A Role-based Access Control Model for Protection Domain Derivation and Management

Trent Jaeger Frederique Giraud Nayeem Islam
Jochen Liedtke
IBM Thomas J. Watson Research Center
30 Saw Mill River Road,
Hawthorne, NY 10532

Abstract

We present a role-based access control (RBAC) model for deriving and managing protection domains of dynamically-obtained, remote programs, such as downloaded executable content. These are programs that are obtained from remote sources (e.g., via the web) and executed upon receipt. The protection domains of these programs must be limited to prevent content providers from gaining unauthorized access to the downloading principal's resources. However, it can be difficult to determine the proper, limited protection domain for a program in which downloading principals need to share some of their resources. Current systems usually rely on one of a number of possible principals to specify the content protection domains, but the exclusion of input from other principals limits the flexibility in which protection domains can be derived and managed. In this paper, we describe a RBAC model for deriving protection domains and managing their evolution throughout the execution of the content. This model accounts for the variety of principals that may be involved in domain derivation and how their input is managed. We demonstrate the use of this model to specify a variety of protection domain derivation and management policies.

1 Introduction

We present a role-based access control (RBAC) model for deriving and managing the protection domains dynamically-obtained, remote programs, such as downloaded executable content. Downloaded executable content are messages that contain programs that are executed upon receipt. It is well-known that downloaded executable content must be executed in a limited protection domain to prevent content providers from gaining unauthorized access to the downloading principal's resources [6] (e.g., private files). However, deriving a proper, limited protection domain and managing this domain over the execution of content can be difficult. For example, the semantics of most user files are not of interest to system administrators, so it is not feasible for them to determine whether accesses to these files should be permitted. Other principals, such as users and application developers, may know these answers, but they are not completely trusted to make access control decisions. Therefore, we develop an RBAC model that defines the access control decisions that principals can make, so that "least privilege" protection domains can be derived and managed.

Current systems demonstrate that multiple principals can provide information to derive content protection domains, but these systems lack a model of how these principals interact to make such decisions. Most systems rely primarily on one principal to make access control decisions, such as the content execution system developer [7, 16, 21] or user [13, 17, 26]. Other systems make no commitment to how protection domains are derived [9, 12, 19, 23]. A few systems use multiple means for deriving a protection domain. FlexxGuard enables users and/or system administrators to define protection domain limits for content, but users have the ultimate decision-making power [2]. In Netscape's Java Capabilities API [1], application developers request access rights within a limited protection domain specified by users or system administrators. In a Tcl flexible content interpreter [18], system administrators

RBAC 97 Fairfax Va USA

define how application developers can limit other content's access rights. These systems demonstrate the need for different principals to make different access control decisions, but the lack of a formal model describing what each can do hinders understanding of access control management.

RBAC models have been used extensively for access control policy management, but presently lack the flexibility to express how multiple principals can affect common access control decisions. RBAC models permit principals to assume a role, which is another principal with its own protection domain (e.g., [10, 22, 24, 27, 28]). RBAC models are often used by system administrators to specify mandatory access control (MAC) policies [20]. A MAC system partitions the world into two groups: (1) system administrators who specify access control policy and (2) users that are controlled by the policy. Thus, RBAC models enable system administrators to describe the rights available to principals, but do not describe how these principals may further delegate these rights. Therefore, users, content execution systems, and application developers must develop ad hoc mechanisms for limiting the rights that are to be delegated to the content that they execute.

In this paper, we define an RBAC model that: (1) associates protection domains with principals and their roles; (2) represents the subset of those domains that can be delegated to other principals and their roles; and (3) stores specifications that determine when delegations are permitted. First, in a manner typical of traditional RBAC models, protection domains can be specified for principals and their roles, such as users, users executing content execution systems, and users executing specific downloaded content. Next, the protection domain in which these principals can grant rights to other principals can be defined. These protection domains limit the rights that principals can grant to downloaded content. Finally, the conditions in which delegations are needed can be specified and also associated with content. Therefore, trusted principals (e.g., system administrators) can specify limits in which users, application developers, and content execution systems can make access control decisions. Also, all principals have a common model for expressing delegations, so application developers do not need to create ad hoc security models to enforce least privilege.

The structure of the paper is as follows. In Section 2, the access control problem is defined. Next, in Section 3, we define the RBAC model. In Section 4, we demonstrate how this model is used to derive protection domains using current policies and innovative policies. In Section 5, we conclude and discuss future work.

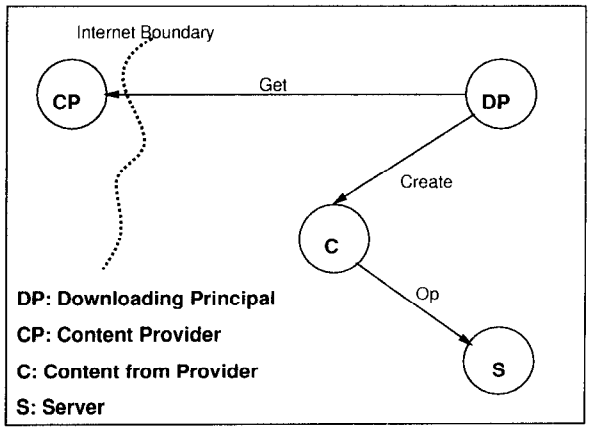

Figure 1: Simple downloaded content execution architecture

2 Problem

Remote programs can be used in applications that range from: (1) those implemented by a single, independent downloaded content to (2) complex applications in which content is used to convey individual operations on behalf of its providers. The security requirements of each of these application models vary in significant ways. For example, the protection domain for a single, independent executable can be the same for each execution and does not affect the protection domains of other content. However, in the case of a complex application, each content's protection domain depends on the way the application is being used. For example, collaborators should only have access to a downloading principal's object if that principal has made it available to the collaboration. Also, the actions in one content may affect the protection domains in other content. For example, the downloading principal may load an object into the collaborative application content which indicates that it is accessible to content from collaborators.

We first consider the security requirements of a single interaction using downloaded content as shown in Figure 1. A *downloading principal* is the principal who receives and executes the content. A system administrator (not shown) can specify access control policy for the downloading principal. A *content provider* is the principal that is responsible for the content. A *server* is a principal that stores and administers access to a

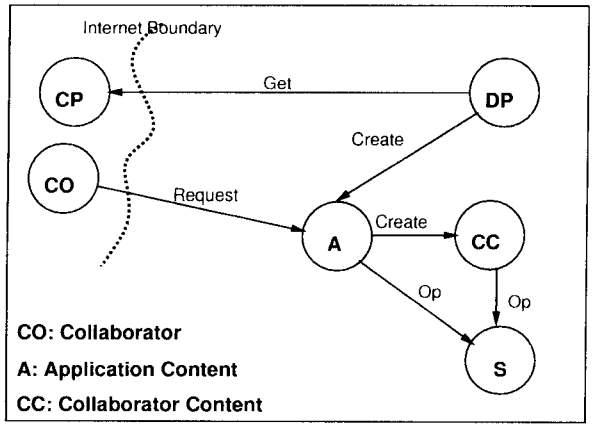

Figure 2: A more complex application implemented using downloaded content

set of objects (e.g., files, communication channels, application objects). A downloaded content execution system works basically as follows. First, the downloading principal *gets* (e.g., HTTP *get*) content from the content provider's server (often using an untrusted network, such as the Internet, as indicated by the dotted line). Next, the downloading principal *creates* a content process. The content process is then executed. It may request *operations* from various servers. In general, these servers may be local or remote (only a local server is shown).

More complex applications can also be implemented using downloaded content. Consider a collaborative application in Figure 2. Downloading principals retrieve the content for implementing their role in the collaboration (application content) as in the first example. They use this content to access their system objects as necessary to implement the collaboration. For example, a collaborative editor would need access to the files that the downloading principal may permit collaborators to edit. However, in this application, actions are implemented by downloading them to each collaborator. Therefore, collaborator content can interact asynchronously with the downloading principal's collaborative editor to modify documents. The collaborators must be limited to perform only the operations that the downloading principal has permitted them to perform. However, the rights the downloading principals permit may often depend on application semantics. For example, in a collaborative editor, only the files that the downloading principal has loaded to be collaboratively edited should be writable by other collaborators. Also, different content may use a different protection domain depending on its provider. For example, some collaborators may be permitted only to read the files, but not write them.

Servers store protection domains for each downloading principal to control the operations that they can perform on server objects. However, downloading principals typically do not trust content providers with all their rights. For example, downloading principals would like to prevent content from: (1) reading their private files; (2) writing executable files; (3) monopolizing their system's CPU; and (4) communicating with unauthorized principals. Unfortunately, traditional operating systems do not provide a mechanism for principals to dynamically restrict the access rights of one of their own processes. In addition, current RBAC models [4, 10, 24] or controlled execution systems [11] require that these limited domains be pre-specified, which can result in larger domains than desired for a single content execution. For example, a RBAC domain for an application would have to encompass all rights that any individual use of the application would ever need. Other systems explore the alternative where users specify protection domains themselves [5, 17], but most users cannot be trusted to make such decisions correctly.

The problems in deriving and managing protection domains are to:

- Enable multiple principals to make access control decisions within limits based on trust in their abilities
- Permit conditional access control decisions
- Map access control requirements to each downloading principal's system

The content protection domain may depend on input from a number of principals, including the downloading principal, content provider, system administration, and application developer, so they should be able to specify access control requirements. However, these principals are either not aware of the semantics of the applications or not completely trusted to make access control decisions, so the rights that they can delegate to content must be limited. Once an initial protection domain is established it may need to be modified based on the conditions that arise in executing the application. For example, new objects may be made available to an application that also need to be made available to content. Also, system administrators and application developers cannot be expected to know how a user organizes his personal files, so it must be possible to map access control requirements to actual system objects and operations.

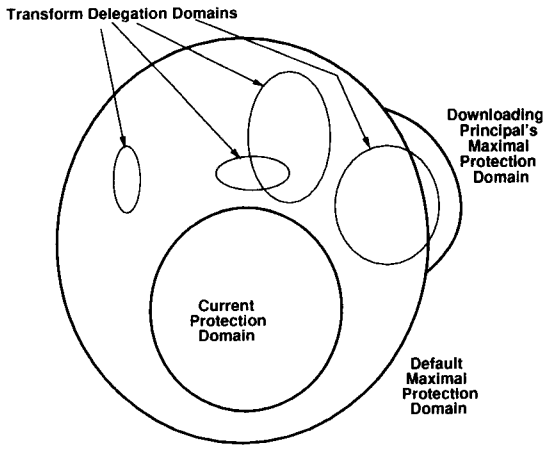

Figure 3: **Protection domain derivation**: Content executes with the *current protection domain*, but may be delegated rights within *transform delegation domains* as long as those rights are within the *maximal protection domain* which may be a combination of maximal protection domains delegated by multiple principals.

3 RBAC Model

We define an access control model based on RBAC principles for deriving content protection domains and managing their evolution throughout the content's execution. Our RBAC model is based on the following concepts:

- **Definition 1**: A *maximal protection domain* of a principal is a superset of all protection domains that the principal's processes can assume.
- **Definition 2**: A *protection domain combination function* that takes multiple maximal protection domains as input and combines them using set operations (e.g., union) to create a single maximal protection domain.
- **Definition 3**: A *delegation protection domain* of a principal defines the protection domain that the principal can delegate to other principals.
- **Definition 4**: A *current protection domain* of a principal's process is the protection domain upon which authorization of that process's operations is based.
- **Definition 5**: A *transform* changes the current protection domain of a process, but is limited by the maximal protection domain. Additionally, transforms may be limited by *delegation protection domains.*

The goal of this section is formalize these concepts using the RBAC model defined below. First, we intuitively describe the model's philosophy (as shown in Figure 3). Principals may specify *maximal protection domains* for content (i.e., the process that executes the content). In addition, a *protection domain combination function* can be defined to specify how a set of maximal protection domains can be combined to form a single maximal protection domain. For example, maximal protection domains for content may be defined for the default case (e.g., by the system administrator), by the downloading principals for access to their personal files, and for the application developer to control access based on the application's state. In one example policy, the union of the three maximal protection domains is the maximal protection domain for the content. However, the maximal protection domain that each principal delegates is limited by each's *delegation protection domain.* The *current protection domain* of the content is always a subset of a combination of the maximal protection domains. The current protection domain can be extended within the limits of the maximal protection domain using *transforms*, objects that associate operations with protection domain modifications. Further, *transform delegation protection domains* limit the rights that a transform can delegate.

The RBAC model associates the delegation protection domains, maximal protection domains, and transforms with roles. Roles are defined in terms of content attributes and the principals involved in the interaction. For example, the default maximal protections can be retrieved based on the content provider, content rating service, content rating, content type, and content application. Other protection domains are associated with roles in a similar manner.

Protection domains are associated with roles using a general representation called a *policy graph* defined below:

- **Definition 6**: A *policy graph* is a tuple, $\{p, G, tm, cm\}$, where: (1) p is a principal; (2) G is a directed graph, $G = \{N, V\}$, where N is a set of nodes and V is a set of vertices; (3) tm, the *traversal method*, describes how the graph is traversed; and (4) cm, the *combination method*, determines how a node's value is combined with the current result.

A policy graph associates downloading principals with a directed graph that represents policy for

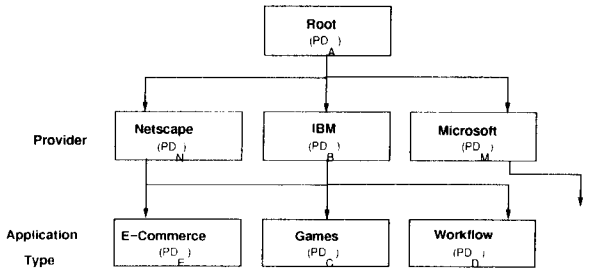

Figure 4: **Policy graph**

handling content. An example policy graph for specifying content protection domains for the protection domain derivation server is shown in Figure 4. In this policy graph, each graph level corresponds to a content attribute (e.g., content provider and application type). Nodes store a mapping between content attribute values and the associated protection domain. Given content attribute values of `provider=IBM` and `application type=games`, the protection domain PD_C is found. The methods describe how the graph is traversed and how the policy values of the traversed nodes are combined. An example traversal method specifies that traversal starts at the root of the graph and repeatedly follows the child link whose node's attribute value includes the content's value for that attribute. The example combination method is the union set operator. A traversal that goes from the root to the IBM node to the games node results in the protection domain $PD_A \cup PD_B \cup PD_C$.

Note that Netscape content shares the same protection domains at the application type level as the IBM content in our example. Therefore, the resulting domain for a content identity of `provider=Netscape` and `application type=games` is $PD_A \cup PD_N \cup PD_C$.

Below, we formalize the definition of protection domains and transforms that we will use in our access control model [15].

- **Definition 7**: An *object group* is a set of objects of the same type (may be a supertype of all objects).
- **Definition 8**: *Protection domains* of a principal are defined by a set of domain rights and exceptions defined below:
  - **Domain rights**: A tuple, $\{type, allowed\_ops, obj\_grp, limit\}$, which describes a set of operations, *allowed_ops*, that the principal can perform on an *obj_grp* of a specific *type*. The *limit* specifies restriction on number of times such an operation can be performed.
  - **Exceptions**: A tuple, $\{type, precluded\_ops, obj\_grp\}$, which describes a set of operations, *precluded_ops*, that the principal is precluded from performing on an *obj_grp* of a specific *type*.
- **Definition 9**: A *transform* is one of three types of tuples, object group transforms, operation transforms, or domain transforms:
  - **Object group transforms**: A tuple, $\{op\_grp, t\_op, t\_type, obj, obj\_grp\}$, that, when this principal executes an operation in *op_grp*, describes a change in the relationship between an object *obj* and an object group *obj_grp* (*t_op* is *add* or *remove*) for all principals. *t_type* indicates whether the type of transform: (1) applied before operation; (2) applied before and rescinded after operation; and (3) applied after the operation.
  - **Operation transforms**: A tuple, $\{op\_grp, rcvr, t\_type, type, ops, obj\_grp, limit\}$, that, when this principal executes an operation in *op_grp*, describes a new domain right (if *ops* are allowed) or exception (if *ops* are precluded) to be added to the principal *rcvr*'s current protection domain. *t_type* is as described above.
  - **Domain transforms**: A tuple, $\{op\_grp, t\_op, principal\}$, that, when this principal executes an operation in *op_grp*, describes a change in the principal's protection domain specified by *t_op* (one of *intersect*, *union*, or *replace*) relative to another principal's protection domain. These transforms are always applied before the operation.

In this model, objects can be aggregated into groups called *object groups* for expression of a common access control requirement. Object groups may refer to a logical group, where mapping functions may be used to identify the system objects that belong to the group. The *protection domain* of a principal to these object groups are described by its domain rights and exceptions. A *domain right* describes the permission to perform a set of operations on an object group, and an *exception* precludes a set of operations from being performed on an object group. This model permits

access rights to be defined concisely for a large set of objects while permitting some objects in the group to override those rights. Principals are authorized to perform an operation on an object if their protection domain contains: (1) at least one domain right that permits the operation and whose limit has not been exceeded and (2) no exceptions that preclude the operation.

Also, developers can specify *transforms* that associate protection domain modifications with operations. These enable a principal's action to delegate rights to another principal, revoke rights from another principal, or revoke rights from itself. Transforms are motivated by Foley and Jacob's association of rights with collaborative activities [8]. A transform can be executed only if the receiving principal is allowed to obtain the rights granted. In addition, transforms may be restricted, so only principals with the rights originally can delegate them. This prevents self-delegation. *Object group transforms* add or remove an object from an object group. This results in the other principals rights being modified based on their access to the object group. *Operation transforms* can add domain rights and exceptions to the receiving principal's protection domain. *Domain transforms* combine the domains of two principals when they interact. For example, the protection domain of untrusted principals can be intersected before they interact.

4 RBAC Model Usage

In this section, we demonstrate the flexibility and power of our RBAC model by using it to define several policies for controlling downloaded executable content. We start by specifying the simple policies from Java-enabled Netscape and the Java Appletviewer. We then demonstrate how the policies enforced by some recent downloaded content execution systems can be expressed using our model.

The flexibility of our model comes from its ability to express: (1) how rights distribution is controlled and (2) limited modifications to protection domains associated with operations. First, the model enables system administrators, content execution systems, application developers, and users to specify the rights that other principals may grant to content. Second, as we will demonstrate in the last two examples, transforms enable the principals to delegate rights to content within their limited domain of control. This enables application developers and users to enforce least privilege based on their expected use of the content.

4.1 Java-enabled Netscape 3.0

Java-enabled Netscape 3.0 [21] has a very simple security policy that demonstrates some basic protection domain specification features of our model. Roles are not needed in this example (or the following one). Content may only communicate with processes at their source IP addresses. The content execution system enforces this policy and there is no flexibility for other principals to modify it.

Using our model, the maximal protection domain for any content is the maximal protection domain specified by the content execution system (i.e., Java-enabled Netscape). For all content, the maximal protection domain is:

- $\{channel, [send|receive], source\_ip\}$

where *channel* refers to a communication channel, *send* and *receive* are the permitted operations, and *source_ip* is a logical object that refers to the content's source IP address (ports greater than 1024). A mapping function converts the *source_ip* object group to the specific group of objects (e.g., 129.9.2.14:>1024) used for authorization.

The maximal protection domain specified by the content execution system becomes the current protection domain for all content. No transforms are specified, so this protection domain is immutable.

4.2 JDK 1.1 Appletviewer

The JDK 1.1. Appletviewer system [13] has only a slightly more complicated policy than Java-enabled Netscape. The appletviewer provides a limited protection domain in which users can grant file access rights to content (in addition to the right to communicate with the source IP machine). The rights the user delegates to content are delegated to all content downloaded.

In this case, the maximal protection domain of content is the union of the maximal protection domains specified by the appletviewer and the downloading principal. The appletviewer specifies its maximal protection domain for content (communicate with the source IP address as described in the previous section) and the delegation protection domain for the downloading principals. This appletviewer policy permits users to delegate read and write privileges to any of their files and read access to a limited set of environment variables. However, execute privileges are not made available for downloading principals to grant. Using our model, the delegation protection domain for downloading principals is (specified for files only):

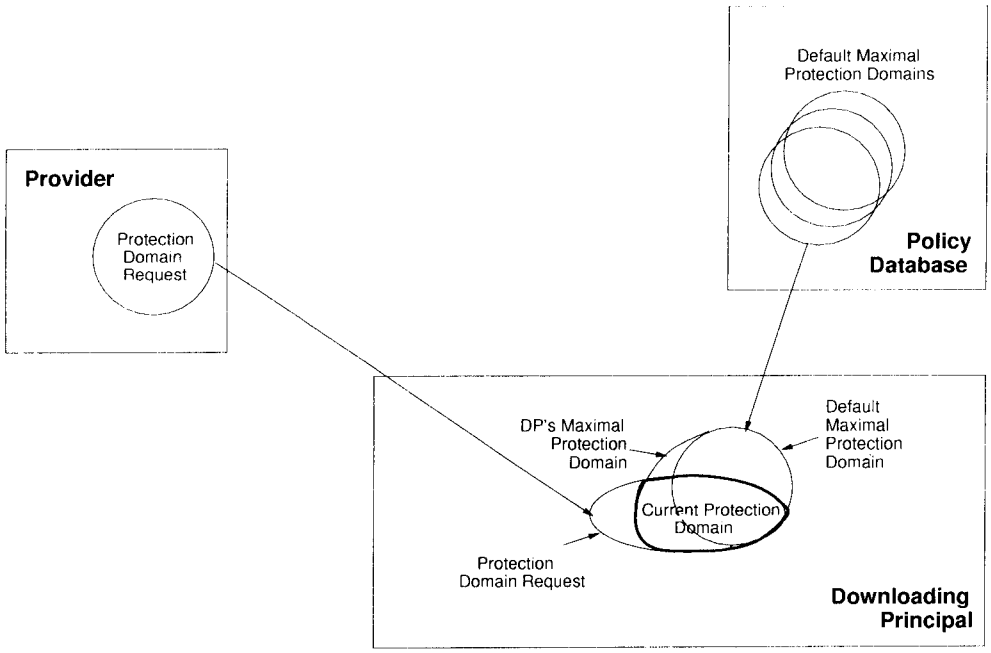

Figure 5: **Domain derivation protocol for FlexxGuard:** (1) derive the *default maximal protection domain* for the content based on its authenticated description; (2) intersect the content provider's *protection domain request* with the *default maximal protection domain*; and (3) if necessary, the downloading principal can grant additional rights (the *DP's maximal protection domain*).

- $\{files, [read|write], user\_files\}$
- $\{files, [read], all\_accessible\_files\}$

This specifies that a downloading principal may grant read and write privileges for their files (e.g., in their home directory subtree), but only read access to all other files that they can access.

Within these limits, downloading principals specify a maximal protection domain for all content. For the appletviewer, downloading principals specify the maximal protection domain in the `.properties` file. The union of the two maximal protection domains becomes the current protection domain for any content that is downloaded and run by the appletviewer. Like Java-enabled Netscape, this protection domain is unchanged throughout the content's execution.

4.3 JDK 1.1 FlexxGuard

JDK 1.1 FlexxGuard is an extension of the JDK 1.1 Appletviewer that enables users to assign more liberal protection domains to authenticated content from trusted sources while still tightly restricting untrusted content [3, 2]. In FlexxGuard, downloading principals or system administrators specify maximal protection domains for content based on their descriptions (i.e., roles). Downloaded content can include a request for a protection domain from the downloading principal (also authenticated). If this request is a subset of the default maximal protection domain, then the requested protection domain is used to authorized content operations (i.e., becomes the current protection domain). If not, then the downloading principal can selectively override and/or modify their default maximal protection domain for content.

Using our model, the content execution system defines the delegation protection domain of users to their normal protection domain. Downloading principals or system administrators express default maximal protection domains for content based on the content's authenticated description. This maximal protection domain unioned with the maximal protection domain granted by the downloading principal at runtime. The current protection domain is the intersection of the requested protection domain provided with the content and the maximal protection domain. The current protection domain is unchanged throughout the content's execution. Therefore, unlike the first two systems, a content's current protection domain in FlexxGuard can be a subset of the content's maximal protection

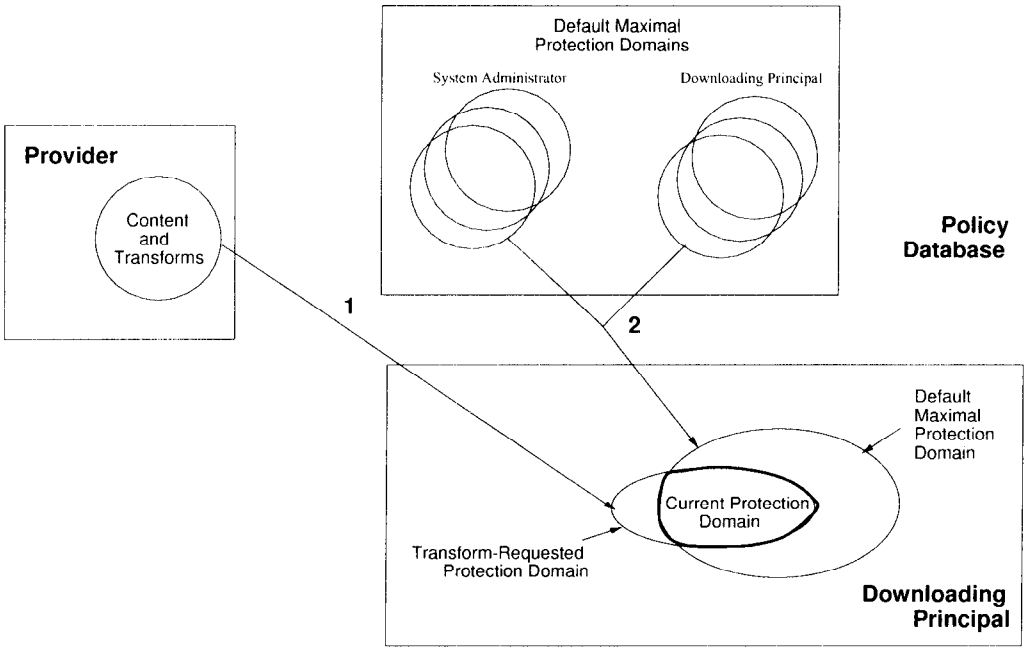

Figure 6: **Domain derivation protocol for Netscape with the Java Capabilities API**: (1) identify principal or role from content certificate's signee; (2) retrieve and union the system administrators and downloading principal's enabling policy to create the *maximal protection domain*; (3) set the initial *current protection domain* of the content to null; and (4) content uses transforms to enable rights within the *maximal protection domain*.

domain.

The key feature of FlexxGuard is its ability to manage default maximal protection domains for downloading principals. A policy graph (see Definition 6) stores associations between content attributes and protection domains that are used to derive the default maximal protection domains. For example, the attributes, manufacturer name, content type, and content name, are used in the policy graph shown in Figure 4. The policy graph is traversed from the root to the descendant nodes whose attributes values match or contain the value of that attribute for the content. For example, content with attributes `manufacturer name=IBM` and `content type=games` would traverse the path shown in Figure 4. FlexxGuard uses the union set operation as its combination function, so the resulting protection is $PD_B \cup PD_C$. An alternative is to use the most recent protection domain which would yield PD_C in this example.

Each policy graph is associated with a principal (e.g., user, role, or group). Therefore, the maximal protection domain is derived based on the identity of the principal. The resulting maximal protection domain is analogous to the role *principal as downloading principal executing an IBM game*.

4.4 Netscape's Java Capabilities API

Netscape's Java Capabilities API also enables downloading principals and systems administrators to specify maximal protection domains for content [1]. Netscape permits downloading principals and system administrators to grant or revoke permissions for content. Like FlexxGuard, principals are not limited in the permissions that they may grant. However, unlike FlexxGuard, a principal may permit or forbid a permission. Forbidding a permission means that content may never have this permission in its maximal protection domain. This model requires that a principal must explicitly revoke any rights that content must not have, so the model is not fail-safe. Content's maximal protection domain is the union of these granted protection domains less any revoked rights.

Permissions within the maximal protection domain are activated by the *enablePrivilege* command. Content can rescind any enabled permissions and can prevent content it calls from enabling a permission (using the commands *revertPrivilege*) and *disablePrivilege*, respectively). The authorization mechanism scans the call stack to determine if any method on the stack has enabled the permission. Therefore, developers must explicitly rescind any permissions that they have en-

abled to prevent called methods with less trust from using those rights. We show how this can be implemented using transforms, but demonstrate a preferable method in the next section in which a single specification rescinds all rights not authorized to the called content (i.e., not within its current protection domain). Note that Wallach *et al.* describe an authorization mechanism for stack introspection that additionally requires that each principal on the stack must have the right within its maximal protection domain before it is authorized [25]. We prefer using the current protection domain, although they may be the same in some instances.

Using our model, protection domains are derived in the following manner using the policy of the Netscape Java Capabilities API (see Figure 6). System administrators and downloading principals associate maximal protection domains with roles. Their delegation protection domains are the entire system. Domain rights indicate the rights granted, and exceptions indicate the rights that are forbidden. The identity of the content certificate's signee is used to determine the content's role. The maximal protection domains are unioned: domain rights domains for which no exception is specified are added into the maximal protection domain. This is precisely the semantics of union in our access control model.

Once the maximal protection domain has been defined, the current protection domain is managed within it. First, the content execution system sets the current protection domain to null. The content itself specifies transforms (see Definitions 5 and 9) when rights are needed for an operation. In the Java Capability API, rights are lexically scoped, so once the "enabling" method exits, the rights are removed from the current protection domain. The granting of the right to read a game score file by a transform can be specified in the following manner using our model:

- $\{initialize, self, bl, files, [read], score\_files\}$

The *initialize* method delegates the right to the content that called it (*self*) prior to the method executing. In our model, revocation of the right must be specified as well. The transform type argument *bl* indicates that the right is to be activated before the operation and removed after the operation. Otherwise, a transform such as the one below would need to be specified for the operation.

- $\{initialize, self, a, files, [-read], score\_files\}$

Netscape's Java Capabilities API only has a command for making lexically scoped protection domain changes (*enablePrivilege*). However, we believe that both lexically-scoped delegations and global delegations are necessary. We also demonstrate this in the next section.

4.5 Protection Domain Derivation Server

In the last example, we demonstrate a Protection Domain Derivation Server for a content execution system that manages complex applications implemented by multiple instances of downloaded executable content [15]. Content is divided into two types: (1) application content and (2) collaborator content. Application content is used directly by downloading principals (e.g., through an interface) to implement their actions. Collaborator content implements other users' actions in the application. For example, a collaborative editor is an application, and collaborators use content to implement their editing operations on documents.

This example's policy is for system administrators to define default maximal protection domains for content and downloading principals. In addition, they define delegation protection domains for other principals, such as downloading principals and application developers. The maximal protection domain for content is the union of the default maximal protection domain and the protection domains delegated by the downloading principal and application developer. This policy enables system administrators to limit content's access to system objects while giving downloading principals and application developers some control over their objects. However, system administrators will need to provide safeguards to prevent spoofing attacks against users. For example, delegation of an execute privilege for an object that can be written by content should be prevented in most cases.

In this example, the initial current protection domains are derived using the protocol displayed in Figure 7. The protocol consists of three stages: (1) mapping the content provider's protection domain request to the downloading principal's system; (2) verifying that the request is within the default maximal protection domain for the content and downloading principal; and (3) resolving any conflicts using the downloading principal's delegation domain.

First, content providers specify a *protection domain request* for their content. The content provider should not be expected to know the structure of the downloading principal's system, so *mapping functions* are defined to convert logical objects in the request to physical objects on the downloading principal's sys-

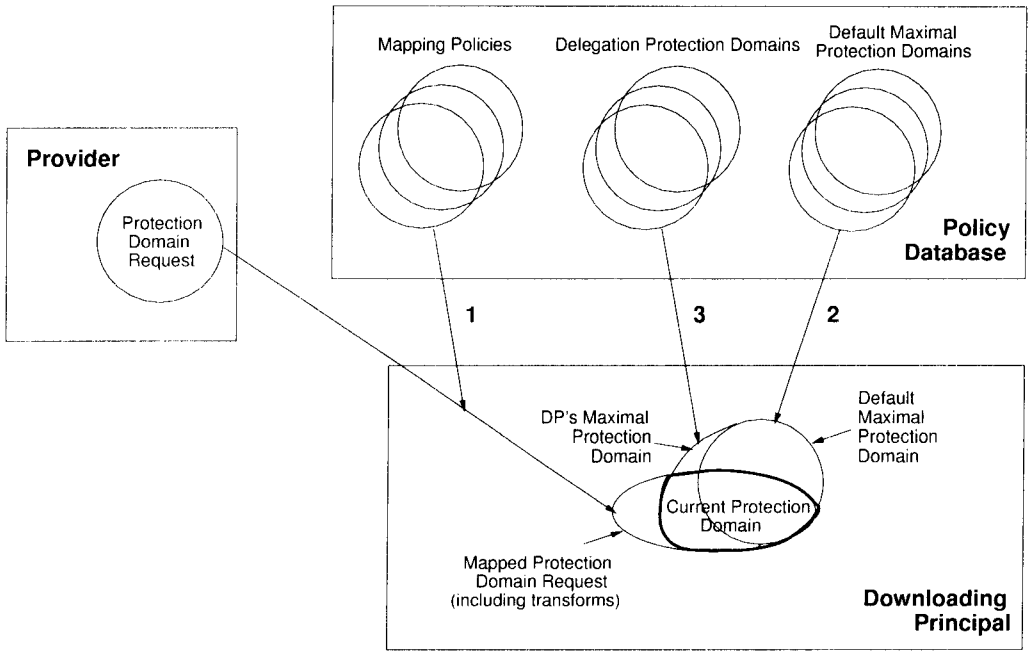

Figure 7: **Domain derivation protocol**: (1) map the content provider's *protection domain request* to a *mapped protection domain request*; (2) retrieve *default maximal protection domain* from the policy graph and intersect with the request; and (3) users can grant additional rights from within their *delegation protection domain.*

tem. For example, application files can be located anywhere in the downloading principal's file system. A mapping function derives their location when the application's content is downloaded. Mapping functions are stored in a policy graph that associates the downloading principal and content description (i.e., role) with the mapping functions by object type and/or logical object name. The resulting domain is called the *mapped protection domain request.*

Next, the mapped protection domain request is compared to the default maximal protection domain to verify that the domain request is acceptable. The protection domain derivation server derives the default maximal protection domain by comparing the content's role (downloading principal, description, and provider) to the downloading principal's policy graph (as in the FlexxGuard example). If the mapped protection domain request is a subset of the content's maximal protection domain then it is verified as acceptable. An acceptable, mapped protection domain request is the initial current protection domain. Since the current content domain is a subset of the maximal content domain, access control model transforms can be used to add new rights as the actions of the downloading principal make them available.

If the mapped protection domain request is not acceptable, then the downloading principal and/or the application developers may be permitted to grant the additional rights requested (i.e., expand the maximal protection domain). The delegation protection domains for the downloading principal are retrieved using the content's role once again. Downloading principals may grant any rights to content that are within their delegation protection domain.

Transforms are used to modify the current protection domain. Transforms are provided with the content's protection domain request, but they are authorized when they are used. We examine a transform that enables the downloading principal to use the application to delegate rights to collaborator content. An example transform is shown in Figure 8[1]. This example specifies that collaborators can access a recording file and its associated annotation file when the `start_replay` operation is executed. Four transforms are specified: (1) an operation transform that delegates read and write access to the recording object to principals assigned to the `chief_scientists` role; (2) an operation transform that delegates read access to the recording object to principals assigned to the `scientists`'s role; and (3) a pair of object group transforms that place the recording and annotation

[1] x indicates a return value of an operation.

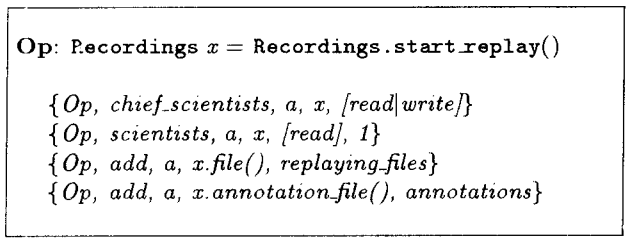

Figure 8: **Transform specification:** Upon `start_replay` operation, delegate access to a recording being replayed (x), its file ($x.file()$), and its associated annotation file ($x.annotation\_file()$).

files into the application object groups. Each operation transform grants rights for the principals to access the recording x returned by `start_replay`. The transforms for `x.file()` and `x.annotations_file()` place these objects in the *replaying_files* and *annotations* object groups, respectively. These transforms enable content providers to perform operations on these object groups as specified by their protection domains. For example, a principal in the *chief scientist* role can both read and write the annotations and the recording.

In addition, we address the problem of automatically revoking transformed rights upon calling less trusted content. A domain transform can be associated with content that intersects protection domains whenever content with a different protection domain is called.

- $\{downloaded_{c}ontent, intersect, called_{p}rincipal\}$

This intersects the current protection domain of the executing content process with the current protection domain of the called principal for any downloaded content method. Calls to system method can be handled as appropriate to those methods. Also, other transforms may be applied to grant some of the restricted rights back (if the called principal is sufficiently trusted). This type of transform does not impose a significant cost if protection domain crossings are well-identified (as would be the case using Tcl 7.5's *alias* operation).

The use of this transform is too costly for current Java virtual machines. Recent proposals for controlled Java interpreters [1, 12] examine the stack to derive the protection domain when a controlled operation is called. However, any interactions that are not on the stack when the controlled operation is called will be not be accounted for in the derivation.

5 Conclusions and Future Work

In this paper, we present a RBAC model for deriving and managing protection domains for remote programs, such as downloaded executable content. In this model, protection domains consist of three parts: (1) a maximal protection domain; (2) a current protection domain; and (3) a set of transforms that modify the current protection domain within the limits of the maximal protection domain. The RBAC model enables derivation of the maximal and current protection domains by associating roles with: (1) delegation protection domains that define the rights that a principal can delegate; (2) maximal protection domains that define the rights that are being delegated to a role; and (3) mapping functions that convert logical object specifications to system objects. The RBAC model enables maximal protection domains to be defined from a set of individual maximal protection domains, so multiple principals can provide input to rights derivation and management. Current protection domains are the intersection of the initial protection domain requested by content providers and the maximal protection domain. Transforms enables protection domains to be modified as content operations are executed.

We demonstrate this model by using it to define the security policies of five different content execution systems. Java-enabled Netscape uses the model's ability to specify a current protection domain. Java appletviewer uses the model's ability to specify a limited delegation protection domain for downloading principals. Java FlexxGuard uses the model's role-based representation to derive maximal and current protection domains. Netscape's Java Capabilities API uses the model's transforms to manage the content's current protection domain. Our Protection Domain Derivation Server uses the model's ability to define limited delegation domains, derive maximal delegation domains, map logical objects, derive current protection domains, and manage current protection domains.

In the future, system constraints need to be integrated into the protection domain management in a more explicit way. For example, constraints such as, users may not grant both execute and write privilege to a file need to be enforced. We have defined a policy for limiting the what rights can be delegated to downloaded executable content [14], but these have not yet been integrated into our RBAC model.

References

[1] Introduction to the capabilities classes, June 1997. Available at http://developer.netscape.com/library/ documentation/ signedobj/capabilities/ 01cap.htm.

[2] R. Anand, N. Islam, T. Jaeger, and J. R. Rao. A flexible security model for using Internet content. *IEEE Software*, 1997. To appear.

[3] R. Anand, N. Islam, and J. R. Rao. A capability-based security model for using Internet content. Technical Report 20664, IBM Research, 1996.

[4] L. Badger, D. F. Sterne, D. L. Sherman, K. M. Walker, and S. A. Haghighat. Practical domain and type enforcement for UNIX. In *IEEE Symposium on Security and Privacy*, pages 66–77, 1995.

[5] A. Berman, V. Bourassa, and E. Selberg. TRON: Process-specific file protection for the UNIX operating system. In *Proceedings of the 1995 USENIX Winter Technical Conference*, pages 165–175, 1995.

[6] N. S. Borenstein. Computational mail as a network infrastructure for computer-supported cooperative work. In *Proceedings of the Fourth ACM Conference on Computer-Supported Collaborativr Work*, pages 67–74, 1992.

[7] N. S. Borenstein. Email with a mind of its own: The Safe-Tcl language for enabled mail. In *ULPAA '94*, pages 389–402, 1994.

[8] S. Foley and J. Jacob. Specifying security for CSCW systems. In *Proceedings of the 8th IEEE Computer Security Foundations Workshop*, pages 136–145, 1995.

[9] F. S. Gallo. Penguin: Java done right. *The Perl Journal*, 1(2):10–12, 1996.

[10] M. Gasser and E. McDermott. An architecture for practical delegation in a distributed system. In *IEEE Symposium on Security and Privacy*, pages 20–30, 1990.

[11] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer. A secure environment for untrusted helper applications. In *Proceedings of the 6th USENIX Security Symposium*, pages 1–14, July 1996.

[12] L. Gong. New security architectural directions for Java. In *IEEE COMPCON '97*, February 1997.

[13] J. Gosling and H. McGilton. The Java language environment: A white paper, May 1996. Available at URL http://www.javasoft.com/docs/language_ environment/.

[14] T. Jaeger. *Flexible Control of Downloaded Executable Content.* PhD thesis, University of Michigan, February 1997.

[15] T. Jaeger, N. Islam, R. Anand, A. Prakash, and J. Liedtke. Flexible control of downloaded executable content. Technical report, IBM Research, 1997. Submitted for journal publication.

[16] T. Jaeger and A. Prakash. Support for the file system security requirements of computational e-mail systems. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, pages 1–9, 1994.

[17] T. Jaeger and A. Prakash. Implementation of a discretionary access control model for script-based systems. In *Proceedings of the 8th IEEE Computer Security Foundations Workshop*, pages 70–84, 1995.

[18] T. Jaeger, A. Rubin, and A. Prakash. Building systems that flexibly control downloaded executable content. In *Proceedings of the 6th USENIX Security Symposium*, pages 131–148, July 1996.

[19] J. Levy and J. Ousterhout. Safe Tcl: A toolbox for constructing electronic meeting places. In *The First USENIX Workshop on Electronic Commerce*, pages 133–135, 1995.

[20] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control: a multi-dimensional view. In *Proceedings of the Tenth Computer Security Applications Conference*, pages 54–62, 1994.

[21] S. Thomas. The Navigator Java environment: current security issues, January 1996. Available at http://developer.netscape.com/library/ documentation/ javasecurity.html.

[22] T. C. Ting, S. A. Demurjian, and M.-Y. Hu. Requirements, capabilities and functionalities of user-role based security for an object-oriented design model. In *IFIP Transactions, Database Security V*, pages 275–296, 1992.

[23] G. van Rossum. Grail – The browser for the rest of us (draft), 1996. Available at http://monty.cnri.reston.va.us/grail/.

[24] S. T. Vinter. Extended discretionary access controls. In *IEEE Symposium on Security and Privacy*, pages 39–49, 1988.

[25] D. Wallach, D. Balfanz, D. Dean, and E. W. Felten. Extensible security architectures for java. Technical Report 546-97, Princeton University, 1997.

[26] J. E. White. *Telescript Language Reference Manual.* October 1995. Available at http://www.genmagic.com/Telescript/TDE/TDEDOCS_HTML/telescript.html.

[27] E. Wobber, M. Abadi, M. Burrows, and B. Lampson. Authentication in the Taos operating system. *ACM Transactions on Computer Systems*, 12(1):3–32, February 1994.

[28] M. E. Zurko and R. Simon. User-centered security. In *Proceedings of the 1996 New Security Paradigms Workshop*, 1996.