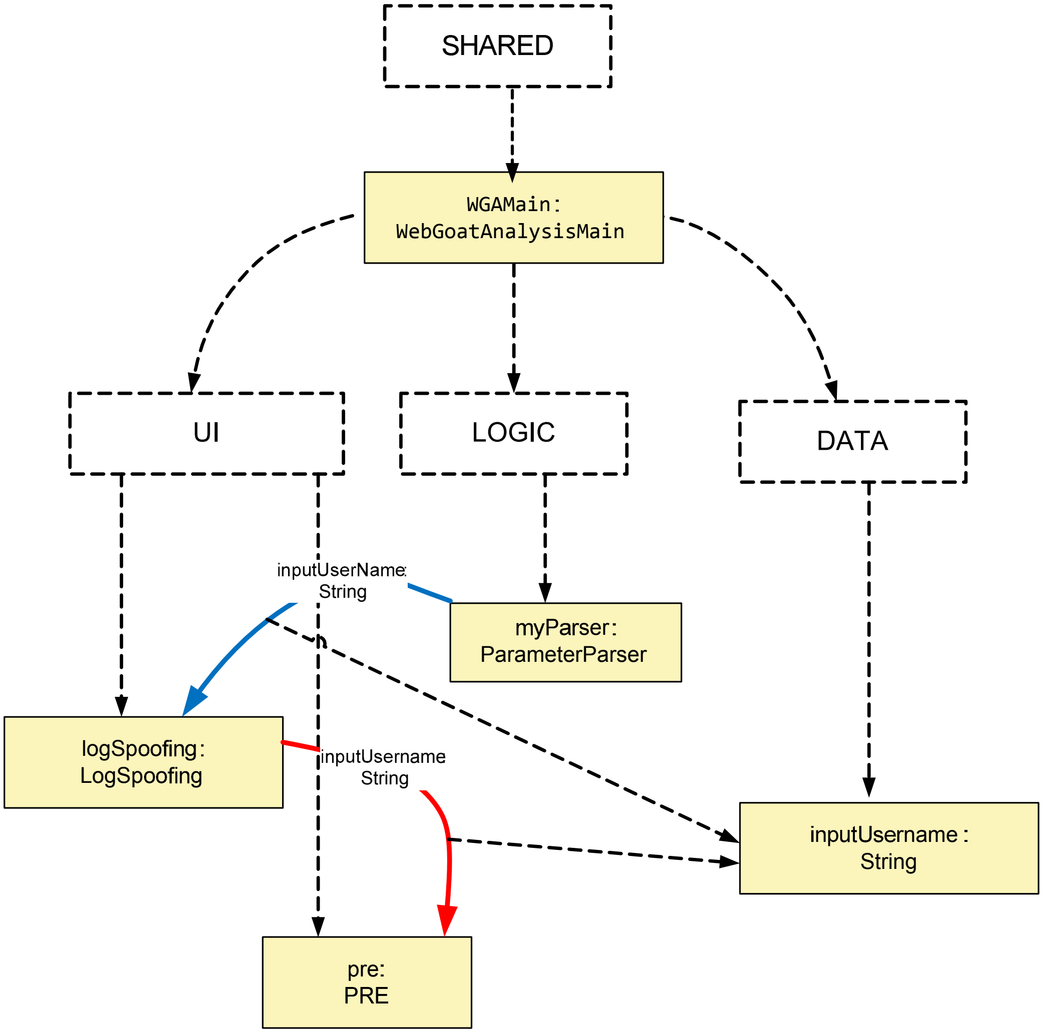


**Figure 5.12:** `logSpoofing:LogSpoofing` exports `inputUsername:String` to `pre:PRE`

```
Suspicious Edge exporting malicious object to Log:
Suspicious edge : myParser:ParameterParser -> logSpoofing:LogSpoofing [:String]
       LogSpoofing createContent s.getParser().getRawParameter(USERNAME,"")

Suspicious edge : logSpoofing:LogSpoofing -> pre:PRE [inputUsername:String]
       LogSpoofing createContent pre.addElement(inputUsername)
```

**Figure 5.13:** Suspicious edges identified by Scoria

a vulnerability is found, they can tamper the content of the hidden field to commit a potential harm to the application. In WebGoat, a hidden field is used to store a cumulative figure of money transactions of an e-commerce shopping cart. An attacker, who intends to buy a list of items of $3000, can send a cumulative figure of $1000

```
class HiddenFieldTampering<U, L, D> extends LessonAdapter<U, L, D>{

String<D> PRICE_TV = new String("2999.99");

Element<U<U,L,D>> createContent(WebSession<L<U,L,D>> s){

       float quantity, total;
              String<lent> price = PRICE_TV;
              DecimalFormat<lent> money = new DecimalFormat("$0.00");
       try{
           price = s.getParser().getRawParameter(PRICE, PRICE_TV);
           quantity = s.getParser().getFloatParameter("QTY", 1.0f);
           total = quantity * Float.parseFloat(price);

       } catch (Exception<lent> e){

           s.setMessage(getLabelManager().get("Invaild data") + this.getClass().getName());
           price = PRICE_TV;
           quantity = 1.0f;
           total = quantity * Float.parseFloat(PRICE_TV);
       }
       if (price.equals(PRICE_TV)) {
                     Input<U<U,L,D>> input = new Input(Input.HIDDEN, PRICE, PRICE_TV);
           input.addElement(PRICE_TV);
       }
    }
}
```

**Figure 5.14:** PRICE_TV is send to the input element

to the server while checking out the items by tampering the hidden field. Thus, a hidden field should not be used as a temporary storage of the sensitive information.

The code-view (Fig. 5.14) of the `HiddenFieldTampering` class shows that `hiddenFieldTampering:HiddenFieldTampering` sends sensitive data (*PRICE_TV*) to the hidden type input object (`input:Input`).

### Implementation

Partial OGraph shows the insecure dataflow (export edge) to the `input:Input` from `hiddenFieldTampering:HiddenFieldTampering` (Fig. 5.15). `PRICE_TV:String` is the confidential flow object that is tampered by the attacker when tv price is stored in the hidden field.

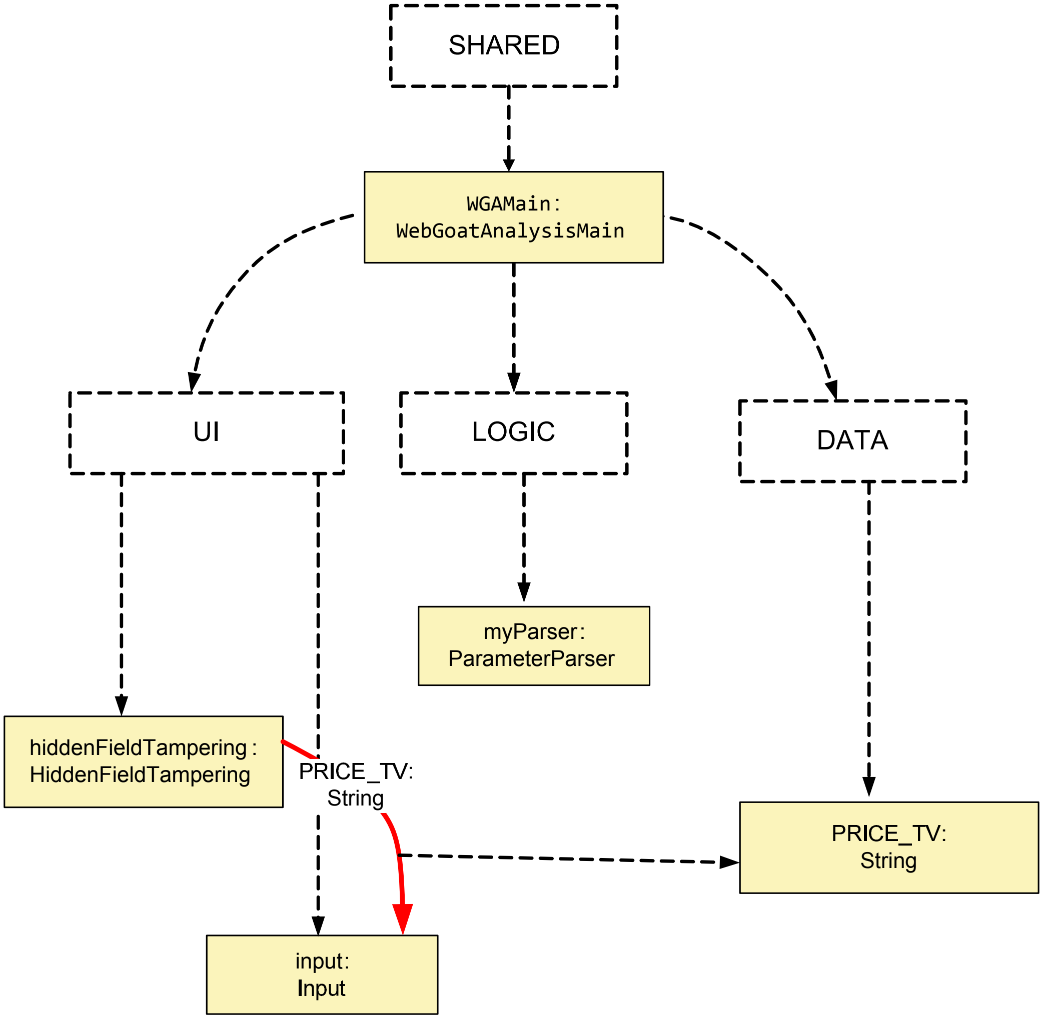


**Figure 5.15:** Vulnerability regarding hidden field tampering (partial OGraph)

```
Vulnerability found regarding Tampering of Hidden Input Field.
Suspicious edge : hiddenFieldTampering:HiddenFieldTampering -> input:Input [PRICE_TV:String]
       HiddenFieldTampering createContent input.addElement(PRICE_TV)
```

**Figure 5.16:** Suspicious edges identified by Scoria

### Result

I implement the constraint using the object transitivity feature of Scoria. I also add two security properties. Scoria finds the suspicious communication edges as shown in the output (Fig. 5.16).

## 5.8 AF-index of the vulnerabilities

Constraint-1 and Constraint-2 have a weighted sum of 5 due to the object transitivity and object provenance features of Scoria. Constraint-3 uses object transitivity and security properties and has an index value of 4. The AF-index of these constraints represents significance of the vulnerabilities in the continuum of architectural flaws.

## 5.9 Conclusion

These three selected vulnerabilities are important and significant in the context of architectural flaws in web applications. Scoria identifies all the three vulnerabilities (mentioned above) without generating any false positive.

**Table 6.1:** AF-indexes of the constraints

| Constraint | Obj. Provenance. (3) | Obj. Transitivity. (2) | Ind. Comm. (3) | Sec. Prop. (1) | AF-Index (1-10) |
|---|---|---|---|---|---|
| Shared Pref. | ✓ | ✓ | | | 5 |
| Android Log | ✓ | ✓ | | | 5 |
| Inf. Disclosure | | ✓ | | ✓(2) | 4 |
| SQLite DB | | ✓ | ✓ | | 5 |
| Path-based Acc. Ctrl. | ✓ | ✓ | | | 5 |
| Log Spoofing | ✓ | ✓ | | | 5 |
| Hidden Fld. Tampr. | | ✓ | | ✓(2) | 4 |

# CHAPTER 6: DISCUSSION

This chapter summarizes the AF-index of the vulnerabilities, discusses the precision of the results, estimated efforts of the subject systems and threats to validity of the approach.

## 6.1 AF-indexes of the vulnerabilities

I summarize the AF-indexes of all the vulnerabilities of the three subject systems. Table 6.1 shows that the vulnerabilities regarding Android sharedpreference and Android log file have an AF-index value of 5. The constraint Information Disclosure has a value of 4 and SQLiteDB has an AF-index value of 5. Path-based Access Control and Log Spoofing have a value of 5 and Hidden Field Tampering has a value of 4. Since the AF-indexes of all the constraints are in the middle of the AF-index scale, the vulnerabilities are closer to being architectural flaws than coding bugs.

## 6.2 Precision of the results

Figure 6.2 shows the precision of the results in terms of the TP (True Positive), FP (False Positive), TN (True Negative) and FN (False Negative). A TP defines that Scoria detects the vulnerability, FP defines that Scoria reports a false vulnerability, TN is the true reporting when the vulnerability is absent in the system and FN is

**Table 6.2:** Accuracy of the result

| System | KLOC | No. of Constraints | TP | TN | FP | FN |
|---|---|---|---|---|---|---|
| Muspy | 6 | 2 | 2 | 1 | 0 | 0 |
| Ermete SMS | 6 | 2 | 2 | 0 | 2 | 0 |
| WebGoat | 25 | 3 | 3 | 0 | 0 | 0 |

defined by not reporting of the vulnerability when it exists. We compute the Precision as TP/(TP+FP) and it is around 80%.

## 6.3 Scoria detects Common Android Architectural Flaws

From the analyses and results, I observe two common architectural flaws in Android application development. First, SharedPreference should not be considered as trusted data storage although SharedPreference storage is application independent in an Android device. Second, SQLiteDB also cannot be a trusted source for storing data since the database is accessible through DDMS. Scoria analysis detects both of these vulnerabilities and shows that both Muspy and Ermete SMS is vulnerable as they do not encrypt the sensitive data before storing in the SharedPreference and SQLiteDB.

## 6.4 Scoria detects Common Webapplication Architectural Flaws

Scoria detects three important architectural flaws of a web application; path-based access control, log spoofing and hidden field tampering. All of the three vulnerabilities are significant in web applications.

## 6.5 Impact of Code implementation over Scoria approach

Refinement of the annotations is impacted by different approaches of coding practices of different subject systems. The typechecker shows warnings if it cannot handle

**Table 6.3:** Estimated Effort of Muspy

| **Phase** | **Estimated Effort** | **Percent %** |
|---|---|---|
| Annotations and object graph extraction | 13 hours | 46.5 |
| Extraction | 5 minutes | less than 1 |
| DFD and OOG conformance | 4 hours | 14 |
| Refine object graph | 4 hours | 14 |
| Writing constraints | 7 hours | 25.4 |
| Total | 28 hours | 100% |

**Table 6.4:** Estimated Effort of Ermete SMS

| **Phase** | **Estimated Effort** | **Percent %** |
|---|---|---|
| Annotations and object graph extraction | 15 hours | 56 |
| Extraction | 3 minutes | less than 1 |
| DFD and OOG conformance | 4 hours | 15 |
| Refine object graph | 3 hours | 11 |
| Writing constraints | 5 hours | 18 |
| Total | 28 hours | 100% |

some code constructs while checking the consistency of the annotations with the code. To address these warnings, code refactoring may be needed. For example, in some coding practice, an anonymous object can be created rather than creating an actual instance of a type. However, the typechecker expects a named instance of a type.

## 6.6 Estimated efforts of finding security vulnerabilities

The estimated effort of the three subject systems for finding security vulnerabilities are shown in Tables 6.3, 6.4 and 6.5. The efforts are computed in terms of Phase, Effort and Percentage of the effort of each phase. The tables show that estimated effort is higher in WebGoat since it is larger in size (KLOC) than other two systems whereas Muspy and Ermete SMS took similar effort.

## 6.7 Threats to Validity

I inject a vulnerability since the subject system Muspy does not contain any vlunerability regarding Android log. Despite the vulnerability being injected, it allows

**Table 6.5:** Estimated Effort of WebGoat

| **Phase** | **Estimated Effort** | **Percent %** |
|---|---|---|
| Annotations and object graph extraction | 40 hours | 60 |
| Extraction | 6 minutes | less than 1 |
| DFD and OOG conformance | 7 hours | 10 |
| Refine object graph | 3 hours | 12 |
| Writing constraints | 12 hours | 18 |
| Total | 67 hours | 100% |

us to understand that Scoria could detect if there is a vulnerability of the Android log. Secondly, the design intent of the security architect can have a significant impact over the approach. The security architect may not be the system architect. Moreover, if the documentation of the system is not sufficient, the security architect may choose different annotations that lead Scoria to extract different OOGs and OGraphs. Such an OGraph may contain objects in unexpected domains. As a result, while writing constraints, security architect may not find the significant vulnerabilities. Also, the analysis may generate more false positives.

# CHAPTER 7: RELATED WORK

This chapter discusses related work and comparative studies of the evaluation of different tools that find security vulnerabilities.

## 7.1 Evaluation of Static Analysis Tools

I divide the related work that highlight the evaluation of static analysis tools into ones that focus on coding bugs and those that focus on architectural flaws.

### 7.1.1 Tools that Focus on Coding Bugs

Nathaniel et al. evaluate most downloaded open source static analysis tool named FindBugs. FindBugs detects common coding bugs, e.g., null pointer dereferencing and overflow of an array .[14] According to the authors, Google used FindBugs and identified 70 null pointer errors in their code. While evaluating FindBugs, alongside positive testing, the authors also perform negative testing for which FindBugs cannot report all the vulnerabilities. Moreover, the authors find that FindBugs misses null pointer dereferencing errors if a particular path of the program is executed. The authors reason that FindBugs does not compute the feasibility of the path.

Nathan et al. evaluate a vulnerability analysis framework named MINESTRONE that detects coding bugs especially memory corruption, null pointer dereferencing, resource drain and number handling errors .[15] The authors explain the four components of the MINESTRONE: DYBOC (detects buffer overflows and underflows), REASSURE (for error recovery mechanism), ResMon and IOC number handling. ResMon detects the vulnerabilities related to the resource leakage. IOC number handling identifies the incorrect numeric value handling errors (e.g., unsafe unsigned-to-signed conversion). According to the authors, MINESTRONE is able to analyze large projects up to 200 KLOC. The authors create two test suites each of which consists of a number of test cases. The first suite contains 340 hand written vulnerable programs and second suite contains a number of open source projects. Each of the test cases consists of a good and bad I/O pairs. The analysis is marked as passed for one test case if it passes for both the I/O pairs. The authors show that MINESTRONE reports 80% of the errors effectively.

Mamun et al. publish a paper on comparative evaluation of four static analysis tools (Coverity Prevent, Jtest, FindBugs and JLint) that find concurrency bugs. The authors aim to find if the commercial tools are better performed than the open source tools.[16] They discuss the working mechanism of each of the tools and different types of concurrency bugs and patterns. According to the authors, non-determinism of the concurrency issues trigger a number of bugs including data races, atomicity violations, synchronization defects, deadlocks and livelocks. The authors use an existing benchmark of 87 unique Java concurrency bug patterns. The results show that the commercial tool JTest performs better in detecting java concurrency bugs with a relatively higher number of false positives. On the other hand, Coverity Prevent detects lowest number of bugs although it produces less false positives.

Vorobyov and Krishnan evaluates two different types of tools: a Model Checking tool named CBMC and a static analysis tool named Parafit .[17] According to the authors, a model checker generates the run time states of the program and for a finite number of states a model checker performs an exhaustive analysis. If the model fails to verify a certain specified property, the result will be a failed verification. The authors evaluate the results with a set of criteria: false Positive, false Negative, execution time and resource consumed. They evaluate the tools using the test cases from three existing benchmarks: Iowa, SAMATE and Cigital. The results show that although CBMC shows greater accuracy (97%) and zero false positive, the analysis took significant amount of execution time (19 hours) and consumes significant memory (2.5 GB at the peak). The authors also conclude that for larger systems CBMC is not quite feasible.

Gomes et al. perform an extensive study over a number of available static analysis tools to evaluate the tools and their performance in different scenarios. According to the authors, some static analysis tools are language specific and some are not. FxCop, StyleCop and CodeIt are specific to the Microsoft .Net framework whereas PMD and JLint are Java-specific. SPlint, PloySpace, CodeSonar and HP Code Advisor are specific for C and C++.[18] Coverity Prevent, Klockwork Insight, Hammurapi, RATS and Understand support C,C++, C# and Java. The authors evaluate the tools with 14 different test cases collected from benchmarks of SM, BIND and WU-FTPD regarding buffer overflow where each test case contains two constraints; a “BAD” case and an “OK” case. A tool passes one test case only if it satisfies both of the constraints. The authors show that the top perform tools in terms of generating true positives is SPlint and PolySpace with a success rate of 87% and 57% respectively.

### 7.1.2 Tools that Focus on Architectural Flaws

Vanciu et al. perform a comparative evaluation of a code level approach, FlowDroid, and an architectural level approach, Scoria. Although these two approaches are different, the evaluation focuses on the precision and recall[11] of the results. The authors explain the Scoria approach and the steps that a Security Information Worker (SIW) is required to follow to perform analysis using Scoria. The authors also explain different code level approaches such as Fortify and AppScan. According to the authors, Scoria uses an object graph whereas FlowDroid uses a precomputed call graph. The authors design a few test cases and hand selects a number of test cases from existing benchmarks (DroidBench and SAMATE) and divide them into a number of equivalent classes. The authors show that FlowDroid detects the vulnerabilities that are more related to coding bugs whereas Scoria shows better precision and recall in detecting architectural flaws.

Zeineb et al. evaluate four static software analysis tools: MOPS, SPlint, GraphMatch and Fortify.[19] The authors explain the detailed methodologies of these tools. MOPS is a model checking approach, SPlint is a dataflow analysis tool, GraphMatch creates a SDG (system dependence graph) that is an extension of a program dependence graph (PDG), and Fortify uses control flow and dataflow analyses, and runs on multiple environments (Windows, Linux and Mac). While MOPS, SPlint and GraphMatch analyze C program, Fortify supports multiple programming languages. The authors show that among these tools only GraphMatch and Fortify detect security vulnerabilities for the selected test cases.

Zitsar et al. assess five static analysis tools: ARCHER, BOON, PloySpace, SPlint and UNO. They use 14 predesigned test cases regarding buffer overflows of three existing software; BIND, SendMail and WU-FTPD .[20] The authors perform two types of evaluation; ability of the tools to analyze the application as a whole, ability of detecting the buffer overflow vulnerabilities separately. The authors input the entire SendMail application (+145 KLOC) and observe that none of the tools has finished the analysis. The authors then evaluate the tools over individual test cases. Based on the results, the authors draw an ROC (Receiver Operating Characteristic) curve that shows that PloySpace is more successful in detecting the vulnerabilities as well as generating less false positives than that of other tools. PolySpace shows p(d) = 0.85 (detection of vulnerabilities) and p(f) = 0.50 (false positives) whereas the closest result to the PolySpace has been shown by SPlint with values of p(d) = 0.60, and p(f) = 0.43 respectively.

Pomorova and Ivanchyshyn perform an assessment of four commercial static analysis tools; PVS Studio, PC-Lint, Goanna Studio and Cppcheck .[21] These tools are suggested by US National Institute of Standards and Technology for static program analysis. The authors evaluate these tools using 25 different test cases of the following categories: Race Condition, Input Validation, Exception, SQL Injection, Buffer

Overflow, Stack Overflow and Integer overflow. The results show that CppCheck has higher precision and recall in combined whereas both PVS-Studio and Goanna have higher precision than CppCheck individually, however, recall is significantly lower than that of CppCheck.

## 7.2 Evaluation of Dynamic Analysis Tools

Egele et al. perform a comprehensive study of dynamic analysis tools in the context of malware detection .[22] According to the authors, the evasion techniques (self modification of the code) employed by a malicious software thwarts static analysis tools and lead to choosing dynamic analysis to find the vulnerabilities. The authors explain different categories of malwares: Worm, Virus, Trojan horse, Spyware, Bot and Rootkit. They also explain different features of malware analysis: Function Call Monitoring, Function parameter analysis, Information Flow Tracking, Instruction Trace and Autostart extensibility points. The authors discuss the techniques of addressing these features and the underlying implementation details of the tools. They also show a behavioral grouping of the tools based on the similarity of the implementation, i.e., addressing a family of malware evasion techniques.

# CHAPTER 8: CONCLUSION

This chapter discusses the contribution of the research and the prospects of the future work.

## 8.1 Contribution

The contribution of this work is twofold. First, I hypothesize that Scoria detects the architectural flaws in large applications from different application domains. I show the results and explain that Scoria analysis detects the architectural flaws in large Android applications as well as in web application with few false positives. Secondly, Scoria detects a few vulnerabilities that are both common and significant in Android application development.

## 8.2 Future Work

WebGoat has more than 60 injected vulnerabilities and I implement three constraints. More constraints can be implemented for WebGoat. Scoria can be evaluated on more desktop applications. One area of future work is the comparative evaluation with other approaches that also aims to find architectural flaws in systems.[23] Scoria can also be evaluated micro test cases relevant to web application security from the benchmark SecuriBenchMicro.[24]

# REFERENCES

[1] List of tools for static code analysis. 2015; http://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis, Software Analysis Tools.

[2] McGraw, G. *Software Security: Building Security In*; Addison-Wesley Professional, 2006.

[3] Shuo Chen, Z. K., Jun Xu; Iyer, R. K. Security Vulnerabilities: From Analysis to Detection and Masking Techniques. IEEE Intl. Working Conference on Source Code Analysis and Manipulation (SCAM). 2006.

[4] Swiderski, F.; Snyder, W. *Threat Modeling*; Microsoft Press: Redmond, WA, USA, 2004.

[5] Abi-Antoun, M.; Aldrich, J. Static Extraction of Sound Hierarchical Runtime Object Graphs. Proceedings of the 4th International Workshop on Types in Language Design and Implementation. New York, NY, USA, 2009; pp 51–64.

[6] Coverity. Static Application Security. 2013; http://www.coverity.com/security/#SAST, Coverity.

[7] Fortify Static Code Analyzer. 2014; http://www8.hp.com/us/en/software-solutions/static-code-analysis-sast/, HP Fortify.

[8] Software Security Built for Developers. 2014; http://www.cigital.com/services/code-review/secure-code-review/secureassist/, Cigital SecureAssist.

[9] Arzt, S.; Rasthofer, S.; Fritz, C.; Bodden, E.; Bartel, A.; Klein, J.; Le Traon, Y.; Octeau, D.; McDaniel, P. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. New York, NY, USA, 2014; pp 259–269.

[10] Vanciu, L. R. Static Extraction of Dataflow Communication for Security. Ph.D. thesis, Wayne State University, Detroit, MI 48201, 2014.

[11] Vanciu, R.; Khalaj, E.; Abi-Antoun, M. Comparative Evaluation of Architectural

and Code-Level Approaches for Finding Security Vulnerabilities. Proceedings of the 2014 ACM Workshop on Security Information Workers. New York, NY, USA, 2014; pp 27–34.

[12] Kumar, M. Multiple vulnerabilities in Firefox for Android Leak Sensitive Information. 2014; http://thehackernews.com/2014/03/multiple-vulnerabilities-in-firefox-for.html, Android Log Leak Firefox.

[13] Institute, I. Insecure Local Storage. 2014; http://resources.infosecinstitute.com/android-hacking-security-part-10-insecure-local-storage/, SqlliteDB tutorial hack.

[14] Ayewah, N.; Hovemeyer, D.; Morgenthaler, J. D.; Penix, J.; Pugh, W. *IEEE Software* **2008**, *25*, 22–29, Special issue on software development tools, September/October (25:5).

[15] Evans, N. S.; Benameur, A.; Elder, M. C. Large-scale Evaluation of a Vulnerability Analysis Framework. Proceedings of the 7th USENIX Conference on Cyber Security Experimentation and Test. Berkeley, CA, USA, 2014; pp 3–3.

[16] Mamun, M. A. A.; Khanam, A.; Grahn, H.; Feldt, R. Comparing Four Static Analysis Tools for Java Concurrency Bugs. Proc. of the Third Swedish Workshop on Multi-Core Computing (MCC-10). 2010; pp 143–146.

[17] K. Vorobyov, P. K. Comparing Model Checking and Static Program Analysis: A Case Study in Error Detection Approaches. 2010; pp 1–7.

[18] Gomes Ivo, G. T., Morgado Pedro; Rodrigo, M. *Faculdade de Engenharia da Universidade do Porto*

[19] Zhioua, Z.; Short, S.; Roudier, Y. Static code analysis for software security verification: Problems and approaches. STPSA 2014, 9th IEEE International Workshop on Security, Trust and Privacy for Software Applications, in COMPSAC 2014, 21-25 July 2014. 2014.

[20] Zitser, M.; Lippmann, R.; Leek, T. *SIGSOFT Softw. Eng. Notes* **2004**, *29*, 97–

106.

[21] Pomorova, O. V.; Ivanchyshyn, D. O. Assessment of the source code static analysis effectiveness for security requirements implementation into software developing process. IEEE 7th International Conference on Intelligent Data Acquisition and Advanced Computing Systems, IDAACS 2013, Berlin, Germany, September 12-14, 2013. 2013; pp 640–645.

[22] Egele, M.; Scholte, T.; Kirda, E.; Kruegel, C. *ACM Comput. Surv.* **2008**, *44*, 6:1–6:42.

[23] Almorsy, M.; Grundy, J.; Ibrahim, A. S. Automated Software Architecture Security Risk Analysis Using Formalized Signatures. Proceedings of the 2013 International Conference on Software Engineering. Piscataway, NJ, USA, 2013; pp 662–671.

[24] Johnson, A.; Waye, L.; Moore, S.; Chong, S. Exploring and Enforcing Security Guarantees via Program Dependence Graphs. Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation. New York, NY, USA, 2015; pp 291–302.

## ABSTRACT

## EVALUATION OF AN ARCHITECTURAL-LEVEL APPROACH FOR FINDING SECURITY VULNERABILITIES

by

**MOHAMMAD ANAMUL HAQUE**

**August 2015**

**Advisor:** Dr. Marwan Abi-Antoun

**Major:** Computer Science

**Degree:** Master of Science

The cost of security vulnerabilities of a software system is high. As a result, many techniques have been developed to find the vulnerabilities at development time. Of particular interest are static analysis techniques that can consider all possible executions of a system. But, static analysis can suffer from a large number of false positives.

A recently developed approach, Scoria, is a semi-automated static analysis that requires security architects to annotate the code, typecheck the annotations, extract a hierarchical object graph and write constraints in order to find security vulnerabilities in a system.

This thesis evaluates Scoria on three systems (sizes 6 KLOC, 6 KLOC and 25 KLOC) from different application domains (Android and Web) and confirms that Scoria can find security vulnerabilities in those systems without an excessive number of false positives.

## AUTOBIOGRAPHICAL STATEMENT

MOHAMMAD ANAMUL HAQUE

### EDUCATION

- Master of Science (Computer Science), August 2015
  Wayne State University, Detroit, MI, USA
- Bachelor of Engineering (Computer Science and Engineering), June 2009
  Jahangirnagar University, Bangladesh