RC23846 (W0511-020) November 3, 2005
Computer Science

IBM Research Report

Interprocedural Analysis for Automatic Evaluation of Role-Based Access Control Policies

Marco Pistoia
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Robert J. Flynn
Polytechnic University
6 Metrotech Center
Brooklyn, NY 11201

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Interprocedural Analysis for Automatic Evaluation of Role-Based Access Control Policies

Marco Pistoia[1] and Robert J. Flynn[2]

[1] IBM Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, USA
pistoia@us.ibm.com,
WWW home page: http://www.research.ibm.com/people/p/pistoia
[2] Polytechnic University, 6 Metrotech Center, Brooklyn, NY 11201, USA
flynn@poly.edu,
WWW home page: http://www.poly.edu/faculty/robertflynn

Abstract. This paper describes an interprocedural-analysis model to represent the flow of security information in systems that have adopted Role-Based Access Control (RBAC). These systems include Java 2, Enterprise Edition (J2EE) and Microsoft .NET Common Language Runtime (CLR). The model allows:

1. Identifying the roles required to execute an enterprise application
2. Detecting potential inconsistencies caused by principal-delegation policies, which are used to overwrite the roles assigned to a user
3. Reporting if the roles assigned to a user by a given policy are redundant, which would constitute a violation of the Principle of Least Privilege, or insufficient, which would make the application unstable
4. Evaluating logical expressions of roles
5. Distinguishing intercomponent resource accesses (in which authorization *is* enforced) from intracomponent resource accesses (in which authorization *is not* enforced)

The algorithms described in this paper have been implemented as part of the Enterprise Security Policy Evaluator (ESPE) tool, built on top of IBM Research's DOMO static analysis engine. This paper presents the results obtained by executing ESPE on several applications.

1 Introduction: Role-Based Access Control Systems

Role-Based Access Control (RBAC) [8] is gaining popularity in managing the security of large enterprise applications. In RBAC, authorization to access restricted resources is controlled through "security roles" rather than through the user identity. Java 2, Enterprise Edition (J2EE) [34, 18] and Microsoft .NET Common Language Runtime (CLR) [10] have adopted a form of RBAC for defining and managing security of enterprise applications. This paper presents an interprocedural analysis framework [23] that models the flow of security information in an RBAC system and allows for automatic detection of security-policy misconfigurations. While the analysis techniques described in this paper are in the context of J2EE code, the basic concepts are applicable to RBAC issues in non-J2EE systems as well.

1.1 Security Roles

A *security role* (*role* for short) is a semantic grouping of access rights [25]. Roles can be assigned to users of an enterprise application. While users and groups are defined at the J2EE server level, roles are application-specific; each application defines its own security roles. In a J2EE application, for example, it is possible to define the role of `Employee` and specify that method `m1` in enterprise bean `Bean1` can be accessed only by those principals who have previously been assigned the role of `Employee`. If user Bob Smith successfully logs on to the J2EE server as `bob` and attempts to execute—directly or indirectly, through a sequence of other method calls—`m1` on `Bean1`, the method invocation will succeed only if `bob` was previously granted the role of `Employee`.

1.2 Declarative Security

J2EE and CLR promote the concept of *declarative security*. This means that it is not necessary to embed authentication and authorization code within an application. Rather, security information is stored along with other deployment information in configuration files that are external to the application code. In J2EE, these configuration files are called *deployment descriptors* and are defined in eXtensible Markup Language (XML). The following fragment defines the `Employee` role and restricts access to method `m1` in enterprise bean `Bean1`:

```
<security-role>
   <role-name>Employee</role-name>
</security-role>
<method-permission>
   <role-name>Employee</role-name>
   <method>
      <ejb-name>Bean1</ejb-name>
      <method-intf>Remote</method-intf>
      <method-name>m1</method-name>
   </method>
</method-permission>
```

A system administrator deploying `Bean1` must be aware that its method `m1` can only be accessed by principals assigned the role of `Employee`, and will have to assign that role to users accordingly.

1.3 Security and Stability Problems in J2EE Applications

This section illustrates the complications that can arise when configuring the security of a J2EE application.

Intra- vs. Intercomponent Calls The J2EE specification [34] dictates that when a restricted resource in a component, such as a method in an enterprise bean, is accessed from another component, the J2EE container must perform an authorization check. However, the container will not perform authorization checks if the same resource is accessed from its own component. It will be shown later that lack of such intracomponent authorization checks can lead to security compromises or unnecessary authorization failures.

Principal Delegation In J2EE, by default, the identity of the principal who initiated a transaction on the client is propagated to the downstream calls. However, some enterprise resources may need to be executed as though they were called by a principal with a different role. For this purpose, J2EE allows associating "principal-delegation policies" with components. A *principal-delegation policy* consists of a `run-as` entry in a component's deployment descriptor. The entry's value is the name of a role specific to the application to which that component belongs. The effect is that all the downstream calls from that component onward will be performed as if the caller had been granted only the role specified in the `run-as` entry. It should be observed that:

1. A principal-delegation policy allows specifying only one role and overrides all the roles possessed by any user up to that point. This implies that:
   - If the user was originally granted multiple roles, a principal-delegation policy will override all those roles with the only role specified by its `run-as` entry.
   - A principal-delegation policy overrides any role set by previously encountered principal-delegation policies.
   - Principal-delegation policies cannot be applied conditionally.
2. After a principal-delegation policy has been applied, there is no automatic rollback. The user will not get any of the original roles unless other principal-delegation policies are specified.

Principal-delegation policies can easily lead to security misconfigurations, violations of the Principle of Least Privilege [29], and stability problems.

Identification of Role Requirements Access to any J2EE enterprise resource can be restricted with one or more roles. It should be noted that:

- When multiple roles are specified for a resource, the operation to apply to those roles is a logical `OR`, indicated with $\vee$. In order for a user to succeed in accessing that resource, that user must have been granted at least one of those roles.
- If multiple enterprise resources access each other, forming a chain of calls, then the user accessing the first of those resources will need to be granted all the roles necessary to access all the resources that will be invoked as a result of that first resource access. In this case, the operation to compute the set of roles needed by the user is a logical `AND`, indicated with $\wedge$. The system

administrator, however, should take into account that the J2EE container will enforce authorization only on intercomponent resource accesses. If an access-restricted resource is reached from its own component, the roles used to protect it should not be used in the computation.
- If a resource can be accessed from different components, each setting its own `run-as` role, a user accessing that resource will only have one of those `run-as` roles, depending on the execution path, because principal-delegation policies are mutually exclusive. Therefore, the logical operation to apply to `run-as` roles is eXclusive OR (`XOR`), indicated with $\oplus$.

This shows that in order to configure the security of a J2EE application correctly, system administrators must be able to infer which roles could be involved with the execution of an application at any given program point and evaluate potentially complex logical expressions of roles.

Inaccessible Resources In J2EE, an enterprise resource can be marked as *inaccessible* by explicitly configuring the deployment descriptor of the resource's component and listing that resource in an `exclude-list`. The intent is for that resource not to be accessed by any user, regardless of the user's roles. When that resource is accessed through an intercomponent call, the J2EE container will perform an authorization check and will prevent access to that resource, according to the intent. However, if the resource is accessed through an intracomponent call, the container will *not* perform any authorization check and the resource will be accessed in spite of the inaccessibility rule.

1.4 Summary of the Contributions of This Paper

The interprocedural-analysis model presented in this paper solves the problems identified in Section 1.3. Specifically, the model:

1. Distinguishes between intra- and intercomponent resource accesses and allows for a correct identification of the security requirements of an enterprise application
2. Allows for automatic identification of security flaws that could be generated by inappropriate use of principal-delegation policies
3. Determines all the roles involved in the execution of a J2EE application at any program point
4. Enables automatic evaluation of the logical expressions of roles that are generated at each program point
5. Detects cases in which resources that were intended to be inaccessible are inadvertently or deliberately accessed through intracomponent calls

2 J2EE Authorization Scenarios

J2EE is a component-based system. Components include enterprise beans [33], servlets [36], JavaServer Pages (JSP) programs [35], and J2EE clients. A component can contain security-sensitive resources that need to be access-restricted.

For example, if any of the business methods of an enterprise bean performs a security-sensitive operation, it is possible to restrict access to that method individually, specifying which role the user executing the application should be granted. Access to a servlet or JSP application can be restricted by limiting access to its Uniform Resource Locator (URL) or Uniform Resource Identifier (URI). Besides restricting access to a URL or URI, J2EE allows restricting access based on the HyperText Transfer Protocol (HTTP) methods `GET`, `POST`, `PUT`, `DELETE`, `HEAD`, `OPTIONS`, and `TRACE`. If a servlet or JSP application's URL or URI has been access-restricted and some HTTP methods have been specified in the access-restriction policy, the methods in the `HttpServlet` object corresponding to those HTTP methods will become access-restricted. For example, if the URL matching a servlet has been access-restricted and, along with that URL, the HTTP methods `GET` and `POST` have been restricted too, the J2EE container will perform access-control restrictions when the servlet's `doGet` and `doPost` methods are invoked through the `service` method as a result of a request from a client. However, if `doGet` or `doPost` are invoked from other methods within the same servlet, no authorization check will be performed. On the other hand, if a servlet's URL has been access restricted, but no HTTP method has been specified, then the access restriction is applied to all the HTTP methods.

When an application does not execute due to an authorization failure in a component, the J2EE container prevents the execution of that component. However, in a distributed component-based system, the container often does not provide the exact execution point at which the problem occurred. For large applications, manually tracking back the error across the distributed call stack can become a very difficult task. The algorithms described in Section 4 automate the process of call stack inspection for program understanding and security analysis in the presence of both intra- and intercomponent calls.

This remainder of this section presents two scenarios highlighting security flaws. The first scenario demonstrates how roles attached to different resources generate logical expressions of role requirements, which may be difficult to evaluate. The second scenario captures points of authorization failure. Both these scenarios emphasize the need for a tool that can not only detect security flaws, but also identify those points in the code where security faults may occur.

2.1 Flaws Due to Inaccurate Evaluations of Role Expressions

In the scenario of Figure 1, the application's entry point can only be accessed by users who have been granted role r_1. However, if a user were only granted role r_1, the application would not execute successfully. In fact, a more accurate analysis reveals that the roles required to access the application through that entry point are $r_1 \wedge (r_2 \vee r_3) \wedge (r_1 \vee r_5) = r_1 \wedge (r_2 \vee r_3)$. Notice that granting roles r_4 or r_5 would constitute a violation of the Principle of Least Privilege:

- The resource restricted with r_4 is only accessed from its own component. Thus, the container will never check that the user has been granted role r_4.
- Since r_1 is explicitly required when the entry point is accessed, role r_1 is sufficient for the user to pass the authorization test for $r_1 \vee r_5$.

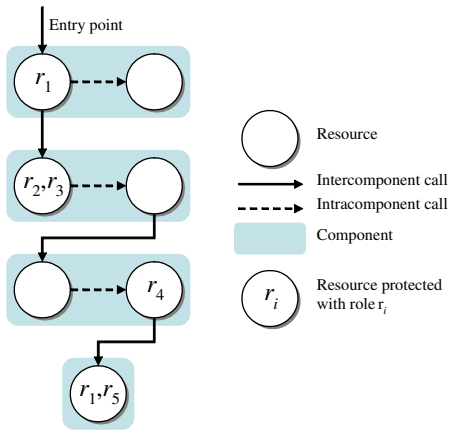

Fig. 1. Difficulties in Evaluating Role Requirements

2.2 Authorization Failures

The application depicted in the call graph of Figure 2 is destined to fail with an error message that, unfortunately, does not reveal where the failure occurred:

```
Application threw an exception: java.rmi.ServerException:
Nested exception is: java.rmi.AccessException: CORBA NO_PERMISSION 9998
```

Understanding the reason of the failure without an automated tool may be difficult and time consuming because it may require performing testing and manual inspection of code and deployment descriptors. If some components have been purchased from a third party, their source code may not be available. In other cases, the application's object code may have been machine generated and there may be no corresponding source code. In addition, J2EE promotes a separation of responsibilities among all the people who are involved with the lifecycle of an enterprise application [34]. Therefore, a deployer facing an authorization failure is typically not the developer who wrote the application and may not have any programming experience to understand where the failure came from, especially when the application is large and complex and accesses other applications through intercomponent calls.

Dynamic analysis is limited too because it relies on having a complete set of test cases covering all possible paths through the set of component resources in the application. In the absence of a complete set of test cases, authorization failures may remain undiscovered until the code is deployed in the enterprise.

A first pass through the code and the deployment descriptors associated with the application in Figure 2 shows that user `bob` has been granted the `Employee` role, which is exactly the role required to invoke `Servlet1`. What may not be so evident is that accessing `Servlet1` will lead to the invocation of `m5` on `Bean5`,

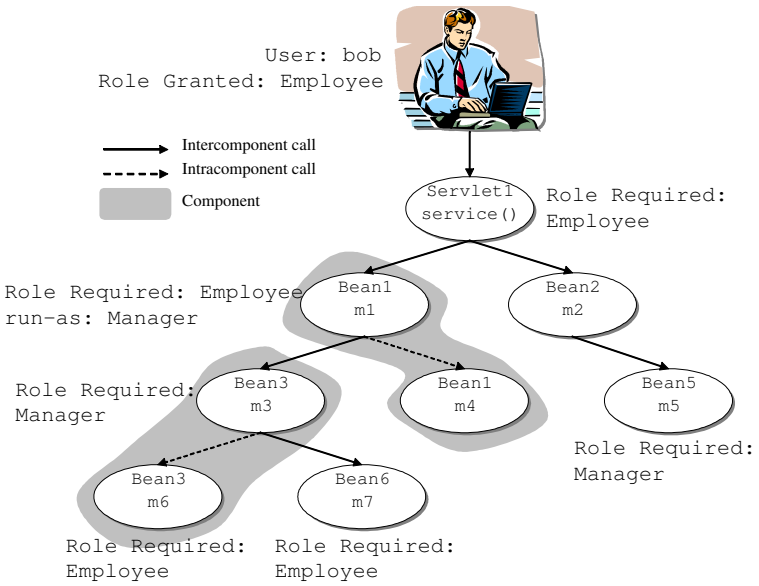

Fig. 2. J2EE Authorization Scenario

which has been access-restricted with the `Manager` role. Since the identity of the user, by default, is propagated with no changes, this invocation will fail because user `bob` does not have the required role.

Another point of failure is the invocation of `m7` on `Bean6`. When `Servlet1` invokes `m1` on `Bean1`, the authorization check succeeds because access to `m1` is restricted with the `Employee` role, which is exactly the role possessed by `bob` and propagated by the container up to this point. Next, `m1` invokes `m3` on `Bean3`. This resource requires the user to be a `Manager`. Fortunately, `Bean1` uses a delegation policy that overrides the roles of the user and forces all the subsequent downstream calls to be performed under the `Manager` role. Therefore, the authorization check for `m3` succeeds. The problem is that `Bean3` does not set the principal-delegation policy back to `Employee`, which is the role required to invoke `m7` on `Bean6`. Therefore, user `bob`, who was granted the role of `Employee` at the beginning, is now denied access to `m7` for not having the role of `Employee`.

Interestingly, but confusingly, the invocation of `m6` on `Bean3` does not cause an authorization failure even though access to `m6` also requires the `Employee` role. The reason is that both `m6` and its predecessor `m3` belong to the same component, `Bean3`, and the container does not perform authorization checks on intracomponent resource accesses.

3 Interprocedural Analysis Framework for RBAC

This section describes the interprocedural analysis framework used by the Enterprise Security Policy Evaluator (ESPE) tool.

3.1 Characteristics of the Call Graph and Pointer Analysis

ESPE is implemented on top of IBM's Data-Object Modeling and Optimization (DOMO) framework [9], a Java bytecode analysis system. ESPE uses DOMO's flow-insensitive pointer analysis with on-the-fly call graph construction. In particular, ESPE configures DOMO for a field-sensitive, intraprocedurally flow-sensitive, interprocedurally flow-insensitive, path-insensitive, and context-insensitive [28] pointer analysis. DOMO takes as input the bytecode of one or more J2EE applications. As output, it provides a call graph and an associated points-to graph modeling the execution of the program. The analysis precision can be customized. In fact, DOMO supports a number of object-oriented call-graph-construction and pointer-analysis algorithms, including:

1. Class Hierarchy Analysis (CHA) [5]
2. Rapid Type Analysis (RTA) [2]
3. Context-insensitive Control-Flow Analysis disambiguating between heap objects according to concrete types (0-CFA) [32]
4. Context-insensitive Control-Flow Analysis distinguishing heap objects based on allocation sites (0-1-CFA) [12], as in Andersen's analysis [1]

As a result, ESPE supports a range of cost/precision analysis trade-offs. To model with sufficient precision a security system such as Java 2, Standard Edition (J2SE), in which authorization checks are based on stack inspection, it has been shown [19, 24] that the interprocedural analysis should be able to disambiguate method calls based not just on the methods being invoked, but also on the receivers on which those methods are invoked and the parameters passed to those methods. However, to detect security problems generated by an RBAC policy on a J2EE application, a high level of context sensitivity is not critical:

- Authorization checks are not performed by a specific function, such as `Demand` in CLR or `checkPermission` in J2SE, but by the J2EE container.
- A restricted resource is not characterized by a parameter passed to a method. Rather, the restricted resource is the method itself.

Therefore, a context-insensitive analysis framework allows analyzing large J2EE applications in a shorter time, without losing in precision. Another important characteristic of the DOMO framework is its ability to distinguish between intra- and intercomponent calls. Consider for example an enterprise bean having remote interface `Bean2`, remote home interface `Bean2Home`, and enterprise bean class `Bean2Bean` [33]. Suppose that method `m1` in enterprise bean `Bean1Bean` calls remote method `m2` on `Bean2Bean`. For this to be possible, `m2` must be a method declared in `Bean2` and implemented in `Bean2Bean`, and a code similar to the following one must be embedded in `m1`:

```
Context initial = new InitialContext;
Object objref = initial.lookup("java:comp/env/ejb/Bean2");
Bean2Home bean2Home = (Bean2Home)
       PortableRemoteObject.narrow(objref, Bean2Home.class);
```

```
Bean2 bean2Objetc = bean2Home.create;
bean2Object.m2;
```

A traditional static-analysis engine would report an edge from a node representing a call to `m1` to a node representing an invocation of `m2` on an object of type `Bean2`, but there would be no edge leading to the actual implementation of `m2` in `Bean2Bean`. Since `Bean2`, which is an interface, does not implement `m2`, the control-flow graph would be truncated at that point. Conversely, DOMO creates an additional edge that links the declaration of `m2` in `Bean2` to the actual implementation of `m2` in `Bean2Bean`, as shown in Figure 3. Calls to methods declared in the remote home, local, and local home interfaces [33] are modeled in a very similar way.

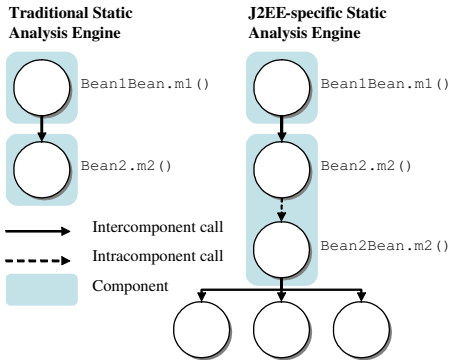

Fig. 3. Traditional vs. J2EE-specific Static Analysis Engines

3.2 Lattices of Roles

Let R indicate the set of all the roles associated with the program under analysis and let $\mathcal{P}(R)$ be the powerset of R. At each program point, the interprocedural-analysis model described in this paper can statically identify:

1. The roles required at that program point
2. The `run-as` roles that a user might possess at that program point, depending on the principal-delegation policies encountered in the execution path

This section presents two lattices [11] of roles used by the interprocedural analysis model to satisfy the two requirements above, respectively.

Role-set Lattice for Role-Requirement Analysis When access control on a resource a restricted with roles $r_1, r_2, \ldots, r_k \in R$ is enforced, the user who

initiated the call needs to show possession of at least one role $r_i, i \in \{1, 2, \ldots, k\}$, since the relation between those roles is a logical `OR`, as explained in Section 1.2. This can be modeled by associating each edge incident to the call-graph node representing resource a to the set of roles $\{r_1, r_2, \ldots, r_k\} \subseteq R$. Using a fixed-point iteration algorithm [17], role sets are then propagated backwards through the call graph-edges as will be described in Section 4.3. When a call-graph edge e is reached by two sets of roles $R_1 = \{r_1, r_2, \ldots, r_k\}, R_2 = \{q_1, q_2, \ldots, q_h\} \in \mathcal{P}(R)$, these two sets must be joined. The fact that edge e has been reached by both R_1 and R_2 means that the roles necessary to traverse e at run time are obtained by evaluating the logical expression $R_1 \wedge R_2 := (r_1 \vee r_2 \vee \ldots \vee r_k) \wedge (q_1 \vee q_2 \vee \ldots \vee q_h)$. Standard set union in $\mathcal{P}(R)$ would not model this requirement correctly. The solution is to map e to the set $\{R_1, R_2\} \in \mathcal{P}(\mathcal{P}(R))$.

Thus, the most appropriate lattice structure that can be used to perform dataflow of roles in this context is $(\mathcal{P}(\mathcal{P}(R)), \cap, \cup)$, where $\cap$ and $\cup$ are the set intersection and union operations on $\mathcal{P}(\mathcal{P}(R))$, respectively. The partial order induced by $\cap$ and $\cup$ is the set inclusion $\subseteq$ defined on $\mathcal{P}(\mathcal{P}(R))$. The *role-set lattice* $(\mathcal{P}(\mathcal{P}(R)), \cap, \cup)$ is finite because R is finite. Specifically, $|\mathcal{P}(\mathcal{P}(R))| = 2^{2^{|R|}}$. Therefore, this lattice is complete and has finite height, $\mathcal{H}(\mathcal{P}(\mathcal{P}(R))) = 2^{|R|}$. Its top and bottom elements are $\mathcal{P}(R)$ and $\varnothing$, respectively.

Role Lattice for Principal-Delegation Analysis It is important to determine statically the possible `run-as` roles with which a component will be executed. For example, let C be a component in the program under analysis and let C_1 and C_2 be two other components accessing C. If C_1 and C_2 specify `run-as` policies that set the user's role to r_1 and r_2, respectively, then C may be executed with $r_1 \oplus r_2$. This means that the role possessed by the user as a result of the principal-delegation policies set by C_1 and C_2 is either r_1 or r_2, depending on the execution path. It would be desirable to annotate the call graph with this information. This suggests that C could be annotated with set $\{r_1, r_2\}$, symbolizing that C can be executed under either r_1 or r_2. Detecting the set of `run-as` roles that can reach a component is achieved by performing a forward propagation of `run-as` role sets along the edges of the call graph representing the executing of the program under analysis, as will be described in Section 4.4.

Since a component's `run-as` policy can specify only one role, an appropriate lattice to model this dataflow of roles is $(\mathcal{P}(R), \cap, \cup)$, where $\cap$ and $\cup$ are now the set intersection and union operations in $\mathcal{P}(R)$, respectively. The partial order induced by $\cap$ and $\cup$ is the set inclusion $\subseteq$ defined on $\mathcal{P}(R)$. This lattice, called the *role lattice*, is finite. As such, it is complete and has finite height, $\mathcal{H}(\mathcal{P}(R)) = |R|$. The top and bottom elements of this lattice are R and $\varnothing$, respectively.

4 Enterprise Security Analysis

This section formalizes the interprocedural analysis implemented in the ESPE tool and discusses experimental results. The analysis can detect the security and stability problems that can arise in the following six situations:

1. The roles required at a program point according to the deployment descriptors are insufficient to run the application, as shown in Case 1 of Figure 4. If a user were granted only those roles, the container would generate an authorization failure during the execution of the application.
2. Some of the roles required at a program point according to the security policy are redundant, as in Case 2 of Figure 4. A user who is not granted any of those roles would still be able to access the application. Therefore, granting those roles would constitute a violation of the Principle of Least Privilege.
3. The role set by a principal-delegation policy is not sufficient to cover the role requirements at subsequent execution points, as in Case 3 of Figure 4. The container will generate a run-time authorization failure.
4. A component's principal-delegation policy is unnecessary because subsequent execution points do not specify any role requirements, as in Case 4 of Figure 4. Such principal-delegation policy constitutes a violation of the Principle of Least Privilege.
5. Access to a resource that was configured as inaccessible is attempted through an intercomponent call, as shown in Case 5 of Figure 4. This is a stability problem because the container will perform an authorization check and generate a run-time authorization failure, regardless of the user's roles.
6. Access to a resource that was configured as inaccessible is attempted through an intracomponent call, as in Case 6 of Figure 4. The container will not perform any authorization check and there will be no run-time authorization failure. However, this is a potential access-control violation since there exists a possibly undetected path reaching a resource that was intended to be inaccessible.

4.1 General Architecture of ESPE

ESPE contains two main components: a deployment-descriptor analyzer and a security-analysis engine. The input to ESPE consists of the object code of one or more J2EE applications and the deployment descriptors of those applications. The object code is analyzed by DOMO, which produces as output a call graph modeling the execution of the applications. ESPE analyzes the deployment descriptors to detect which resources have been access-restricted with roles, and which components use principal-delegation policies. Based on this information, the deployment-descriptor analyzer produces two mappings: one mapping associates enterprise resources with the roles necessary to access them; the other mapping associates each component with the `run-as` role specified by that component, if any. Next, ESPE analyzes both the call graph and the security mappings, and identifies security and stability problems such as those shown in Figure 4.

After the call graph has been built, as explained in Section 4.2, the ESPE engine performs two analyses: the Role-Requirement Analysis and the Principal-Delegation Analysis. These two analyses are described in Sections 4.3 and 4.4, respectively. Section 4.5 presents the experimental results obtained by executing ESPE on a number of test cases.

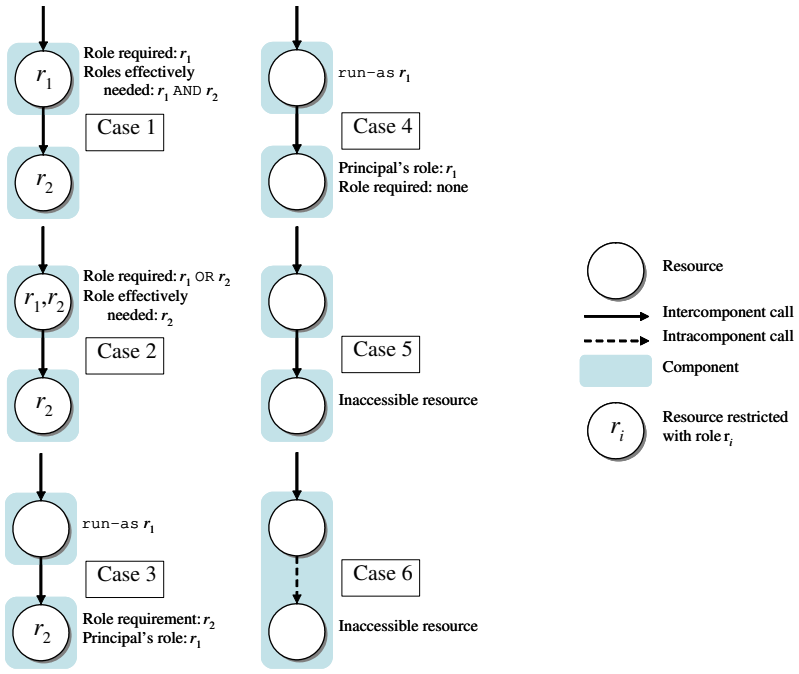

Fig. 4. Security and Stability Problems Detected by ESPE

4.2 Call Graph Construction

An ESPE analysis takes as input the interprocedural call graph $G = (N, E)$, representing the execution of one or more J2EE applications. Additionally, ESPE takes as input the deployment descriptors of those applications. The call graph is built using DOMO. For the call graph to precisely model the execution of a J2EE application, it is necessary to correctly identify the sets of entry points:

- For each servlet or JSP program, the entry points are the life-cycle methods, which are called by the servlet container as a result of invocations from client programs. These methods are `init`, `service`, and `destroy`.
- For each enterprise bean in the application, the entry points are all the methods declared in the remote, remote home, local, and local home interfaces.
- For each J2EE client, the only entry point is the `main` method.

It should be noted that the entry methods of the applications under analysis can have parameters, which for instance methods include the receiver object. At analysis time, the values and object sources of these parameters are unknown, since they are part of the client application, which will only be available at run time. For each parameter object, CHA is used to build the class hierarchy rooted at the parameter's declared type. When a call back from that parameter object

is encountered, DOMO can model it by looking for possible implementations of the invoked method in the class hierarchy.[3]

4.3 Role-Requirement Analysis

This section presents the RBAC interprocedural analysis performed by ESPE to compute the actual access-control requirements for all the resources within a J2EE program and statically detect security and stability problems due to improper role assignments. The analysis described in this section is called *Role-Requirement Analysis*.

Mapping of Call-Graph Nodes to Required Roles The first step of the Role-Requirement Analysis algorithm is to identify which resources in the application are access-restricted and the roles required to access them. In the context of this work, a *resource* is a J2EE application method. If a resource is reachable from the program entry points, then it will be represented by a node of G. Mapping nodes of G to roles is a straightforward operation that requires scanning the deployment descriptors of all the application components. Once the access-restricted resources have been identified along with the roles necessary to access them, a simple call-graph traversal allows locating the nodes that represent those resources, and mapping them to the roles necessary to access those resources. A function $\nu : N \rightarrow \mathcal{P}(R)$ is therefore constructed that maps call graphs nodes to sets of roles. If $n \in N$ and $\nu(n) = \{r_1, r_2, \ldots, r_k\}$, this means that to invoke the resource represented by n the user must have been granted $r_1 \vee r_2 \vee \ldots \vee r_k$. While the graph is being traversed, each intercomponent edge $e = (m, n) \in E$ such that $\nu(n) \neq \varnothing$ is identified.

Interprocedural Analysis for Role-Requirement Detection The next step in the algorithm is to identify which roles are effectively required to invoke a resource. The scenario depicted in Figure 1 shows that when multiple restricted resources access each other, forming an invocation sequence, then the user accessing the first of those resources will have to be granted not just the role needed to access the first of those resources, but all the roles necessary to invoke all the resources that are subsequently accessed through intercomponent calls. Understanding the roles effectively required to access a resource a in an invocation sequence requires detecting all the roles needed to access all the resources directly and indirectly reachable from a through intercomponent accesses. This computation can be performed through a *reverse role-set propagation* across the edges of G.

The reverse role-set propagation in the call graph is initialized by mapping each intercomponent edge $e = (m, n) \in E$ leading to an access-restricted resource

[3] This solution may be overly conservative if the declared type has many subclasses, and can be unsound if the declared type is non-final and the program being analyzed is incomplete [28].

node $n \in N$ to the singleton $\{\nu(n)\} \in \mathcal{P}(\mathcal{P}(R))$. For example, the intercomponent edge connecting the two nodes in Case 2 of Figure 4 would be mapped to role-set requirement $\{\{r_2\}\}$, while the entry edge would be mapped to role-set requirement $\{\{r_1, r_2\}\}$. This concludes the initialization phase.

Next, each of these edges propagates the role-set lattice element associated with it to its predecessor edges, regardless of whether those predecessor edges are inter- or intracomponent edges. The operations performed on the elements of $\mathcal{P}(\mathcal{P}(R))$ associated with each edge is set union, $\cup$. Every time the role-set requirements associated with an edge change as a result of the propagation being performed, that edge must in turn propagate its role requirements to its predecessor edges.

Excluding intracomponent edges from the initialization phase—even when these lead to access-restricted resources, such as the resource restricted with role r_4 in Figure 1—correctly models the absence of intracomponent authorization checks at run time.

This entire process can be described in terms of dataflow equations [17, 20]. For each edge $e = (m, n) \in E$, let $\mathrm{Gen_R}(e)$ and $\mathrm{Kill_R}(e)$ be the subsets of $\mathcal{P}(\mathcal{P}(R))$ corresponding to the role sets *generated* and *killed* by e, respectively:

- If e is an intercomponent edge, then $\mathrm{Gen_R}(e) := \{\nu(n)\}$. If e is an intracomponent edge, then $\mathrm{Gen_R}(e) := \varnothing$. Therefore, $\mathrm{Gen_R}(e) \neq \varnothing$ if and only if e is an intercomponent edge and the resource represented by n has been access-restricted with some roles.
- $\mathrm{Kill_R}(e) := \varnothing$ since in this dataflow analysis role sets are never subtracted.

This defines two functions, $\mathrm{Gen_R}, \mathrm{Kill_R} : E \to \mathcal{P}(\mathcal{P}(R))$ that can be used to define the following standard *dataflow equation system*:

$$\mathrm{Out_R}(e) = \mathrm{Gen_R}(e) \cup (\mathrm{In_R}(e) \setminus \mathrm{Kill_R}(e)) \tag{1}$$

$$\mathrm{In_R}(e) = \bigcup_{f \in \tilde{\Gamma}^+(e)} \mathrm{Out_R}(f) \tag{2}$$

for every edge $e = (m, n) \in E$, where:

- $\mathrm{Out_R}(e)$ and $\mathrm{In_R}(e)$ are the elements of $\mathcal{P}(\mathcal{P}(R))$ corresponding to the roles *propagated* from and *reaching* e, respectively
- $\setminus : \mathcal{P}(\mathcal{P}(R)) \times \mathcal{P}(\mathcal{P}(R)) \to \mathcal{P}(\mathcal{P}(R))$ is the set difference operation in $\mathcal{P}(\mathcal{P}(R))$
- $\tilde{\Gamma}^+ : E \to \mathcal{P}(E)$ is defined by $\tilde{\Gamma}^+(e) := \{f = (u, v) \in E : u = n\}$

The recursive computation of the functions $\mathrm{In_R}, \mathrm{Out_R} : E \to \mathcal{P}(\mathcal{P}(R))$ performed by resolving Equations (1) and (2) converges to a fixed point after at most $\mathcal{O}(|E| \cdot \mathcal{H}(\mathcal{P}(\mathcal{P}(R)))) = \mathcal{O}(|E| \cdot 2^{|R|})$ iterations [11]. Upon completion, the reverse role-set propagation yields a mapping of edges to role requirements. Figure 5 shows a call graph annotated with role-requirement information at the initialization and termination phases of the Role-Requirement Analysis algorithm.

The mapping of intercomponent edges to role requirements can be used to compute which roles are actually necessary to invoke an access-restricted

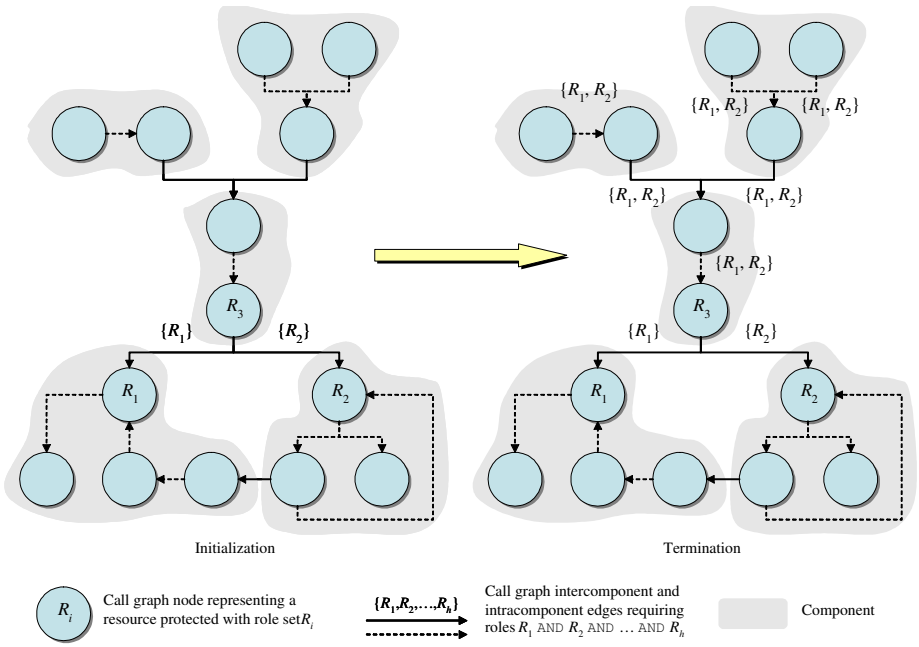

Fig. 5. Role-Requirement Analysis Initialization and Termination

resource, and this information can in turn be used to detect stability problems and security flaws. Specifically, the call graph can be annotated with a function $\Lambda : E \rightarrow \mathcal{P}(\mathcal{P}(R))$ that associates each edge $e \in E$ to the element $\Lambda(e) \in \mathcal{P}(\mathcal{P}(R))$ defined by $\Lambda(e) := \mathrm{Out_R}(e)$.

If $e = (m, n)$ and $\Lambda(e) = \{R_1, R_2, \ldots, R_h\}$, with $R_i = \{r_{i1}, r_{i2}, \ldots r_{ik_i}\}, \forall i \in \{1, 2, \ldots, h\}$, then a user accessing the resource represented by n from the one represented by m needs to be granted a set of roles obtained by evaluating the following expression:

$$\tilde{\Lambda}(e) := \bigwedge_{i=1}^{h} R_i = \bigwedge_{i=1}^{h} (\bigvee_{j=1}^{k_i} r_{ij}) \tag{3}$$

Stability- and Security-Problem Detection Figure 6 shows the initialization and termination of the reverse role-propagation algorithm applied to the scenario of Figure 1. It can be seen that the roles effectively needed by a user to access the application through its entry point are $r_1 \wedge (r_2 \vee r_3) \wedge (r_1 \vee r_5) = r_1 \wedge (r_2 \vee r_3)$. If only role r_1 were granted, like a superficial parsing of the deployment descriptor would seem to suggest, the container would generate an authorization failure. This scenario shows how the reverse role-set propagation can be used to detect stability problems similar to Case 1 in Figure 4 and avoid authorization failures at run time.

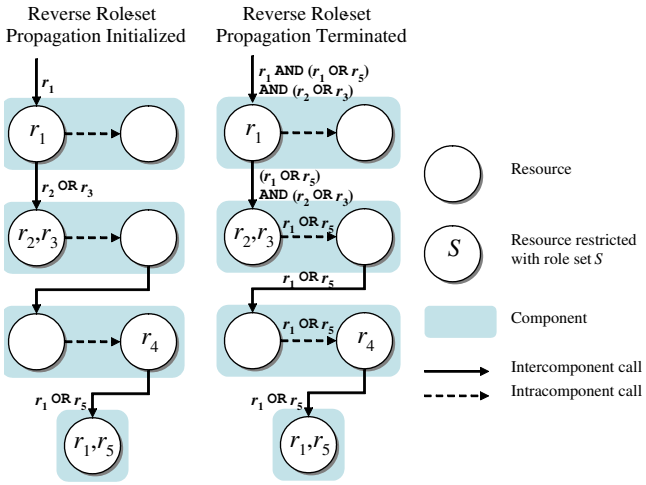

Fig. 6. Reverse Role-Set Propagation Algorithm

In other cases, after the reverse role-set propagation algorithm has terminated, comparing the roles necessary to invoke the resource represented by a node to the roles associated with the outgoing edges can help preventing violations of the Principle of Least Privilege. For example, in the scenario of Case 2 in Figure 4, the roles necessary to invoke the entry point are $r_1 \vee r_2$. However, at the end of the reverse role-set propagation, the only edge leaving the root node will be mapped to role r_2. The only role effectively needed by a user to access the application through the entry point is r_2. Granting the user role r_1 alone would cause an authorization failure, while granting the user $r_1 \wedge r_2$ would constitute a violation of the Principle of Least Privilege because r_1 is redundant.

The detection for both these problems can be formalized as follows. Let $n_0 \in N$ be any call-graph node, with $\nu(n_0) = R_0$ and $\Gamma^+(n_0) = \{n_1, n_2, \ldots, n_h\}$, where $\Gamma^+ : N \rightarrow \mathcal{P}(N)$ is defined by $\Gamma^+(n) := \{m \in N : (n, m) \in E\}$. Let $e_1 = (n_0, n_1), e_2 = (n_0, n_2), \ldots, e_h = (n_0, n_h) \in E$ be n_0's outgoing edges, with $\Lambda(e_i) = \{R_{i1}, R_{i2}, \ldots, R_{ik_i}\}, \forall i \in \{1, 2, \ldots, h\}$, as shown in Figure 7.

Let $\tilde{\nu}(n_0)$ be the subset of R_0 defined by:

$$\tilde{\nu}(n_0) = R_0 \cap \bigcap_{i=1}^{h} \bigcap_{j=1}^{k_i} R_{ij}$$

Set $\tilde{\nu}(n_0)$ can help detecting whether a stability or security problem can occur at run time when a user accesses the resource represented by n_0 through an intercomponent edge e_0:

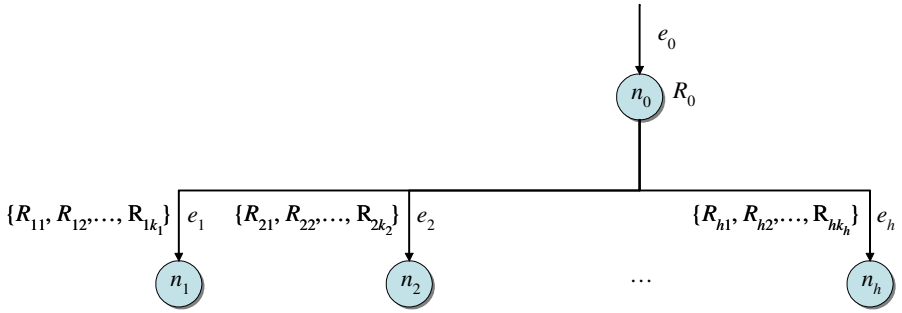

Fig. 7. Detection of Redundant Roles

1. If $\tilde{\nu}(n_0) \neq \varnothing$, then the user should only be granted one of the roles in $\tilde{\nu}(n_0)$. This means that, if $\tilde{\nu}(n_0) = \{q_1, q_2, \ldots, q_t\}$, the user accessing the resource represented by n_0 should be granted $q_1 \oplus q_2 \oplus \ldots \oplus q_t$. Specifically:
   - Any additional role in $R_0 \setminus \tilde{\nu}(n_0)$ is unnecessary. Granting it to the user would constitute a violation of the Principle of Least Privilege.
   - Granting the user none of the roles in $\tilde{\nu}(n_0)$ is potentially insufficient. Run-time authorization failures are possible.
2. If $\tilde{\nu}(n_0) = \varnothing$, then simply granting the user any of the roles in R_0, as the deployment descriptor would seem to suggest, might be insufficient and cause run-time authorization failures. In this case, for no authorization failure to occur, the user should be granted a logical combination of roles obtained by evaluating $\tilde{\Lambda}(e_0)$ as defined by Equation (3).

This analysis should be performed for any node n_0 corresponding to an access-restricted resource.

Security Flaws Due to Inaccessible Resources ESPE encodes inaccessible resources as if they were available only to users with an `Inaccessible` role. If a node in the call graph is mapped to the `Inaccessible` role, then:

1. All the intercomponent edges incident into that node are indications of potential stability problems. Case 5 in Figure 4 is an example of this situation.
2. All the intracomponent edges incident into that node are indications of potentially unintended security violations. Case 6 in Figure 4 is an example of this situation.

Using the reverse role-propagation algorithm, an `Inaccessible` role requirement can be propagated in the call graph from an inaccessible resource, just like any other role requirement. This allows detecting possible execution paths leading to resources that were intended to be inaccessible.

4.4 Principal-Delegation Analysis

If principal-delegation policies in an application's security configuration are detected, ESPE performs an additional analysis, called *Principal-Delegation Analysis*, to identify security misconfigurations that could lead to violations of the Principle of Least Privilege or stability problems.

Mapping of Call-Graph Nodes to `run-as` Roles The first step of the Principal-Delegation Analysis consists of mapping call-graph nodes to `run-as` roles. The application's deployment descriptors are scanned, and each component is mapped to the singleton containing the `run-as` role it sets, or to $\varnothing$ if it does not set any `run-as` role.

Next, a simple call-graph traversal allows locating the nodes that represent resources in the components that have been identified, and mapping those nodes to the `run-as` roles of their components. A function $\rho : N \to \mathcal{P}(R)$ is therefore constructed that associates call graphs nodes with singletons of roles. If $n \in N$ and the resource represented by n is in a component C with a principal-delegation policy that sets the `run-as` role to r, then $\rho(n) := \{r\}$, otherwise $\rho(n) := \varnothing$. While the graph is traversed, the subset E' of E containing all the intercomponent edges $e = (m, n) \in E$ such that $\rho(m) \neq \varnothing$ is identified.

Interprocedural Analysis for `run-as` Role Propagation The next phase of the Principal-Delegation Analysis is to perform a *principal-delegation role propagation*, which is a forward propagation similar to the algorithm that computes reaching definitions [20]. At the end of the principal-delegation role propagation, each intercomponent edge e in the call graph will be mapped to a set containing all the `run-as` roles that can reach e in the call graph.

The principal-delegation role propagation is a fixed-point iteration [17, 20] over E, initialized with E'. In the propagation process, each intra- and intercomponent edge propagates the roles it has accumulated to its successor edges. At run time, the operation performed on the `run-as` roles associated with each edge is $\oplus$, because each edge cannot be traversed with more than one `run-as` role. At analysis time, however, since it may be important to keep track of all the possible `run-as` roles that can reach any edge at run time, it is convenient to represent each role as a singleton, and then perform set unions. Every time the role set associated with an edge changes as a result of a propagation, that edge must in turn propagate its role requirements to its successor edges.

When an edge e is encountered as a result of a `run-as` propagation from a predecessor, the incoming `run-as` role set is unioned with the role set already computed for e up to that point. However, if $e = (m, n) \in E'$ and $\rho(m) = r$, then the singleton $\{r\}$ overrides, or *kills* [20], whatever set was computed up to that point for e.

Excluding intracomponent edges from the initialization phase—even when these originate from components having principal-delegation policies—correctly models the requirement that a component's principal-delegation policy overrides the role of a user only when that component invokes other components.

This entire process can be described in terms of dataflow equations [17, 20]. For each edge $e = (m, n) \in E$, let $\mathrm{Gen_r}(e)$ and $\mathrm{Kill_r}(e)$ be the subsets of $\mathcal{P}(R)$ corresponding to the roles *generated* and *killed* by e, respectively:

- If e is an intercomponent edge, then $\mathrm{Gen_r}(e) = \{\rho(m)\}$. If e is an intracomponent edge, then $\mathrm{Gen_r}(e) = \varnothing$. Therefore, $\mathrm{Gen_r}(e) \neq \varnothing$ if and only if e is an intercomponent edge and the resource represented by m belongs to a component with a non-empty principal-delegation policy.
- If $e \in E'$, then $\mathrm{Kill_r}(e) = R$. Otherwise, e is either an intracomponent edge or an intercomponent edge such that $\rho(m) = \varnothing$, and $\mathrm{Kill_r}(e) = \varnothing$.

This defines two functions, $\mathrm{Gen_r}, \mathrm{Kill_r} : E \to \mathcal{P}(R)$, based on which it is possible to define the following *dataflow equation system*:

$$\mathrm{Out_r}(e) = \mathrm{Gen_r}(e) \cup (\mathrm{In_r}(e) \setminus \mathrm{Kill_r}(e)) \tag{4}$$

$$\mathrm{In_r}(e) = \bigcup_{f \in \tilde{\Gamma}^-(e)} \mathrm{Out_r}(f) \tag{5}$$

for every $e = (m, n) \in E$, where:

- $\mathrm{Out_r}(e)$ and $\mathrm{In_r}(e)$ are the elements of $\mathcal{P}(R)$ corresponding to the `run-as` roles *propagated* from and *reaching* e, respectively
- $\setminus : \mathcal{P}(R) \times \mathcal{P}(R) \to \mathcal{P}(R)$ is the standard set difference operation in $\mathcal{P}(R)$
- $\tilde{\Gamma}^- : E \to \mathcal{P}(E)$ is defined by $\tilde{\Gamma}^-(e) = \{f = (u, v) \in E : v = m\}$

The recursive computation of the functions $\mathrm{In_r}, \mathrm{Out_r} : E \to \mathcal{P}(\mathcal{P}(R))$ performed by resolving Equations (4) and (5) converges to a fixed point after at most $\mathcal{O}(|E| \cdot \mathcal{H}(\mathcal{P}(R))) = \mathcal{O}(|E| \cdot |R|)$ iterations [11]. Upon completion, the principal-delegation role propagation yields a mapping of edges to `run-as` role sets, each set representing the `run-as` roles that can reach the associated edge. Figure 8 shows a call graph annotated with principal-delegation information at the initialization and termination phases of the Principal-Delegation Analysis algorithm.

Specifically, the call graph can be annotated with a function $\Omega : E \to \mathcal{P}(R)$ that associates each edge $e \in E$ to the element $\Omega(e) \in \mathcal{P}(R)$ defined by $\Omega(e) := \mathrm{Out_r}(e)$. In particular, the mapping of intercomponent edges to sets of `run-as` roles can be used to understand which principal-delegation roles can propagate when an access-restricted resource is invoked, and detect stability problems and security inconsistencies.

Stability Problems Due to Incorrect Principal-Delegation Policies Let $e = (m, n) \in E$ be an intercomponent edge and let $\Omega(e) = \{r_1, r_2, \ldots, r_k\} \in \mathcal{P}(R)$ be the set of `run-as` roles that can have reached e in the call graph. From a logical point of view, this means that the role expression associated with e is $r_1 \oplus r_2 \oplus \ldots \oplus r_k$, or that, when e is traversed at run time, the user initiating the resource access will have exactly one of the roles in the set $\{r_1, r_2, \ldots, r_k\}$, depending on the execution path prior to traversing e. If n is access restricted with the set of roles $\nu(n) = \{q_1, q_2, \ldots, q_h\}$, this means that to access the resource

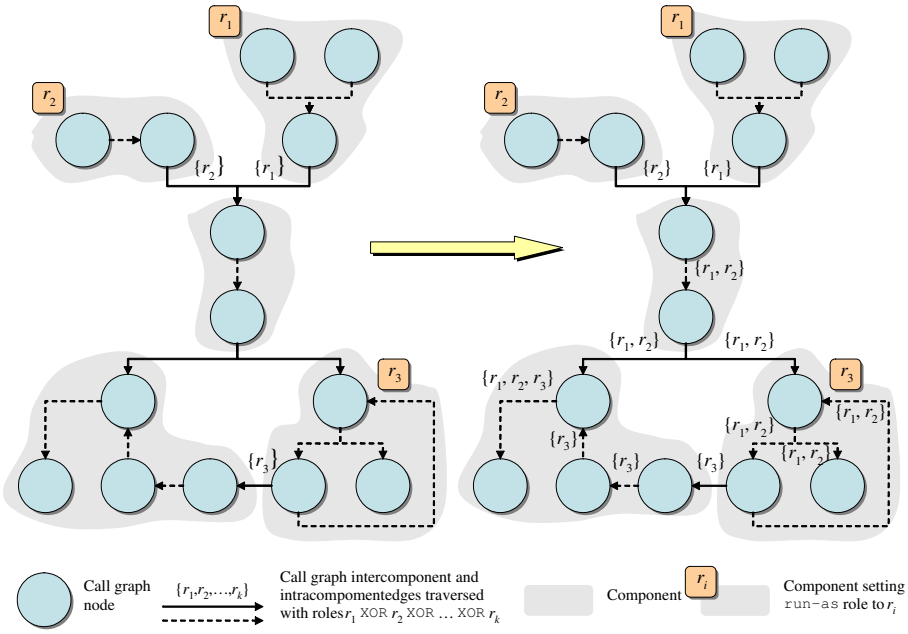

Fig. 8. Principal-Delegation Analysis Initialization and Termination

represented by n the user will have to be granted $q_1 \vee q_2 \vee \ldots \vee q_h$. This scenario is graphically represented in Figure 9.

If $\nu(n) \neq \varnothing$, the security configuration shown in Figure 9 will not generate an authorization failure and a consequent stability problem for the application if and only if $\Omega(e) \neq \varnothing$ and $\Omega(e) \subseteq \nu(n)$. In other words, it must be $k \geq 1$ and $\{r_1, r_2, \ldots, r_k\} \subseteq \{q_1, q_2, \ldots, q_h\}$. An example of this scenario is Case 3 in Figure 4.

Security Flaws Due to Redundant Principal-Delegation Policies The scenario depicted in Figure 9 shows also how the Principal-Delegation Analysis can be used to detect security flaws. Let $e = (m, n) \in E$ be an intercomponent edge and let $\Omega(e) = \{r_1, r_2, \ldots, r_k\}$ be the `run-as` role set associated with e upon termination of the `run-as` role propagation algorithm. If $k \geq 1$ and n is not access-restricted at all ($h = 0$, according to the notation introduced above, or $\nu(n) = \varnothing$), then this is a potential violation of the Principle of Least Privilege. An example of this scenario is Case 4 in Figure 4.

4.5 Experimental Results

ESPE runs as a stand-alone application and can be launched on top of a J2SE V1.4 run-time environment from the command line or from Eclipse V3.1 [7]. It

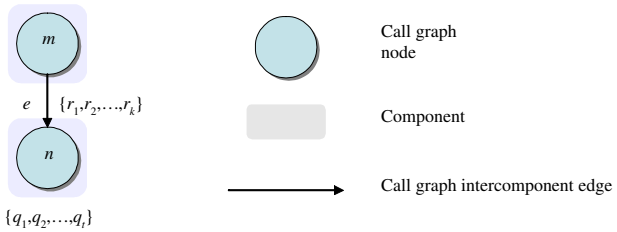

Fig. 9. Principal-Delegation Analysis Scenario

can analyze J2EE V1.3 and V1.4 deployed applications packaged in Enterprise ARchive (EAR) files or separate Java ARchive (JAR) and Web ARchive (WAR) files [34]. These files contain the object code and deployment descriptors of one or more applications. Source code is unnecessary since ESPE analyzes object code. ESPE analyzes standard J2EE applications regardless of the platform vendor.

All the library files used by the applications at run time must be part of the analysis to allow ESPE to provide sound results. Currently, for the reports provided by ESPE to be meaningful, the deployment descriptors of the applications being analyzed must contain security configuration information, including the definition of at least one role and an RBAC policy. In the future, ESPE will provide useful results even when the applications it analyzes have not been configured for security, or the configuration is not complete yet. In such cases, ESPE will still be able to report what are the dependencies between different resources that can potentially be protected, and this information can in turn be used by developers (rather than just deployers) to write applications without security inconsistencies or to figure out role requirements in advance. This feature will be especially useful in aspect-oriented J2EE environments such as JBoss Application Server [16], in which developers can directly specify role requirements.

This section summarizes the results of the analyses performed on a number of J2EE applications: `PetStore` [22], `Bookstore` [6], `EnrollerApp` [13], `SavingsAcc` [13], and anonymous production-level application `A`. The results reported in Table 1 are from running ESPE on an IBM T40 ThinkPad with an Intel Pentium M 1.6 GHz processor, 1 GB of Random Access Memory (RAM), and Microsoft Windows XP SP2 operating system. ESPE was run inside Eclipse V3.1 using a J2SE V1.4.2_08 run-time environment.

From a security point of view, except for the commercial application `A`, none of the other benchmarks came with predefined roles, the reason being that applications downloaded from the Internet typically do not come with a security configuration. Assigning the security configuration to an application is a task that the J2EE Specification delegates to the deployer [34] and must be done based on the system on which the application will run. Before analyzing those other applications, it was therefore necessary to manually deploy those applications, define relevant roles, and use those roles to restrict access to security-sensitive resources. We assigned roles based on the introspection performed on

Application	Size	Call Graph		Time (sec.)		Memory	Roles	Problems
	(KB)	Nodes	Edges	DOMO	ESPE	(MB)		
`PetStore`	1,282	6,465	2,336	124.69	1.89	117	4	6
`Bookstore`	359	16,269	86,448	1,899.94	3.97	162	3	4
`EnrollerApp`	15	2,212	10,060	63.42	0.36	220	4	2
`SavingsAcc`	10	2,164	9,799	62.77	0.21	227	4	0
`A`	2,580	618	1,007	72.23	1.24	239	4	8

Table 1. Characteristics of the Applications Analyzed

the applications by Sun Microsystems' Deployment Tool for Java 2 Platform Enterprise Edition 1.4, without any specific knowledge of the applications' source code—similar to what a system administrator would do.

ESPE detected several security problems on the analyzed applications. For `EnrollerApp`, both problems were due to an inappropriate principal-delegation policy. In `Bookstore`, two of the reported problems were generated by an inaccessible resource. The remaining problems were due to insufficient role assignments. Only one false alarm was detected (in `PetStore`) using DOMO's implementation of RTA, which suggests that RTA is sufficient for ESPE's authorization analyses.

The absence of an application's source code makes it very hard to track down arbitrarily long calling paths within the application and the underlying libraries. Therefore, without ESPE, detecting the security problems reported in Table 1 would have been extremely difficult.

5 Related Work

Ferraiolo and Kuhn introduced RBAC in 1992 [8]. Subsequently, Sandhu, Coyne, Feinstein, and Youman described mechanisms for constructing and analyzing RBAC models and implementations [30]. More recently, Schaad and Moffett [31] employed the Alloy specification language [14] to model RBAC. In particular, they used the Alloy constraint analyzer Alcoa [15] to verify key characteristics of the model, such as separation of duties assigned to roles.

Several static and dynamic analysis techniques have been suggested in the area of Web applications, but these works do not provide a significant support for distributed applications, nor do they deal with security issues. For example, Ricca and Tonella [27] propose a Unified Modeling Language (UML) model for Web applications, but their model deals primarily with structural testing of interactive features of Web applications, such as HyperText Markup Language (HTML) forms. Brucker and Wolff [3] introduce a mechanism for dynamic analysis of distributed-component systems using the Object Constraint Language (OCL) [37] of the UML standard to formalize component specifications.

An EJB object's confinement can be breached when a direct reference to the EJB object is returned to a client. Such a reference could allow the client program to access security-sensitive fields or invoke enterprise bean business methods without going through the EJB interfaces. In particular, this implies

that any access-control restriction on those methods could be bypassed. The purpose of the work of Clarke, Richmond, and Noble [4] is to enforce confinement of EJB objects. They propose coding guidelines that, if observed, prevent confinement breaches. Additionally, they describe a straightforward code inspection algorithm that checks for violations of those guidelines in enterprise bean programs. However, their analysis only considers EJB components; servlets and JSP programs are not analyzed, and RBAC issues are not taken into account.

Finally, Naumovich and Centonze [21] describe how the J2EE authorization model, which is designed to restrict access to EJB methods, can be extended to restrict access to the data handled by those methods. They use points-to analysis [28] to identify which EJB methods access security-sensitive data and the mode of access (read and/or write). Subsequently, RBAC on that data can be achieved by enforcing RBAC on the methods accessing the data. Their work can also be used to find access-control inconsistencies whereby two EJB methods accessing the same data in the same mode should be restricted with the same roles.

In addition to RBAC [25], Java offers a low-level access control mechanism to protect static resources, such as the file system, network, and operating system properties [26]. Access control decisions are based on the origin of the code and/or the principal running the code. Both static and dynamic analysis techniques are employed in modeling security and authorization. Much of this work has been applied to eliminate or minimize redundant authorization tests, or define alternatives to the current approach to defining authorization points within code. This work is specifically designed for J2SE authorization problems and it assumes that call graph algorithms are available to translate the theoretical approach into a practical implementation. However, many of the well-known call-graph-construction and dataflow algorithms [28] do not correctly model J2EE intercomponent calls. Additionally, in J2EE, access control is enforced differently from the way it is enforced in J2SE. Previous work on J2SE access control assumes that authorization checks are obtained by passing a `Permission` object to a `checkPermission` function, but this assumption cannot be made for J2EE because access restrictions are enforced by the container in a vendor-specific way.

6 Acknowledgments

Thanks to Dr. Stephen J. Fink, the lead architect of IBM Research's DOMO static analyzer, for his invaluable contributions to the implementation of ESPE.

References

1. Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, May 1994.
2. David F. Bacon and Peter F. Sweeney. Fast Static Analysis of C++ Virtual Function Calls. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 324–341, San Jose, CA, USA, 1996. ACM Press. Also published in ACM SIGPLAN Notices 31(10).

3. Achim D. Brucker and Burkhart Wolff. Testing Distributed Component Based Systems Using UML/OCL. In K. Bauknecht, W. Brauer, and Th. Mück, editors, *Proceedings of Informatik 2001*, volume 1 of *Tagungsband der GI/ÖCG Jahrestagung*, pages 608–614, Vienna, Austria, 2001. Österreichische Computer Gesellschaft.
4. Dave Clarke, Michael Richmond, and James Noble. Saving the World from Bad Beans: Deployment-Time Confinement Checking. In *Proceedings of the 18th annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 374–387, Anaheim, CA, USA, 2003. ACM Press.
5. Jeffrey Dean, David Grove, and Craig Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, pages 77–101, Aarhus, Denmark, August 1995. Springer-Verlag.
6. Harvey M. Deitel, Paul J. Deitel, and Sean E. Santry. *Advanced Java 2 Platform: How to Program*. Prentice Hall, Upper Saddle River, NJ, USA, September 2001.
7. Eclipse Project, `http://www.eclipse.org`.
8. David F. Ferraiolo and D. Richard Kuhn. Role-Based Access Controls. In *Proceedings of the 15th NIST-NCSC National Computer Security Conference*, pages 554–563, Baltimore, MD, USA, October 1992.
9. Stephen J. Fink, Julian Dolby, and Logan Colby. Semi-Automatic J2EE Transaction Configuration. Technical Report RC23326, IBM Corporation, Thomas J. Watson Research Center, Yorktown Heights, NY, USA, 2004.
10. Adam Freeman and Allen Jones. *Programming .NET Security*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, June 2003.
11. George Grätzer. *General Lattice Theory*. Birkhäuser, Boston, MA, USA, second edition, January 2003.
12. David Grove and Craig Chambers. A Framework for Call Graph Construction Algorithms. *ACM Trans. Program. Lang. Syst.*, 23(6):685–746, November 2001.
13. Sun Microsystems, J2EE 1.4 Tutorial, `http://java.sun.com/j2ee/1.4/download.html#tutorial/`.
14. Daniel Jackson. Alloy: a Lightweight Object Modelling Notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.
15. Daniel Jackson, Ian Schechter, and Hya Shlyahter. Alcoa: The Alloy Constraint Analyzer. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 730–733, Limerick, Ireland, 2000. ACM Press.
16. JBoss, Inc., `http://www.jboss.com`.
17. Gary A. Kildall. A Unified Approach to Global Program Optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 194–206, Boston, MA, USA, 1973. ACM Press.
18. Larry Koved, Anthony J. Nadalin, Nataraj Nagaratnam, Marco Pistoia, and Theodore Shrader. Security Challenges for Enterprise Java in an E-business Environment. *IBM Systems Journal*, 40(1):130–152, 2001.
19. Larry Koved, Marco Pistoia, and Aaron Kershenbaum. Access Rights Analysis for Java. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 359–372, Seattle, WA, USA, November 2002. ACM Press.
20. Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, June 1997.
21. Gleb Naumovich and Paolina Centonze. Static Analysis of Role-Based Access Control in J2EE Applications. *SIGSOFT Software Engineering Notes*, 29(5):1–10, September 2004.

22. Sun Microsystems, Java PetStore, `http://java.sun.com/developer/releases/petstore/`.
23. Marco Pistoia. *A Unified Mathematical Model for Stack- and Role-Based Authorization Systems*. PhD thesis, Polytechnic University, Brooklyn, NY, USA, May 2005.
24. Marco Pistoia, Robert J. Flynn, Larry Koved, and Vugranam C. Sreedhar. Interprocedural Analysis for Privileged Code Placement and Tainted Variable Detection. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, pages 362–386, Glasgow, Scotland, UK, July 2005. Springer-Verlag.
25. Marco Pistoia, Nataraj Nagaratnam, Larry Koved, and Anthony Nadalin. *Enterprise Java Security*. Addison-Wesley, Reading, MA, USA, February 2004.
26. Marco Pistoia, Duane Reller, Deepak Gupta, Milind Nagnur, and Ashok K. Ramani. *Java 2 Network Security*. Prentice Hall PTR, Upper Saddle River, NJ, USA, second edition, August 1999.
27. Filippo Ricca and Paolo Tonella. Analysis and Testing of Web Applications. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 25–34, Toronto, ON, Canada, 2001. IEEE Computer Society.
28. Barbara G. Ryder. Dimensions of Precision in Reference Analysis of Object-Oriented Languages. In *Proceedings of the 12th International Conference on Compiler Construction*, pages 126–137, Warsaw, Poland, April 2003. Invited Paper.
29. Jerome H. Saltzer and Michael D. Schroeder. The Protection of Information in Computer Systems. In *Proceedings of the IEEE*, volume 63, pages 1278–1308, September 1975.
30. Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-Based Access Control Models. *Computer*, 29(2):38–47, February 1996.
31. Andreas Schaad and Jonathan D. Moffett. A Lightweight Approach to Specification and Analysis of Role-Based Access Control Extensions. In *Proceedings of the 7th ACM Symposium on Access Control Models and Technologies*, pages 13–22, Monterey, CA, USA, 2002. ACM Press.
32. Olin Shivers. Control Flow Analysis in Scheme. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, June 1988.
33. Enterprise JavaBeans™ Specification, `http://java.sun.com/products/ejb/`.
34. Java™ 2 Platform Enterprise Edition Specification, `http://java.sun.com/j2ee`.
35. JavaServer Pages™ Specification, `http://java.sun.com/products/jsp/`.
36. Java™ Servlet Specification, `http://java.sun.com/products/servlet/`.
37. Object Management Group, Object Constraint Language Specification, `http://www.omg.org/uml/`.