Data-Centric Security: Role Analysis and Role Typestates

Vugranam C. Sreedhar
IBM TJ Watson Research Center
Hawthorne, NY, USA
vugranam@us.ibm.com

ABSTRACT

In J2EE and .NET roles are assigned to methods using external configuration files, called the deployment descriptors. Assigning roles to methods, although conceptually simple, in practice it is quite complicated. For instance, in order for a deployer to assign a role r to a method m, the deployer must understand the set of roles R that are assigned to each method n that can be invoked directly or indirectly from m, and that r has to be "consistently" assigned with respect R. Understanding such role consistency is a nontrivial task. Also, in J2EE roles are defined with respect to method access and not data access. Therefore, in order to protect sensitive data, one has to encode data access control using method access control. This can lead to interesting and subtle access control problems when accessing sensitive data, including information leakage through data flow from one method to another.

In this paper we focus on *data-centric security* by presenting two concepts:

- *Role Analysis:* We present a simple interprocedural static analysis for detecting security problems when objects are accessed by multiple methods that do not have compatible or consistent assignment of roles. We then introduce the notion of an object "escaping" a role and present a simple interprocedural static analysis for computing the set of objects that may escape a role.

- *Consistency-Based Security and Role Typestates:* We extend J2EE method-based role assignment to consistency-based role assignment. In this paper we will focus on assigning roles to *typestates* rather than methods.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Languages, Security

Keywords

RBAC, Role Analysis, Role Typestates, Role Escape Analysis

1. INTRODUCTION

Role Based Access Control (RBAC) is a popular mechanism for defining and managing access to security sensitive resources [9, 26, 10]. In RBAC, security properties, such as access control to sensitive resources, are controlled through *roles*. Users are assigned with one or more roles, who then inherit the security properties associated with the roles. RBAC provides greater security by preventing users from obtaining inconsistent or incompatible security properties. J2EE [1] and .NET [17] support RBAC by restricting the roles to method access.[1][2] In J2EE a role r is simply a named set of methods M, and whenever a principal p is assigned the role r the principal can then access any of the methods in M.[3] In J2EE, there are two ways to specify access control security: (1) declarative or container security and (2) programmatic or application security. In container security, access control to sensitive resources are defined in an external configuration file, as part of a deployment descriptor. The container then manages the access control to sensitive resources. In application security, access controls are encoded within the application and the application directly manages the access control to sensitive resources. The J2EE framework encourages the use of declarative security since it enables greater flexibility by separating security properties from the application logic. We will use declarative security in our discussions.

To illustrate how security roles are defined in J2EE, consider an example application of the Observer Pattern shown in Figure 1 [13]. The Observer Pattern is made of two entities: a *subject* that contains data of interest and (2) one or more *observers* that display the data of interest. The Observer Pattern defines a one-to-many dependency between

[1]The security model of J2EE and .NET are quite similar, and for simplicity we will illustrate our examples using the Java programming language.
[2]J2EE^{TM} and the Java^{TM} programming language are the trademark of the Sun Microsystems. .NET^{TM} is a trademark of the Microsoft Corporation.
[3]In practice only application entry methods are assigned roles (see also Section 2.2).

```
 1: interface IObserver{
 2:  void update(Data info);
 3:}
 4: interface ISubject{
 5:  void addObs(IObserver obs);
 6:  void removeObs
 7:}
 8:class ObsId implements IObserver{
 9:   ObsId(){} ;
 10:   void update(Data info){
 11:      info.printId() ;
 12:   }
 13:}
 14:class ObsName implements IObserver{
 15:   ObsName(){} ;
 16:   void update(Data info){
 17:      info.printName() ;
 18:   }
 19:}
```

```
20:class Subject implements ISubject{
21:  private ArrayList obsList;
22:  private Data  data;
23:  Subject() {
24:     obsList = new ArrayList() ;
25:     data = new Data() ;
26:  }
27:  setData(String name, int id){
28:    dat.name = name ;
29:    data.id = id ;
30:    notifyObs() ;
31:  }
32:  void addObs(IObserver obs){
33:    obsList.add(obs);
34:  }
35:  void removeObs(IObserver obs){
36:     obsList.remove(obs);
37:  }
38:  private void notifyObs(){
39:     for(int i = 0;
40:          i<obsList.size(); i++){
41:       IObserver obs =
42:          (IObserver)obsList.get(i);
43:       obs.update(data );
44:     }
45:  }
46:}
```

```
47:class Data{
48:    String name ;
49:    int id ;
50:    Data(){ } ;
51:    printName(){
52:      System.out.print(name) ;
53:    }
54:    printId(){
55:      System.out.print(id) ;
56:    }
57:  }
58:  class Driver {
59    public static void main(String[] args)
60:    {
61:        Subject sub  = new Subject();
62:        ObserverId oid = new ObsId() ;
63:        sub.addObs(oid) ;
64:        ObsName oname = new ObsName() ;
65:        sub.addObs(oname) ;
66:        sub.setData("Ramanujam", 1729) ;
67:    }
68:  }
```

Figure 1: An example Observer Pattern

the subject and the set of observers. Whenever the data in the subject is updated, all the observers that have been registered with the subject for the data are notified of the update. In J2EE the security roles are defined within an assembly descriptor as shown below:

```
<assembly-description>
  <security-role>
    <role-name> Manager </role-name>
    <role-name> Notifier </role-name>
    <role-name> Director </role-name>
    <role-name> DisplayId </role-name>
    <role-name> DisplayName </role-name>
  </security-role>
</assembly-description>
```

Once the security roles are defined, the application deployer then associates each role with a set of methods. Figure 2 shows how roles are defined as a set of method permissions. For instance, the `Notifier` role has the permission to invoke the method `setData` defined in the `Subject` class.[4] For the example shown in Figure 2 a principal with the `Notifier` role is allowed to access (that is, invoke) the method `setData` that is defined in the class `Subject`. Similarly, a principal with the `DisplayId` role is allowed to access all the methods (denoted by the `*` operator) that is defined in the class `ObsId`.[5] Also, methods that are not part of any role, are allowed to be accessed by any principal.

Now consider a principal `Mark` who is authorized with the role `DisplayId`.[6] Assume that `Mark` goes ahead and invokes the method `Driver.main`. Since this method is not part of any role, the access control manager does not prevent `Mark` from executing the method. Now when `Mark` attempts to invoke `Subject.addObs` (indirectly through `Drive.main`), a security exception will be thrown since `Mark` does not have either the `Director` role or the `Manager` role. At this point an application deployer may assign the `Director` role to `Mark` so as to avoid an access control exception, which in turn may violate the principle of least privilege. Essentially what this means is that in order to associate a role r to a method m, an application deployer has to understand the set of methods S that can be directly or indirectly invoked by m. Understanding such control flow is not a trivial task, especially by an application deployer. In the J2EE security model one can also delegate permissions by associating the `<run-as>` tag to a class C in the deployment descriptor. The effect of such an association is to grant the permission of C to all methods that can be directly or indirectly invoked from C. A careless run-as delegation can introduce some interesting and unforeseen permission problems (see our companion paper Pistoia et al. for more details [25]).

One main problem with RBAC in J2EE is that roles are defined for *controlling access to methods* and *not for explicitly controlling access to data*. In other words, there is no explicit mechanism for controlling access to data fields and object instances. Access control to data and objects have to be done implicitly by giving access to methods which in turn accesses the data. An important security implication of such access control through methods is that information could potentially be leaked through data flow across methods. Consider the example shown in Figure 1. Using pointer analysis [15] one can determine that the object referenced in statement `43:` is accessed by more than one method (e.g., `Subject.notifyObs`, `Subject.addObs`, and `ObsId.update`). Unless the roles are assigned correctly or consistently among these methods there could be potential access control and information flow security prob-

[4]In J2EE roles are typically defined for EJB components rather than arbitrary classes. For simplicity we will use examples of general Java classes, instead EJB examples.

[5]In J2EE one can use such access control specification for other Web resources such as access to a HTTP method on a URI (Universal Resource Identifier).

[6]In the rest of the paper we assume authentication mechanism similar to J2EE authentication mechanism [1].

```
<method-permission>
   <role-name> Notifier </role-name>
      <method>
         <class-name> Subject </class-name>
         <method-name> setData </method-name>
      </method>
   <role-name> Manager </role-name>
      <method>
         <class-name> Subject </class-name>
         <method-name> addObs </method-name>
         <method-name> removeObs </method-name>
      </method>
  <role-name> Director </role-name>
      <method>
         <class-name> Subject </class-name>
         <method-name> * </method-name>
      </method>
   <role-name> DisplayId </role-name>
      <method>
         <class-name> ObsId </class-name>
         <method-name> * </method-name>
      </method>
   <role-name> DisplayName </role-name>
      <method>
         <class-name> ObsName </class-name>
         <method-name> * </method-name>
      </method>
<method-permission>
```

Figure 2: An example role definition

lems. Consider once again the role assignment shown in Figure 2. We can see that the method `ObsName.update` and and the method `ObsId.update` are part of two different roles. Therefore, when `obs.update()` is invoked at statement `43:`, the target method could be either `ObsId.update` or `ObsName.update` depending on the runtime type of `obs`. Since, `ObsName.update` and `ObsId.update` have two different roles, there could be a potential security problem when the method is dispatched. Therefore it is important to ensure that both methods are assigned the same role. The security problems that arise are either: (1) too many permissions are given to a principal, which in turn may violate the principle of least privilege, or (2) inadequate permissions are given, which can lead to unnecessary access control exceptions. There can be subtle security violation, such as information tainting and information leakage, when the principle of least privilege is compromised. Notice that a simple propagation of roles over the call graph of the program is not sufficient to detect such data flow security problems.

In this paper we focus on *data-centric security* by introducing two concepts:

- *Role Analysis:* We present a simple interprocedural static analysis for detecting security problems that may arise when objects are accessed by multiple methods that do not have compatible or consistent assignment of roles to methods. We then present the notion of an object "escaping" a role and a simple interprocedural static analysis for computing the set of objects than may escape a role.
- *Consistency-Based Security and Role Typestates:* We extend J2EE's method-based role assignment to consistency-based role assignment. In this paper we will focus on assigning roles to *typestates* rather than methods. The typestate description of a class is a configuration of its fields, and it essentially abstract consistency properties of the fields [8].

The rest of the paper is organized as follows: In the next section we review the necessary background material for modeling and analyzing roles. In Section 3 we discuss role analysis, including role escape analysis. In Section 4 we discuss consistency-based security, including role typestate analysis. In Section 5 we discuss some of the related work and conclude in Section 6.

2. RBAC MODELS

In this section we review some background material related to role modeling and static analysis for propagating roles. In Section 2.1 we briefly introduce a call graph and a pointer graph representation. In Section 2.2 we introduce role graph representation for modeling role structure that is based on a lattice structure.

2.1 Program Representation

Static analysis is a process for determining the relevant properties of a program without actually executing the program [21]. A program consists of a collection of methods or procedures and is represented using a call graph (CG) [14]. A CG $G = (N, E)$ is a directed graph, where N is a set of nodes and E is a set of edges. The set of nodes N is partitioned into two disjoint sets: (1) N_c is a set of call site nodes and (2) N_m is a set of method nodes. The set of edges E is also partitioned into two disjoint sets: (1) $E_{m,c}$ is a set of edges from a method node x_m to each call site node x_c defined in the method, and (2) $E_{c,m}$ is a set of edges from a call site node x_c to each method node x_m that can possibly be invoked from the call site x_c. A path is a sequence of edges starting from some node in the call graph. If there is a path from a method node x_m to another method node y_m then we say that the method corresponding to x_m may directly or indirectly invoke the method corresponding to y_m. Figure 3 illustrates the call graph for the example program of Figure 1.

Pointer analysis consists of computing points-to information, represented as a pointer graph, at relevant program points [15]. A pointer graph consists of a set of nodes representing compile-time objects and pointer references. For heap allocated objects, we say that p can point-to an object O if p can contain the address of O. Typically the address of O is not know at compile-time and one then assign compile-time labels to heap objects. There are several different kinds of pointer analysis depending on precision and cost of the analysis [15]. Typically flow insensitive and context insensitive analysis tends to be cheaper, but less precise than flow sensitive and context sensitive analysis. Figure 4 illustrates a pointer graph that is computed using a flow insensitive pointer analysis. We will use the notation $p \rightarrow O$ to denote that p points-to O. We will also assume objects have fields and each field has a name that can be accessed from the object type or class. We access fields using dot-notation, $p.f$ or $O.f$, and so we can state that $p.f \rightarrow O$ to denote that p points to some object O_1 and the field f of O_1 points to another object O.

2.2 Role Modeling

J2EE security is defined in terms of: (1) *principals* or authenticated users, (2) *roles* that define named job func-

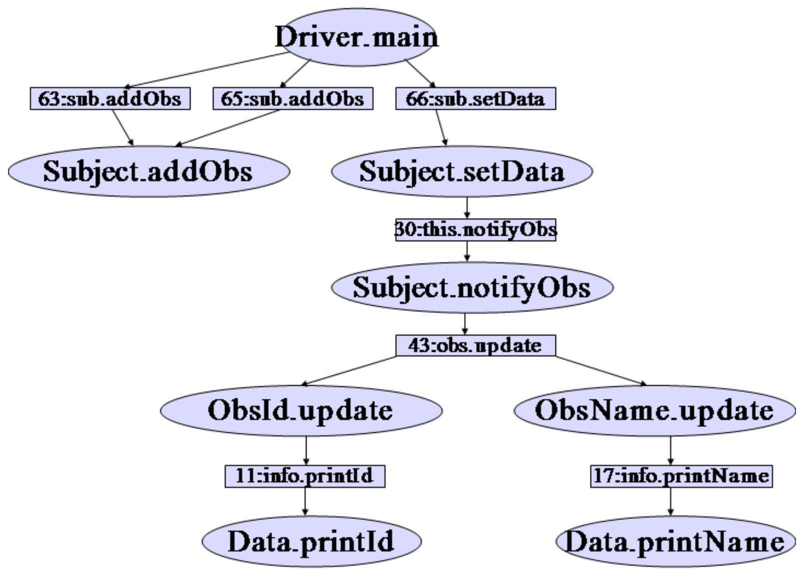


Figure 3: Call graph for the example program shown in Figure 1.

tions, and (3) *permissions* that maps a role to a set of operations or methods defined in one or more classes.[7] For instance, consider the `Subject` class that contains the methods `setData`, `addObs`, `removeObs`, and `notifyObs`. A role such as `Notifier` (see Figure 2) is given the permission to invoke `setData`. On the other hand a role such as `Director` is given the permission to invoke all four operations. Notice that a principal with `Director` role can invoke any operation that a principal with a `Manager` role or a `Notifier` role can invoke. In other words, `Director` is considered to be "senior" to `Manager` and `Notifier` [10]. Let C denote a set of classes for which we want to provide RBAC and let M denote a set of methods in C. A permission Per is a mapping from role $r \in R$ to a subset of methods in M. Now let $Per(r)$ denote the set of method permissions assigned to a role r. For instance, $Per(\texttt{Manager}) = \{\texttt{Subject.addObs}, \texttt{Subject.removeObs}\}$

Definition 1 (Senior Roles). *Let r and s be any two roles in R. We say that r is a* senior role *of s, denoted as $r \succeq s$, if $Per(r) \supseteq Per(s)$.*

We can conversely define *junior role* as follows: $r \preceq s$ if $Per(r) \subseteq Per(s)$. Next we define a *role graph* in which nodes represent roles and edges represent the senior (junior) relations. In this paper we will assume that the role graph form a lattice structure, with $\top$ representing a role that is the union of all method permissions and $\bot$ representing a role whose permission set is the empty set [10]. A lattice is a partial order with a join and a meet for every pair of nodes. Given any two nodes x and y, the set union of x and y is the join ($\sqcup$), and the set intersection of x and y is the meet ($\sqcap$).

Now let $Per_i(r)$ be a set of method permission that is *initially assigned* a role r (as specified in the deployment descriptor). In J2EE a method permission can be assigned to more than one role. Let $Role_i(m)$ denote the set of roles that an application deployer initially assigns to m. In other words, $Role_i(m) = \{r | m \in Per_i(r)\}$. For instance, $Role_i(\texttt{Subject.setData}) = \{\texttt{Notifier}, \texttt{Director}\}$. Now if a method is not explicitly assigned to any role, then it can be accessed by any principal. Note that one can also use the $*$ to indicate all the methods in a class as belonging to a specific role.

[7]In J2EE roles are often defined for Enterprise JavaBeans (EJBs), and in this paper for simplicity, we will simply use Java classes instead of EJBs.

3. ROLE ANALYSIS

In this section we present two role analyses: (1) role consistency analysis for computing consistent role assignment and (2) role escape analysis for computing the set of objects that may escape a role.

3.1 Role Consistency Analysis

Let r be a role assigned to m (by an application deployer), and let m' be a method that can be invoked directly or indirectly from m. A principal who is assigned the role r should also have permission to invoke the method m', i.e., m' should also be in $Per(r)$. Often an application deployer has handle to only entry methods of an application or a component. These entry methods are essentially the application programming interface (API) of the application.[8] Therefore an application deployer has the capability to define permission sets and assign roles only based on the entry methods defined in an API. Let A be an application, let M_e be a set of entry methods to A (as defined in its API), and R be the set of roles defined using the entry methods. Figure 5 illustrates the application or component model. Notice that an entry method such as $m1$ can directly or indirectly invoke another entry method $m2$. The key question to ask is: how to define consistent roles and permission sets that is based on only the entry methods of an application? Consider an

[8]In the case of a Java application, the entry methods are the set of `public` and `protected` methods defined in interfaces and classes.

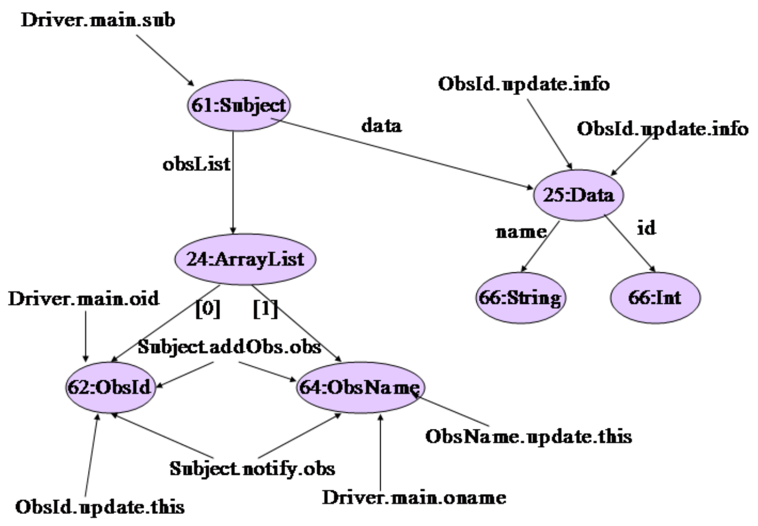


Figure 4: Pointer graph for the example program shown in Figure 1.

entry method $m_1 \in M_e$ and let $r_1 \in R$. Now let us define r_1 to include m_1 in its permission set, that is $m_1 \in Per(r_1)$, and therefore $r_1 \in Role(m_1)$ (see Figure 5). What this essentially means is that if a principal p is assigned the r_1, then p has the permission to invoke m_1. Now let m_2 be some other method reachable from m_1, and so p should also have the permission to invoke m_2. Let $Per(r_2) = \{m_2, m_3\}$, and so $m_2 \in Role(r_2)$. We have to ensure that r_1 and r_2 have consistent permission set. There are two cases to consider:

- m_2 is not in $Per(r_1)$, and in this case p should also be assigned the role r_2 so as to avoid access control exception. On one hand, by assigning r_2 to p, we also give p the permission to access m_3, which violates the principle of least privilege. On the other hand, $m3$ may be in different sub-component, in which case separating the roles makes sense in some situations, even if the principle of least privilege is violated.

- m_2 is in $Per(r_1)$, and in this case the assignment is consistent.

One way to compute consistent role assignment is to first determine the set of methods that are reachable from an entry point in the call graph of the program. Let L be the set of methods that are reachable from an entry point $m \in M_e$ in the call graph. Let $L'_m = L \cap M_e$, and so L'_m is a set of methods that are reachable from m and the set of method are also entry methods. Now let r be a role with $m \in Per(r)$. Since a method $l' \in L'_m$ is also reachable from m, we have to ensure that $l' \in Per(r)$. So rather than defining arbitrary roles and permission set, a deployer first computes L'_m for each entry method m. The deployer can then define roles based on the entry method m and L'_m by ensuring that $L'_m \subseteq Role(m)$.

Typically applications are composed of many components and the components must be assigned with consistent roles. Consider the example shown in Figure 1 and the role assignment shown in Figure 2. Let $Role_i$ denote the initial set of role assignments. In order for a principal to invoke

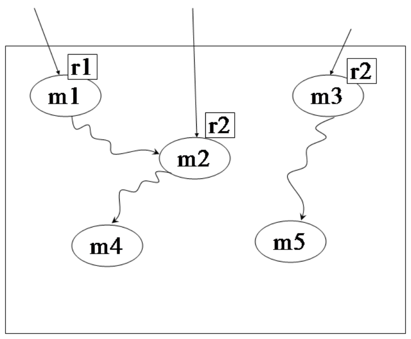


Figure 5: An application model with entry methods and role assignments.

`Subject.setData` the principal not only needs the `Notifier`, but also `DisplayId` and `DisplayName` roles. In the Figure 6 we have annotated the call graph with $Role_i$ and $Role^+$, which are the initial role assignments and the minimum roles needed to invoked the methods. Computing $Role^+$ for each method is a straightforward backward propagation of roles over the call graph and at each step we only propagation *junior* roles up the call graph. That is, we perform a lattice join at each step and propagate the joined role information.

The role analysis described previously is based on reachability over call graph of a program. Although role propagation over call graph is important in preventing certain kinds of access control problems, it does not detect access problems that could happen due to data flow across methods.[9] Subtle problems may arise due to data flow across methods. For instance, the receiver expression (such as

[9]A limitation of J2EE security model is that it focuses on securing the mechanism (that is, methods) that access sensitive information or resources. For instance, J2EE does not allow one to specify roles for class fields and class instances.

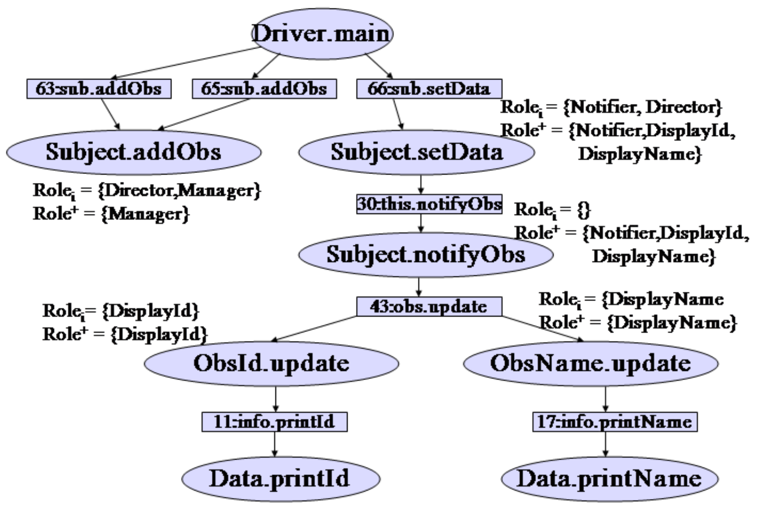


Figure 6: Call graph annotated with $Role_i$ and $Role^+$ for each method.

`43: obs.update(data)` can target more than one method. We have to make sure that the roles assigned to the target method is consistent with the role assigned to method that invokes the object.

Also, an object is often accessed by multiple methods that have different roles assigned to them. Consider the points-to graph shown in Figure 4 and in particular the object `62:ObsId`. This object is directly accessed by the methods `Driver.main` (which created the object), `ObsId.update` (via the `this` parameter), and `Subject.notifyObs`. Now consider the role assignment and method permission shown in Figure 6. The method `ObsId.update` is assigned only to role `Display`. Unfortunately since the object `62:ObsId` is also accessed by other methods, there can be inconsistencies among roles assigned to different methods that access the same object. This can lead to information leakage.

Given the pointer graph of a program, we compute for each object the set of methods that *directly access the object*. Let us call the resulting graph as *method-annotated pointer graph* (MAPG). It is straightforward to compute this information if we use fully qualified names for references. Consider the pointer graph shown in Figure 4 and the object `62:ObsId`. The set of methods that access this object can be read off from the pointer graph, which is `Driver.main`, `Subject.addObs`, `Subject.notify`, and `ObsId.update`. Since any of these methods have the "capability" to access the object, the object must be assigned a role that is the *least upper bound* of the roles assigned to these methods. Now a principal who wants to access the object referenced by `62:ObsId` must have a role that is at least equal to the least upper bound of the roles assigned to the methods that access the object. Unfortunately, in J2EE it is not possible to assign roles to objects or class instances (also see Section 4).

3.2 Role Escape Analysis

In this section we introduce the concept of role escape analysis inspired by method and thread escape analysis [7]. Escape analysis is a procedure for determining the set of objects that escape the life time of a method or a thread. Consider the following simple example:

```
1: void foo() {
2:  LinkedList head  = new LinkedList() ;
3:  bar(head) ;
4: }
5: void bar(LinkedList h){
6:   LinkedList n = new LinkedList() ;
7:   n.next = null ;
8:   h.next = n ;
9: }
```

The object `6:bar.LinkedList` created at `6:` escapes the method `bar` because there is an access path to the object from the parameter reference `h`. On the other hand the object `2:foo.LinkedList` created at `2:` and the object `6:bar.LinkedList` created at `6:` does not escape `foo`. One simple way to compute whether an object O escapes a method M is first to construct the pointer graph and then check if there is a path to O in the graph that can be reached by some node O' that is accessed in some other method M' and the life time of M' exceeds the life time of M.

In *role escape analysis* we extend the method escape analysis as follows. An object O *role escapes* a method M with role R if there is a path in the pointer graph from some node O' that is accessed in some other method M' with role R' and $R \not\preceq R'$. Conversely, an object O is *role confined* to a M if O does not role escape M.[10] The role escape analysis essentially consists of determining if a node p in the pointer graph that is annotated with method m and role $r \in Role^+(m)$ is reachable from another node p' that is annotated with m' and role $r' \in Role^+(m')$ and $r \not\preceq r'$.

Notice that the converse of role escape, which is role confinement, of an object is closely related to information flow security of Bell and LaPadula's Multi-Level Security with

[10]Our definition of role escape is more general than the traditional method or thread escape, where by we define method lifetime as a role.

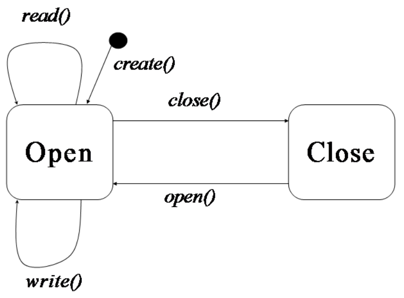


Figure 7: A typestate diagram for the `File` example.

Mandatory Access Control [5]. Since we model roles using a lattice structure, if an object O does not escape a role R then the object cannot leak information outside the role. Also, if an object escapes a role, the object can become "tainted" by an external principal with access to the object [19, 24]

4. CONSISTENCY-BASED SECURITY

It is often the case that an application developer has a better understanding of the application that he or she is developing than an application deployer or the system administrator. On the other hand an application deployer has a better understanding of the application deployment environment than an application developer. The key idea behind consistency-based security (CBS) is to focus on the *consistency properties* of data and methods. An application developer simply identifies and defines the consistency properties as code annotation. A simple consistency property could be that two fields of a Java class are modified by the same set of methods. Another consistency property is the *typestate property*, which is often used to specify ordering of methods in an application. In CBS, a tool can scan the code and present the developer's consistency properties to an application deployer. An application deployer can then use the consistency properties as a guide for associating security properties to roles, and then grant authorization to users.

4.1 Data Consistency Security

In this section we will illustrate a simple consistency property in which two or more fields of an object are all accessed by the same set of methods. Given this property, an application deployer can then associate the same role for all methods that access these fields. Consider a simple employee record in a company.

```
enum EmpType = {RSM, Manager, SoftEng, Staff} ;
class Employee {
   String fname ; // first name
   String lname ; // last name ;
   int id ;       // company identity number
   int ssn ;      // social security number
   String gender ;
   String ethnic ; // ethnicity
   EmpType etype ;
}
```

Supposing an application developer is developing a business application, such as payroll application, and decides that `gender`, and `ethnic` have the same "consistency property". A simple example of a consistency property is that the two fields are always accessed by the same set of methods. Another example of a consistency property is *access rights*; if a user is permitted to access one of the two data fields, then the user is automatically allowed to access the other data field. Yet another example of a consistency property is *encryption*; the values of the two data fields must use the same encryption/decryption keys. Typically an application developer has a better understanding of the consistency properties than the application deployer. For instance, if `gender`, and `ethnic` are always accessed by the same set of methods, the developer can then define a simple consistency property classification called `race`.

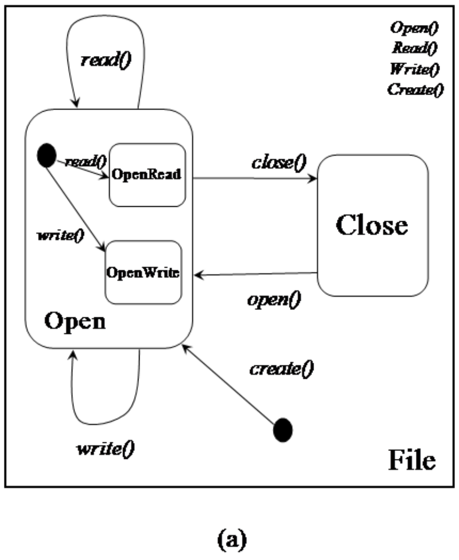
```
class Employee {
    access(name, idy)  String fname ;
    access(name, idy)  String lname ;
    access(idy)        int id ;
    access(idy)        int ssn ;
    access(race) String gender ;
    access(race) String ethnic ;
}
```

Often a field (or a method) can belong to more than one class of a consistency property. For instance, `fname` and `lname` belong to two different classes of the `access` property. A consistency property consists of two parts: (1) a property name such as `access` and a set of classifications, such as `name`, `idy`, and `race`. Given the consistency properties defined by an application developer, a natural question to ask is how will the application deployer use them. An application deployer first must query the application for all application-defined properties. Next, the application deployer associate *roles* to consistency properties. Supposing an application deployer defines two roles called `Manager` and `NonManager`, and associates the `Manager` role with `access(name,idy,race)` and `NonManager` role with `access(name)`. With this association, a `Manager` can access all the defined fields, whereas a `NonManager` can only access the fields `fname` and `lname`. Notice that a deployer does not need to understand how a developer defined the consistency properties. The deployer only has to know what set of consistency properties have been defined by the developer.

4.2 Role Typestates

In this section we define RBAC by focusing on the *typestate consistency property*. The approach that we advocate in this section is very simple: an application developer defines the typestate properties and an application deployer then assign roles based on the typestate properties. Typestates provide much richer information than simple method interfaces to an application deployer.

Strom and Yemini introduced the concept of typestate as an extension to the notion of a type by requiring that a variable be in certain *state* before operations on the variables can be performed [27]. In other words, certain preconditions must be met prior to performing any operation on a variable. Typestate was originally introduced for tracking certain kinds of flow-sensitive errors or bugs, such as uninitialized variables. For object-oriented (OO) programs, the typestate of an object (that is, an instance of a class) is a description or a configuration of all its fields [8]. In OO programs, a method that is invoked on an object o typically



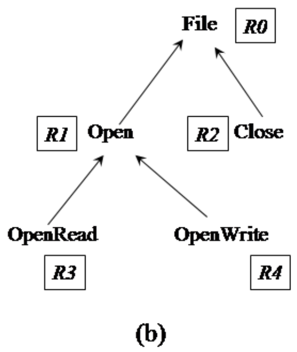


Figure 8: (a) A hierarchical typestate diagram and (b) Role hierarchy assignment.

has a partial view of the object o. One can use typestates to define a consistent view of an object prior to an invocation of a method on the object. Consider a simple file object that contains two typestates `Open` and `Close`.

```
class File {
typestate = {Start, Open, Close}
// @requires Start
File create(String fname) ; // create file
// @requires Close
void open() //
// @requires Open
String read() ; // returns the content of file
// @requires Open
void write(String data) ; // write to file
// requires Open
void close() ; // close the file
}
```

A typestate diagram is a finite state diagram, with nodes denoting the typestates of a class and the labeled edges denoting transition between typestates of a class. The labels on the edges correspond to (a subset of) methods defined in the class. The source and destination nodes of a transition correspond to pre- and post-conditions of the method that labels the transition, respectively. An execution of a method takes an instance of a class from one typestate to another typestate. Figure 7 illustrates a typestate diagram for the `File` class. The typestate diagram specifies the life-cycle of how an instance of the `File` class goes between `Open` and `Close` typestates. A method, such as `open()`, is executed only when its pre-condition typestate of the `File` object is `Close`, and after the execution the new typestate of the object is `Open`. Both `read()` and `write()` methods can execute only when the `File` object is in `Open` state.

The typestate diagram is an external specification of a class. It prescribes the order in which a client can invoke various methods defined in the class. Only an application developer understands the lifecycle of an object. Given such a specification, an application deployer can now associate security properties to typestates. For instance, an application deployer can associate `Manager` role to `Start`, `Open` and `Close` typestates, and `Engineer` role to `Open`. What this essentially means is that only a `Manager` is allowed to execute all methods of the class, whereas an `Engineer` is allowed to execute only `read()` and `write()` methods.

We partition a typestate diagram into a hierarchical structure and use the notion of hierarchical typestate diagram [16]. A hierarchical typestate diagram consists of a set of *states* and a set of *transitions*. A *state* can be composed of other states, called the *sub-states*. This enables modeling of complex hierarchical typestate diagram by abstracting away detailed behavior into multiple levels. States that do not contain sub-states are called *simple states*, whereas states that have sub-states are called *composite states*. States may be nested to any level. A *transition* relates two states: a source state and a destination state. Transitions are labeled and each label corresponds to a method invocation. An invocation of a method can cause a state transition. The hierarchical states in a typestate diagram induces a tree structure. Figure 8 illustrates a hierarchical typestate diagram in which we have expanded the `Open` state shown in Figure 7.

For role typestates, it is natural to model roles using a tree structure. This is illustrated in Figure 8(b). The definition of *senior relation* is simpler: $r \succeq s$ if r is a *parent* of s in the role hierarchy tree. We assign roles to hierarchical states in the typestate diagram. If s is assigned the role r then the $Per(r) = \{t | t \in Root(s)\}$, where $Root(s)$ is a set of all states in the sub-tree that is rooted at s in the typestate hierarchy tree. Consider the role assignment shown Figure 8(b), and the permission set for each roles are $Per(R0) = \{\mathtt{File}, \mathtt{Open}, \mathtt{Close}, \mathtt{OpenRead}, \mathtt{OpenWrite}\}$, $Per(R1) = \{\mathtt{Open}, \mathtt{OpenRead}, \mathtt{OpenWrite}\}$, $Per(R2) = \{\mathtt{Close}\}$, $Per(R3) = \{\mathtt{OpenRead}\}$, and $Per(R4) = \{\mathtt{OpenWrite}\}$. Once again, consider the role assignment shown in Figure 8(b). If a principal p is assigned the role $R1$ then p can invoke both `read` and `write`, but cannot invoke close. Now if p is assigned $R3$ then p can only invoke the read method.

Now consider a method such as File.close() and a principal p who is assigned the role $R1$. The principal p should not be allowed to invoke File.close() even if the current typestate of the object referenced by the principal is Open. This is because, the method File.close() creates a state transition from Open to Close and p is not allowed to be in state Close. So only principals with role File are essentially allowed to execute File.close(). Given a set of roles assigned to typestates, we can easily compute the roles needed for methods—it is simply the least common ancestor of the roles assigned to pre- and post-state of the method. Notice that when a File object is passed around to different methods that are accessed by different principals, we only have to ensure that roles are consistent on the current typestate of the object and not have to worry about the methods themselves.

Typestates are a useful formalism for modeling access patterns, including message patterns in Web Services. Once such message patterns are modeled using typestates, we can assign roles to states using our approach. We are currently exploring this line of research in Web Service security.

5. DISCUSSION AND RELATED WORK

The main focus of this paper is to address the challenges of role assignment and role semantics as it is currently used in building J2EE applications. The current method-based access control is limiting in terms of its expressive power. We believe that some form of data-centric security should complement the method-based security.

Ferraiolo and Kuhn [9] introduced the notion role-based access control for managing and controlling access to sensitive resource. Since then RBAC model has become popular because of its generality in expressing a diverse set of access control policies [26, 22]. Li and Tripunitara propose security analysis techniques for ensuring that security properties are correctly maintained even in the presence of delegating administrative privileges [18]. Their approach is not directly related to J2EE RBAC model, although their approach can be used to detect some of the access control problems with delegation in the J2EE model.

Pistoia et al. presents static analysis techniques for identifying inconsistent security role assignment in J2EE applications [25]. Pistoia et al. model roles as logical expressions rather than as set of method, and these logical expression can get very complex. We use a simpler lattice structure for modeling roles. Pistoia et al. also use a more precise context sensitive analysis for tracking logical expressions and also deal with role delegation semantics of J2EE model. To simplify the presentation we chose to ignore role delegation, which in J2EE is ambiguous. Pistoia et al. approach models the method-based role assignment, and does not deal with data flow aspect of RBAC. In our paper we introduce the notion of role escape analysis to specially address the need for data-centric security. We also presented consistency-based security, along with role typestates for specifying access control security.

The role analysis described in this paper is also related to static analysis described by Naumovich and Centonze [20]. They also rightfully recognize the need for extending J2EE RBAC to explicitly deal with data fields, but do not give any solution for how to extend the model. Instead they focus on detecting access-control inconsistencies when two or more J2EE methods accessing the same data in the same mode (either read or write) but are assigned with different roles. They also do not discuss the concept of role escape analysis and role typestates. Our consistency-based security and role typestate is a way to extend J2EE model to deal with data-centric security.

Java 2 Standard Edition (J2SE) provides security mechanisms for protecting resources (e.g., file system) in terms of run-time stack inspection. The security access control model is based on properties that includes code origin and principal's execution environment. Banerjee and Naumann give static analysis techniques for characterizing safe expressions, that is, expressions that never cause security violations. Banerjee and Naumann have presented several results in a series of papers on access control and information flow control [2, 3, 4]. They approach the problem using language techniques, such as type theory, to detect security violations and safety violations. We follow program analysis techniques for RBAC and also use typestate properties for RBAC. We also introduce the notion of role escape analysis and the notion of role typestates to model and detect security problems in J2EE. Recently Pistoia et al presented an interprocedural analysis to detect portions of trusted Java code should be made privileged, and also to ensure that there are no tainted variables in privileged code [24].

Park and Goldberg introduced the term escape analysis in the context of functional program analysis for statically determining parts of a list that are passed to a function do not escape the function call, and hence can be allocated on stack [23]. Recently there have been several work on escape analysis in the context of Java [7, 6, 12]. Escape analysis techniques have been applied both for allocating objects on the call stack and for eliminating synchronization in multithreaded programs. Our work on role escape analysis generalizes the traditional escape analysis, where we can consider method and thread lifetime to be an instance of roles. Role escape analysis is useful for detecting confinement properties of objects, and for detecting information leaks through objects [11].

Role typestates combines the notion of roles and typestates. The original work on typestate focused on finding flow-sensitive errors [27]. DeLine and Fahndrich extended the classical typestate theory to objects [8]. They use pre- and post-conditions to express allowed transition rules between the typestates of the object, and typestates to express predicates (or constraints) over the objects concrete state, which includes the field states. Now by associating roles to typestates we define a more general model for specifying roles—instead of specifying roles on methods, we now specify them on typestates.

6. CONCLUSIONS

In this paper we brought out some of the security issues related to J2EE RBAC model. We illustrated several examples that show the limitation of method-based role assignment. We then discussed the concepts that bring out data-centric security model. We believe consistency based security, that includes typestate security, should complement method-based security. Our next step is to define and extend data centric security in the context of web applications, including web services [11].

Acknowledgement

I wish thank Larry Koved, Samuel Weber, and Xiaolan Zhang for commenting on the earlier drafts of the paper.

7. REFERENCES

[1] Eric Armstrong, Jennifer Ball, Stephanie Bodoff, Debbie Bode Carson, Ian Evans, Dale Green, Kim Haase, and Eric Jendrock. *The J2EE 1.4 Tutorial* . Sun Java System Application Server Platform Edition , 2005.

[2] Anindya Banerjee and David A. Naumann. Representation independence, confinement and access control [extended abstract]. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 166–177, New York, NY, USA, 2002. ACM Press.

[3] Anindya Banerjee and David A. Naumann. Secure information flow and pointer confinement in a java-like language. In *CSFW '02: Proceedings of the 15th IEEE Computer Security Foundations Workshop (CSFW'02)*, page 253, Washington, DC, USA, 2002. IEEE Computer Society.

[4] Anindya Banerjee and David A. Naumann. Stack-based access control and secure information flow. *J. Funct. Program.*, 15(2):131–177, 2005.

[5] D. E. Bell and L. J. LaPadula. Secure computer system: unified exposition and multics interpretation. Technical Report MTR-2997, MITRE Corporation, March 1976.

[6] Bruno Blanchet. Escape analysis for object-oriented languages: application to java. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 20–34, New York, NY, USA, 1999. ACM Press.

[7] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape analysis for Java. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, Denver, Colorado, November 1999.

[8] R. DeLine and M. Fahndrich. Typestates for objects. In *18th European Conference on Object-Oriented Programming*, 2004.

[9] David F. Ferraiolo and D. Richard Kuhn. Role-based access controls. In *15th NIST-NCSC National Computer Security Conference*, pages 554–563, Baltimore, MD, USA, October 1992.

[10] David F. Ferraiolo, Ravi Sandhu, Serban Gavrila, D. Richard Kuhn, and Ramaswamy Chandramouli. Proposed nist standard for role-based access control. *ACM Trans. Inf. Syst. Secur.*, 4(3):224–274, 2001.

[11] Elena Ferrari and Bhavani Thuraisingam. *Web and Information Security.* Idea Group Publishing, 2006.

[12] FredericVivien and Martin Rinard. Incrementalized pointer and escape analysis. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 35–46, New York, NY, USA, 2001. ACM Press.

[13] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements od Reusable Object-Oriented Software.* Addison-Wesley Publishing Company, New York, NY, 1995.

[14] David Grove and Craig Chambers. A framework for call graph construction algorithms. *ACM Trans. Program. Lang. Syst.*, 23(6):685–746, November 2001.

[15] Michael Hind. Pointer analysis: haven't we solved this problem yet? In *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 54–61, New York, NY, USA, 2001. ACM Press.

[16] Patrick Lam, Viktor Kuncak, and Martin Rinard. Generalized typestate checking using set interfaces and pluggable analyses. *SIGPLAN Not.*, 39(3):46–55, 2004.

[17] Brian A. LaMacchia, Sebastian Lange, Matthew Lyons, Rudi Martin, and Kevin T. Price. *.NET Framework Security* . Pearson Education, 2002.

[18] Ninghui Li and Mahesh V. Tripunitara. Security analysis in role-based access control. In *SACMAT '04: Proceedings of the ninth ACM symposium on Access control models and technologies*, pages 126–135, New York, NY, USA, 2004. ACM Press.

[19] V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in java applications with static analysis. In *Proceedings of the 14th USENIX Security Symposium*, 2005.

[20] Gleb Naumovich and Paolina Centonze. Static Analysis of Role-Based Access Control in J2EE Applications. In *Workshop on testing, analysis and verification of web services*, pages 1–10, New York, NY, USA, 2004. ACM Press.

[21] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[22] S. L. Osborn, R. Sandhu, and Q. Munawer. Configuring role-based access control to enforce mandatory and discretionary access control policies. *ACM Transactions on Information and System Security*, 3(2), February 2000.

[23] Young Gil Park and Benjamin Goldberg. Escape analysis on lists. In *PLDI '92: Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, pages 116–127, New York, NY, USA, 1992. ACM Press.

[24] Marco Pistoia, Robert J. Flynn, Larry Koved, and Vugranam C. Sreedhar. Interprocedural analysis for privileged code placement and tainted variable detection. In *19th European Conference on Object-Oriented Programming*, pages 362–386, 2005.

[25] Marco Pistoia, Vugranam Sreedhar, and Robert Flynn. Static evaluation of role-based access control policies in distributed component-based systems. Technical Report RC23836 (W0411-166), IBM TJ Watson Research Center, IBM Research Division, Yorktown, NY, November 2004.

[26] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[27] R. Strom and S. Yemini. Typestate: a programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12(1), Jan 1986.