Implementation of a Discretionary Access Control Model for Script-based Systems

Trent Jaeger and Atul Prakash
Software Systems Research Laboratory
Department of Electrical Engineering and Computer Science
University of Michigan, Ann Arbor, MI 48109-2122
E-mails: {jaegert|aprakash}@eecs.umich.edu

Abstract

Powerful applications can be implemented using command scripts. A command script is a program written by one user, called a writer, and made available to another user, called the reader, who executes the script. For instance, command scripts could be used by Mosaic, the popular World-wide Web browsing tool, to provide fancy interfaces to services, such as banking, shopping, etc. However, the use of command scripts presents a serious security problem. A command script is run with the reader's access rights, so a writer can use a command script to gain unauthorized access to the reader's data and applications. Existing solutions to the problem either severely restrict I/O capability of scripts, limiting the range of applications that can be supported, or permit all I/O to scripts, potentially compromising the security of the reader's data. We define a discretionary access control model that permits users to flexibly limit the access rights of the processes that execute a command script. We use this model in a prototype system that safely executes command scripts available from Mosaic.

Keywords: Discretionary access control, script-based systems, authorization, authentication, operating systems, file systems.

1 Introduction

Powerful applications can be implemented using command scripts. A command script is a program written by one user, called a writer, to act on the writer's behalf when another user, called the reader, executes the script. Examples of systems that utilize command scripts include: (1) Mosaic, the popular information server for the World-wide Web; (2) Telescript[1] [17], a system meant for building electronic marketplaces; and (3) active or enabled mail systems [2, 3, 6]. For example, Mosaic uses command scripts to define server actions when a client wants to access information from the server.

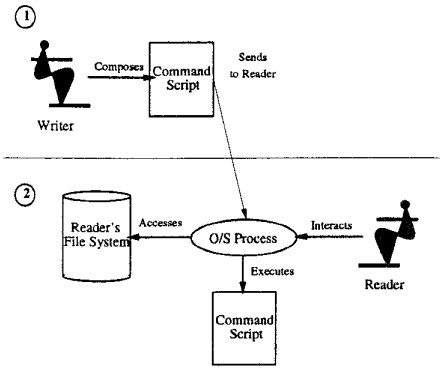

Figure 1: Command script execution

Unfortunately, the use of command scripts also presents a major security risk. Figure 1 demonstrates how a command script is composed and executed. First, a writer composes the command script. Through some mechanism (e.g., Mosaic or enabled mail) the command script is transferred to the reader. When the reader reads the command script (number 2 in the Figure), a process is created to execute the command script. This process runs on the reader's machine and is owned by the reader, so the command script is executed with the reader's access rights. A malicious writer can use these additional access rights to: (1) read and write the reader's private objects; (2) execute applications, such as `mail`, to masquerade as the reader to other users; and (3) read the password file, `/etc/passwd`, on the reader's machine.

[1] Telescript is a registered trademark of General Magic, Inc.

File system security in the above systems is provided typically either by severely limiting the ways that I/O can be performed within a command script or by trusting that users will not write improper command scripts. This is exemplified by the Safe-Tcl [3] language for enabled mail command scripts (an extension of the Tcl language [13]). Safe-Tcl provides two interpreters: (1) a trusted interpreter and (2) an untrusted interpreter. The trusted interpreter provides no security, so it is meant to be used for interaction with trusted sources. The untrusted interpreter provides tight security by replacing all the I/O functions with safer I/O functions that only permit I/O to a single, public directory. This provides security for the file system, but it limits the ways in which command scripts can be used. We find that both solutions are somewhat unsatisfactory. Severely limiting the I/O capability of command scripts makes it difficult to build applications that perform I/O. On the other hand, permitting all I/O can cause the reader's objects to be accidentally or maliciously accessed or modified by the script. Borenstein has also recognized these two options do not meet the needs of some applications, so he has left open the possibility of "power-augmenting extensions" to the untrusted interpreter.

Our goal is to define an access control model that permits readers to execute command scripts with a level of security between the two extremes. Our initial effort, the intersection security model [7], enables readers to permit I/O to command scripts from trusted writers while protecting the private objects of both the reader and the writer. In the intersection model, access is limited to only the objects that the reader and writer share.

There are several limitations to the intersection model, however: (1) the writer may not be a known principal on the reader's machine, so the intersection may be null; (2) the reader may want to further limit the access rights of a command script to perhaps prevent the execution of unsafe applications or to prevent access to system objects, such as `/etc/passwd`; and (3) the reader may want to permit access to some private objects to complete the interaction. The access rights needed for executing a command script can vary greatly depending on the goal of the interaction, the application, and the writer. In this paper we define a discretionary access control model that permits the reader to flexibly define the access rights for executing a command script. In addition, the writer can also grant access rights to the command script using our model. We demonstrate the use of this access control model in the implementation of a system that executes Mosaic server scripts.

The structure of the paper is as follows. In Section 2, we define the problem and list our assumptions. In Section 3, we describe the implementation of an application server on Mosaic and the security problems encountered. In Section 4, we outline the security requirements of the application server. In Section 5, we define our discretionary access control model. In Section 6, we detail a prototype implementation of a script execution system for the Mosaic application server that uses our access control model. In Section 7, we present conclusions and outline directions for future work.

2 Problem Statement

In conventional systems, *principals* (e.g., users, machines, etc.) execute processes that perform *operations* (e.g., read, write, etc.) on *objects* (e.g., files, printers, etc.). The permissions of a principal to perform operations on system objects are called the *access rights* of the principal in the system. In the execution of a command script, the reader is the principal who executes a process whose code (i.e., the command script) is written by another principal, the writer. Since the reader is the process' principal, the process can perform any operation on any object that the reader can. This level of access rights for a command script is unacceptable because the writer can gain access rights to the reader's private objects. However, in some cases, the reader may need to grant limited access rights to enable the command script to perform the I/O (i.e., the operations on system objects) necessary to implement the interaction. Meanwhile, objects not involved in the interaction should be protected from access. The reader must be able to flexibly limit a command script's access rights, such that the reader can be confident that the objects that need to be protected are.

In addition, the writer may also wish to grant access to objects for use by the reader. The writer should also be able to limit the set of objects made available to the reader based on the purpose of the command script.

The assumptions we make in solving the problem stated above are as follows. First, we assume a multi-user system that contains objects that are organized in an hierarchical, name space. We use Unix [2] syntax for the name space in our examples because of its familiarity. Also, the granularity of access control is limited to the object-level. Therefore, if a book is an

[2] Unix is a registered trademark of the Unix Open Foundation, Inc.

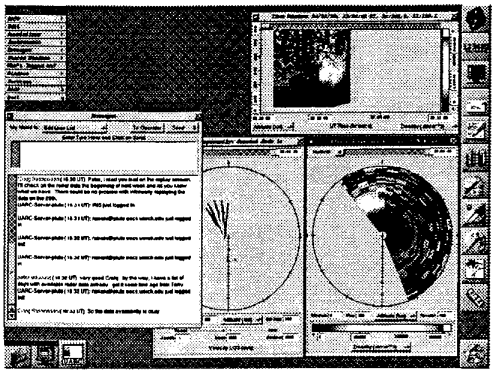

Figure 2: The UARC System Interface

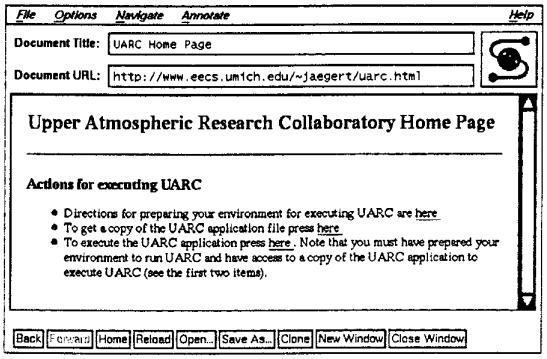

Figure 3: UARC Home Page

object, access control on the chapters in the book is only possible if the chapters are also objects. Lastly, we assume the presence of a secure operating system, such as Trusted Mach [1], that provides authentication of principals, secure communication channels, a secure initialization procedure, etc.

3 Example Problem

An example where the reader must permit the command script to perform some limited I/O is the World-wide Web (WWW) server for the Upper Atmospheric Research Collaboratory (UARC) [4] system. The UARC system provides several services for remote, collaborative analysis of atmospheric test data. For example, users can read analysis data and write annotations to that analysis data. Also, users can write recordings of analysis sessions to replay later or provide to other users [10]. The interface of the UARC system is shown in Figure 2.

We enable clients to access our UARC-WWW server by defining a home page for the server on Mosaic, which we call the UARC Home Page (see Figure 3). The UARC Home Page entries make the following capabilities available:

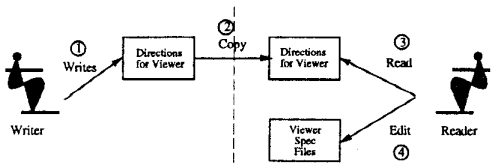

Figure 4: Get directions for viewer

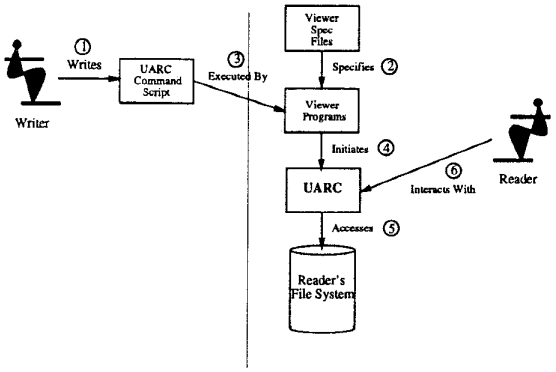

Figure 5: Execute UARC command script

1. Get directions for specifying a viewer for executing the UARC command script
2. Obtain a copy of the UARC application
3. Execute the UARC command script

In order to execute the UARC command script from Mosaic, each reader (i.e., client) must specify a viewer for executing the script. In the first entry, we provide directions for the reader to edit the necessary specification files (e.g., `.mailcap`, `.mime.types`, etc.) to add this viewer (see Figure 4). The details of this specification are provided in the Implementation Section. In addition, a reader may not have access to a copy of the UARC application. A second entry in the UARC Home Page uses `ftp` to copy a UARC executable to the reader's machine. The UARC executable must be protected from theft and forgery by an intruder, so the server must encrypt the UARC executable using both the writer's private key (for authentication) and then the reader's public key (for secrecy). Selection of the third entry initiates the execution of the UARC command script (see Figure 5). We are only concerned about forgery of the command script, so only authentication of the writer is necessary. The UARC command script is processed using

the viewer specified by the directions. The viewer collects access control restrictions from the reader and executes the UARC application while enforcing those restrictions.

Execution of the UARC command script could result in the following security problems. The process that executes the UARC command script is run on the reader's machine and is owned by the reader. Therefore, the command script is run with the reader's access rights. We believe that the readers have a reasonable amount of trust in the developers of the UARC system, but they and their system administrators are hesitant to execute a command script under such conditions. Readers are concerned about the accidental modification of their private data. System administrators are worried about the script's access to local system data and their inability to trace a breach of security if one occurs.

It is worth noting at this stage that executing the command script with the access rights of the writer is not an effective solution to this problem. In many cases, the reader interacts directly with the command script or an application triggered by the command script. If the application or command script provides I/O functionality, the reader could use this functionality to access the writer's private data.

4 Security Requirements

In order to effectively execute the UARC application from Mosaic [3], the command script should be able to perform a certain amount of I/O. Below, we list the access rights required to perform I/O in a particular analysis session. We assume that the reader intends to record this analysis session, but not add any annotation data.

1. The right to perform a write operation on the reader's recording directory to add a new recording

2. The right to perform a read operation on any recording, annotation, or data files the reader may possess

3. The right to perform an execute operation on the UARC application file

The following access rights must be revoked to ensure that the reader's security is not violated:

1. The right to perform a read operation on system objects, such as the password file

2. The right to perform a write operation on any objects for which the write operation is not explicitly granted

3. The right to perform an execute operation on any objects for which the execute operation is not explicitly granted

The access rights for the command script should be limited in such a way that the following usability requirements are met:

1. A modification to the access rights of the process executing the command script should not affect the access rights of any unrelated process in the same authentication realm.

2. All objects to be used by UARC should be accessible "in-place." That is, the reader should not be required to move objects to a "safe" location just so that the command script can use them.

These access rights must be enforced for: (1) the process that executes the command script; (2) any descendant process of the command script process; and (3) any service process that is used by any process in (1) or (2).

The only security problem for the writer in this example is to control access to the UARC application. Only members of the UARC project should be able to access the UARC application. Mosaic uses passwords and/or the client's (reader's) internet address to restrict access to web documents. We rely on the use of a password to restrict access to the UARC application in our implementation.

The I/O requirements of UARC are such that read access to a possibly large number of objects is necessary. In addition, the identity of these objects may not be known until after the UARC application has been started. For example, the reader may want to read old data, annotations, or recording files based on what the reader sees in the current analysis session. Therefore, it is not feasible to require that the reader move all the objects that are to be read to a special, public directory prior to every use of UARC. However, only one file is to be executed and write access is needed for only a single directory. The security requirements of the reader dictate that write access should be limited only to the recording files and execute access should be limited only to the UARC application.

[3] We will use UARC as our example, but the arguments made hold for any executable application.

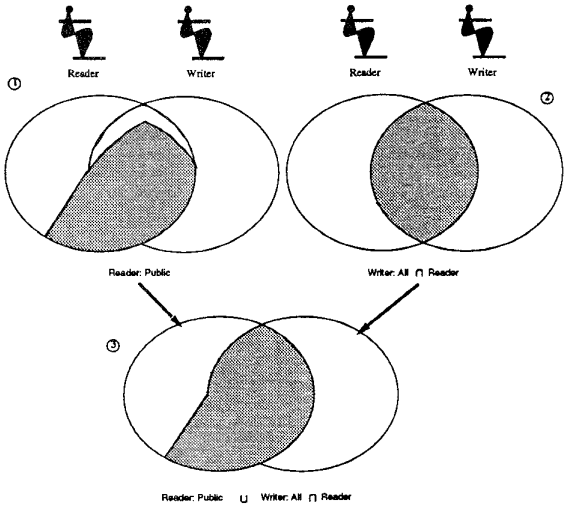

Figure 6: Operation access rights: The reader shares all public rights, and the writer shares all rights that reader also has.

5 Access Control Model

We define a discretionary access control model for specifying the access rights available to a command script. The goal of the discretionary access control model is to enable the reader and writer of the command script to grant access to system objects that are necessary to execute the command script effectively while protecting other system objects from access.

Below, we define the major concepts of the discretionary access control model:

- **Definition 1**: A *writer*, w, is a principal that forwards the command script to the reader for execution. Note that the writer does not necessarily compose the command script.
- **Definition 2**: A *reader*, r, is a principal that executes the command script.
- **Definition 3**: An *object access right* of a command script, $oba \in OBA$, is a tuple, $ob = (obj, OP_{obj})$, where obj is a unique identifier of the object and OP_{obj} is a set of operations (e.g., read, write, execute) that the command script can perform on the object. Object access rights can be either granted or revoked.
- **Definition 4**: A *sharing function* of a principal i for a command script, $sf(i)$, is a function, $sf(i) = s_i$ or $sf(i) = s_i \cap j$ where s_i is a sharing value (one of `none`, `public`, or `all`) that specifies a class of objects accessible to i and j is another principal. For example, if the writer specifies $sf(writer)$ =`all` $\cap reader$, the writer grants permission to all of the writer's objects shared by the reader. The set of objects shared by the writer is shown in #2 of Figure 6.
- **Definition 5**: An *operation access right* of a command script, $opa \in OPA$, is a tuple, $opa = (op, sf(r), sf(w))$, where: (1) op is an operation; (2) $sf(r)$ is the sharing function of the reader for op; and (3) $sf(w)$ is the sharing function of the writer for op. The value of opa specifies that the command script has permission to perform op on the union of the objects represented by the domains $sf(r)$ and $sf(w)$ (see Figure 6). If an opa value is not specified for an operation op, then $opa = (op, none, none)$.
- **Definition 6**: A *command script computation*, c, is a set of processes that execute a command script. In our case, a computation is a set of processes, $p \in P$, including the process that executes the script, its descendant processes, and any service processes these processes use.
- **Definition 7**: *Command script access rights*, ar, is a tuple, $ar = (r, w, OPA, OBA_g, OBA_n)$, where: (1) r is the identity of the reader that executes the command script; (2) w is the identity of the writer of the command script; (3) OPA is a set of operation access right specifications; (4) OBA_g is a set of object access rights granted to the command script; and (5) OBA_n is a set of negative object access rights of the command script. The order of precedence of the access rights specifications is (from highest to lowest): (1) OBA_n; (2) OBA_g; and (3) OPA_g. The command script access rights must be enforced on all processes in the command script computation.

In this discretionary access control model, the access rights of a command script are specified by operation and by object. Operation access rights permit the reader and the writer to limit the operations that can be performed on a class of objects. For example, the read operation can be limited to only the writer's public objects. Object access rights permit the reader and writer to grant or revoke operations on a specific object. For example, execute access may be precluded for `mail`, but read access may be granted to a private object, such as a recording.

In the UARC example, the writer does not provide any access rights to the reader, but the reader needs to limit access. We specify the UARC command script's access rights as shown in Table 1. These access

ar Attribute	*Value*
Reader	UARC_reader
Writer	UARC_writer
OPA	{(read,public,none)}
OBA_g	{(~/UARC,{execute}), (~/recording{read,write})}
OBA_n	{(/etc,{read,write,execute})}

Table 1: UARC command script access rights

rights are interpreted in the following order: (1) operation access rights; (2) object access granted; and (3) object access revoked. First, the $(read, public, none)$ operation access rights provide read access to all the reader's public objects. Since operation access rights are not provided for other operations, these operations are precluded on all objects. The object access granted specifications, (~/UARC,{execute}) and (~/recording{read,write}), override the operation access rights by granting access to the perform the execute operation on UARC and the write operation in the recording directory object, respectively. Note that we do not permit write access to any existing objects in the `~/recording` directory, just the directory itself. The object access revoked specification (/etc,{read,write,execute}) also overrides the operation access rights, by precluding access to the read operation on a public object. Object access rights revoked always supersede those granted if there is a conflict. We address how a reader determines an *ar* value in the Implementation Section.

6 Implementation

In this section, we detail the implementation of a Mosaic script execution prototype that uses our discretionary access control model. The prototype implements the command script execution process shown in Figure 7. The prototype supports two major actions in the process: (1) selection of the command script access rights (tasks 1-7) and (2) enforcement of these access rights as the command script is executed (task 8). Selection of access rights is difficult because it is not possible to identify all the access rights needed to execute a command script by examination. The command script access rights should be enforced even when an application is to be executed.

In our current implementation of the script execution prototype, only the reader's access rights are granted to the command script (i.e., $sf(writer)$ =**none** for all scripts). In order to enable a writer to delegate access rights to the script execution prototype, the writer must be able to grant access rights with a long life time to enable the writer's access rights to be available whenever the reader needs them. The work in [14] addresses this problem.

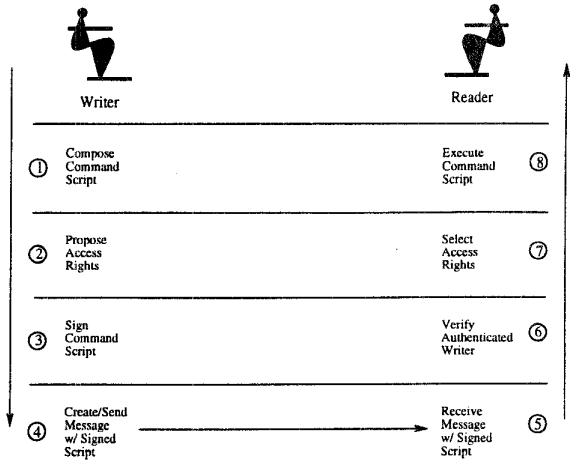

Figure 7: Command Script Execution Process

6.1 Selecting Access Rights

First, the reader must select the access rights for the computation that executes the command script. The reader needs assistance to determine what access rights are required to run the command script, however. A combination of the writer's knowledge of what the command script does and the reader's knowledge of trust in the writer determine the access rights for the command script.

There are three sources of information for determining the necessary access rights for a command script: (1) the script; (2) the writer; and (3) the reader. Since the script may call compiled applications or the reader may want to perform actions with the script not directly specified in the script, it is not possible to read the script and collect the complete set of access rights that are required to execute the script effectively. Our solution is to permit the writer to propose the access rights for the command script and to require the reader to select the permissible access rights. The command script is run with the access rights selected by the reader. Since users are prone to make errors, we add a file that defines the maximum I/O permitted for "safe" interactions. Any access rights that the reader accepts that are outside the safe access rights would require an explicit confirmation by the reader.

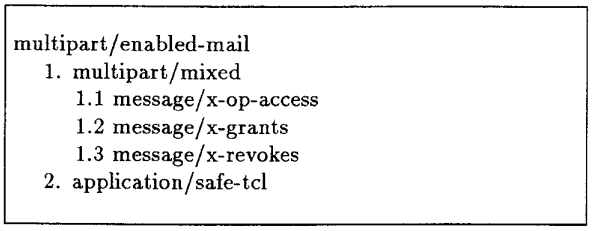

Figure 8: Enabled mail message structure

The prototype provides the reader with: (1) an authenticated identity of the writer; (2) a command script; and (3) the writer's access rights proposal for the command script. The reader uses this information to select the access rights for the command script or refuse execution of the command script. The reader may edit the access rights proposal to either increase or decrease the set of access rights available to the script.

To write a command script that includes an access rights proposal, a writer uses a MIME-compatible mailer such as **mhn** or **metamail** to compose an enabled mail message file that has entries for both the command script and the access rights proposal. We describe the use of an enabled mail message written using **mhn**. We define a enabled mail message of the structure shown in Figure 8. The MIME type of an enabled mail message is **multipart/enabled-mail**. The access rights proposal is specified in the **multipart/mixed** section. **message/x-op-access**, **message/x-grants**, **message/x-revokes** specify the operation access rights, object access rights granted, and object access rights revoked, respectively, for the command script. For reasons that we will describe in the following section, the interpretive language Safe-Tcl [3] is used for writing command scripts. The entry under the MIME type **application/safe-tcl** contains the command script.

The writer signs the message file using a public key cryptosystem (we use the RSA-based system PGP [4] since it has already been integrated with Safe-Tcl) to enable the reader to authenticate the writer. We assume a secure key distribution mechanism exists. Any reader can then verify the identity of the writer. Therefore, the reader can be certain that the message originated from the specified writer (unless another user stole the writer's secret key).

After signing the original message file, the writer creates a new MIME message file with one entry, **application/signed-enabled-mail**, which contains the signed message file that, in turn, contains the command script. This new message file is named **exec.uarc** and is the UARC command script file for the UARC-WWW server.

In order for the reader's machine to process the message file, the reader's profile must be defined with the proper viewers to process the MIME messages. The following entries must be added to the reader's profile:

- **.mime.types**: application/x-uarc uarc
- **.mailcap**: application/x-uarc; xterm -e show -file %s
- **.mh_profile**: mhn-show-application/signed-enabled-mail: decode-pgp %f
- **.mh_profile**: mhn-show-multipart/enabled-mail: %pem_play3 %F

The **.mime.types** entry permits Mosaic to recognize that a file with a **.uarc** extension corresponds to the **application/x-uarc** MIME type. The **.mailcap** entry specifies how MIME entries of certain types are to be processed. The application **show** processes **mhn** messages, and the new **xterm** provides an outlet for messages generated by **show**. The **.mh_profile** entries specify the different scripts that are used by **show** to process different MIME entries. **decode-pgp** is used to verify **application/signed-enabled-mail** entries. **em_play3** processes the verified enabled mail message. Note that the reader's profile must be secured in order to ensure that the profile can be trusted.

When the **exec.uarc** file is downloaded by Mosaic, the **show** command is executed to process the message file. Since the MIME type of the message file is **application/signed-enabled-mail**, the **decode-pgp** script is executed. This script verifies the identity of the writer and stores this identity in the variable **$PGP_SIGNATURE**. Note that if the writer does not sign the message file, then the variable **$PGP_SIGNATURE** is not set. If this variable does not have a value, then the command script application assumes that the writer does not have access to the machine. The use of a shell environment variable to store the authenticated identity of the writer will be secure if the operating system can protect the address space of this shell from tampering. We have assumed a secure operating system for this reason.

The **decode-pgp** script contains a second call to **show** to display the enabled mail message. The MIME type of this message is **multipart/enabled-mail**, so the script **em_play3** processes this message. This

[4] PGP is a trademark of Phillip Zimmermann

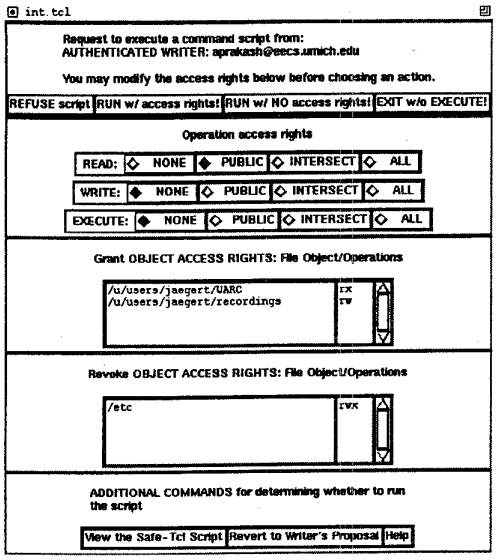

Figure 9: Select access rights

script triggers Safe-Tcl's untrusted interpreter to obtain the access rights for the UARC command script from the reader and then to execute this command script with those access rights. The access rights are obtained using the Tcl interface displayed in Figure 9. This window is called from the Safe-Tcl initialization function `.safetclrc`. This window initially displays the writer's proposal for the access rights. If the writer failed to supply a proposal, then the initial proposal is for no access rights to be given to the script (i.e., an operation sharing value of **none** for all operations). The reader may edit any part of the proposal, execute the command script, or refuse to execute the command script. The access rights are bound to a Safe-Tcl variable that is used to setup the command script execution environment. In the next section, we describe several implementations of a mechanism for setting the access rights for the command script and executing the command script while enforcing these access rights.

Note that in addition to the mailer, the cryptosystem, and the operating system, the **decode-pgp** and system's `.safetclrc` scripts must be trusted by the reader. Therefore, no user should be able to replace either of these scripts. Otherwise, a malicious writer could write a script that substitutes the reader's access rights with whatever rights the writer wants. Tighter integration of the functionality of these scripts with Safe-Tcl would enable us to reduce the number of trusted programs required.

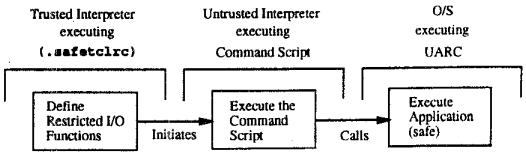

Figure 10: Safe-Tcl script execution

6.2 Executing the Command Script

The final step in the command script execution process is to execute the command script using only the access rights selected by the reader. We examine three implementations of this step: (1) Safe-Tcl's untrusted interpreter; (2) Safe-Tcl's untrusted interpreter integrated with Kerberos version 5; and (3) Safe-Tcl's untrusted interpreter and the Taos distributed operating system. Safe-Tcl's untrusted interpreter can restrict access to objects from the command script, but cannot restrict access once an application is initiated from the command script. Kerberos enables the command script access rights to be enforced by applications that are also Kerberos-aware. In order to ensure enforcement of the command script access rights for all applications, the operating systems' authorization mechanism must enforce those rights. We examine using the Taos distributed operating system.

6.2.1 Safe-Tcl

The Safe-Tcl implementation works as shown in Figure 10. First, we load our restricted I/O functions into Safe-Tcl's untrusted interpreter. This is done using the Safe-Tcl initialization file, `.safetclrc`. Safe-Tcl permits functions defined in the trusted interpreter to be loaded into the untrusted interpreter. Since `.safetclrc` is run in the trusted interpreter, it can define the restricted I/O functions and load them into the untrusted interpreter. The command script access rights must be hard-coded into these I/O functions to prevent tampering by the writer's script. If the access rights are stored in a global variable, the writer's script could modify the value. We then process the command script using the modified, untrusted interpreter. If an application is executed from the command script the untrusted interpreter's authorization mechanism is no longer applicable, so only "safe" applications can be executed.

Restricted I/O is permitted in Safe-Tcl's untrusted interpreter using two functions, **safe_open** and **safe_exec**. These two functions both authorize a request for access to an object against the command script's access rights. If the authorization is success-

```
/*
* Safe-Tcl function that opens an object for
* (one of r, r+, w, w+, a, a+, x) if access rights
* are authorized. Returns a file descriptor if
* successful. Else, no value is returned.
*/

safe_open(obj, access)
{
   /* if safe_authorize obj and access w/ access_rights
   * open obj */
   if (safe_authorize (obj, access))
      return open(obj, access)
   else reporterr()
}
```

Figure 11: The **safe_open** function

ful, **safe_open** and **safe_exec** call the Safe-Tcl I/O commands **open** and **exec**, respectively. **open** enables an object to be read or written, and **exec** executes an application. In addition, we define safe versions of the functions **puts** and **gets** to read and write objects. These commands prevent I/O using the global file descriptors, **stdout**, **stdin**, and **stderr**. Therefore, all I/O from the command script is performed using objects opened by **safe_open** and executed by **safe_exec**.

safe_open takes a name and an access mode specification for an object and returns a file descriptor on a successful open. No value is returned upon a failed open. Pseudocode for **safe_open** is shown in Figure 11. Whether an open is permitted is determined by the function **safe_authorize** which compares the request to the command script's access rights.

Before detailing **safe_authorize** let us examine the authorization requirements of **safe_exec**. **safe_exec** takes an application and an argument list and executes the application with the argument list if the execution is authorized (see Figure 12). Authorization is more complex for **safe_exec** than it is for **safe_open** because the execution of an application cannot be controlled by the untrusted interpreter. Therefore, about the only way to ensure that an execution is safe, is to compare the structure of the command to a catalog of safe calls. This requires that a system administrator catalog the safe command call structures. This is likely to be a tedious and error-prone task. We find this part of the solution less than satisfactory.

The pseudocode for the **safe_authorize** function is shown in Figure 13. **safe_authorize** converts the access mode to a set of operations and authorizes access to perform those operations on the object. In Tcl, the **r+** access mode means that the object is to

```
/*
* Safe-Tcl function that executes an application. An
* application is specified by its complete file name
* and arguments. If access is authorized or user
* approves the execution of a "not unsafe" application,
* the application is executed. Return result.
*/

safe_exec(appl, args)
{
   /* Hard-coded safety specs */
   safety_specs = safe_calls_string
   /* Determine if the appl is accessible and safe */
   if (safe_authorize (appl, execute)) &&
      (safety_specs are satisfied by appl and args))
      result = exec(appl, args)
   /* Give the user the option to OK the fn's exec */
   else if (safety_specs are satisfied by appl and args) &&
         (reader has execute privilege for appl) &&
         (user_ok (writer, appl, args))
      result = exec(appl, args)
   else
      reporterr()
   return result
}
```

Figure 12: The **safe_exec** Function

```
/*
* safe_authorize: authorize obj and access using
* access rights for command script.
*/

safe_authorize(obj, access)
{
   /* Access rights are hard-coded to prevent spoofing s*/
   OBA_n = object access rights revoked
   OBA_g = object access rights granted
   op_groups = A set of group names from map for each op
   op_principals = A principal name from map for each op
   /* Convert access to ops - prevent spoofing */
   find ops in access
   foreach op in ops {
   /* oba_n = set of negative rights for obj * /
      if (oba_n ∈ OBA_n for obj && op is precluded by oba_n)
         return FALSE
      else {
         /* oba_g = set of positive rights for obj */
         unless (oba_g ∈ OBA_g for obj && oba_g grants op) {
            groups = op_groups(op)
            prin = op_principals(op)
            if ((negated op on obj for any prin or groups) ||
               (no prin or groups has op access to obj))
               return FALSE
         }
      }
   }
   return TRUE
}
```

Figure 13: The **safe_authorize** function

OPA Value	*Principal*	*Groups*
None	None	None
Public ∩ Writer*	Anyuser	None
Public ∩ Writer	None	None
Public	Anyuser	None
All ∩ Writer*	Anyuser	Writer ∩ reader
All ∩ Writer	None	None
All	Reader	Reader's groups

Table 2: Conversion from OPA to authorization principals and groups (* – Writer is a known principal in the reader's realm)

be opened to permit both read and write operations on the object. Access rights are checked in the following order: (1) OBA_n; (2) OBA_g; and (3) OPA. The reason for this is performance. We can use indexing techniques to quickly retrieve the oba_n and oba_g values for a specific object. Each OPA value is converted to the name of a principal and possibly a set of groups. If either the principal or one of the groups can be authorized to perform an operation on the object, then this operation is authorized. The mapping between OPA values and principals and groups is shown in Table 2. For example, an operation access rights value of `public`, corresponds to the principal that has all public rights (e.g., `anyuser` in the Andrew File System and `nobody` in Unix).

The `safe_authorize` function cannot be implemented as a procedure in the untrusted interpreter, however. This is because the writer's script could redefine the function such that the function authorizes every request. Thus, `safe_authorize` can be implemented in one of two ways: (1) by embedding the `safe_authorize` code into `safe_open` and `safe_exec` or (2) by extending the Safe-Tcl trusted interpreter by adding `safe_authorize` as an unsafe function. The first solution increases the complexity of the code for both `safe_open` and `safe_exec`. The second solution requires modifying the code for Safe-Tcl's trusted interpreter which renders it unportable. We have opted for the first solution.

6.2.2 Safe-Tcl and Kerberos

A second implementation option is to integrate Safe-Tcl's untrusted interpreter with Kerberos version 5 [8]. This is referred to as "kerberizing" [12] Safe-Tcl's untrusted interpreter. Kerberizing Safe-Tcl's untrusted interpreter enables the untrusted interpreter to execute other kerberized applications that also have the ability to enforce the command script

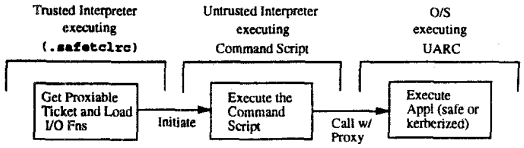

Figure 14: "kerberized" Safe-Tcl script execution

access rights. These other applications must have a compatible representation for access rights in order to perform authorization correctly, however. This is not necessarily the case, at present, so we must ensure that the command script starts only kerberized applications that can understand the command script access rights.

Transfer of access rights between the command script process and an application is accomplished using a special type of Kerberos ticket called a proxy [11]. A *proxy ticket* is a ticket that a principal creates to delegate its access rights to another principal. Using Kerberos proxies, a principal can delegate any subset of its access rights to another principal. The actual access rights to be delegated are specified in the `authorization data` field of the ticket. The format of this field is determined by the application, so we use the command script access rights format of our access control model for the `authorization data` field.

Conceptually, command script execution using a kerberized untrusted interpreter would proceed as follows (shown in Figure 14): (1) `.safetclrc` obtains a proxiable ticket from Kerberos that enables the untrusted interpreter to create proxy tickets to execute kerberized applications; (2) also `.safetclrc` loads the safe I/O functions including a revised version of `safe_exec` that can recognize other kerberized applications; (3) the command script is then executed; and (4) if the command script access rights allow, safe or compatibly kerberized applications can be executed from the command script. If a kerberized application is to be executed, the untrusted interpreter must obtain a proxy ticket from Kerberos using the proxiable ticket obtained earlier. The untrusted interpreter passes the proxy ticket and any input data to the application. The application uses the access rights specified in the proxy ticket to authorize the application's I/O.

The sequence of actions listed above are implemented by a sequence of messages between Safe-Tcl's untrusted interpreter and either Kerberos or other kerberized applications. Figure 15 shows the sequence of messages. Kerberos message specifications are quite complex, so some of the details are omitted from the

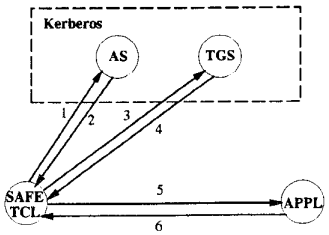

Figure 15: Messages in "kerberized" Safe-Tcl implementation

succeeding description for simplicity reasons. Complete message specifications can be found in RFC 1510 [8].

Messages 1 and 2 implement the initial authentication exchange of the reader with Kerberos to obtain a proxiable ticket. When command script access rights are selected (in the `.safetclrc` script), the reader must authenticate with Kerberos if any applications could be executed from the command script. In message 1, the reader submits the reader's secret key (using an extended version of the interface shown in Figure 9) and the specifications for a proxiable ticket to the Kerberos authentication server (AS). The AS returns a proxiable ticket-granting ticket from which proxy tickets can be created in message 2. Inside the ticket-granting ticket is a session key that is used for encrypting and decrypting all future messages exchanged with Kerberos.

Once the session key and the proxiable ticket have been obtained, `.safetclrc` loads the restricted I/O functions. With the exception of `safe_exec`, the definitions of the I/O functions are unchanged from the previous implementation. `safe_exec` is revised to permit kerberized applications to be executed (see Figure 16). The first condition (specified in bold in the Figure) lists the requirements for executing a kerberized application: (1) the command script access rights permit execution of the application and (2) the application is registered as a compatibly-kerberized application. The second requirement limits the set of kerberized applications that can be used to only those that also can also enforce the command script access rights. The session key and proxiable ticket are hard-coded in the `safe_exec`, so they may be used to create proxy tickets in `krb5_exec`.

If an application is kerberized and the command script access rights permit its execution, the function `krb5_exec` is called to execute the application. The pseudocode for `krb5_exec` is shown in Figure 17. First, a request for a proxy ticket is generated using the proxiable ticket and the access rights. The

```
/*
* Safe-Tcl function that executes either a kerberized,
* safe, or user-approved application. An
* application is specified by its global object name
* and arguments. Return result.
*/

safe_exec(appl, args)
{
   /* If no proxiable tkt, don't execute appl */
   unless (PROXIABLE_TKT) reporterr()
   /* Hard-coded safety specs and kerberized appls */
   safety_specs = safe_calls_string
   kerberized_appls = krb_appls_string
   /* Is appl is accessible and kerberized? */
   if ((safe_authorize (appl, execute)) &&
      (member(appl,kerberized_appls)))
      result = KRB5_EXEC(appl, args, S_KEY,
                         PROXIABLE_TKT, RIGHTS)
   /* Determine if appl is accessible and safe */
   else if (safe_authorize (appl, execute)) &&
      (safety_specs are satisfied by appl and args))
      result = exec(appl, args)
   /* Give the user the option to OK the fn's exec */
   else if (safety_specs are met by appl and args) &&
         (reader has execute privilege for appl) &&
         (user_ok (writer, appl, args))
      result = exec(appl, args)
   else
      reporterr()
   return result
}
```

Figure 16: The Kerberos-aware **safe_exec** Function

proxiable ticket is used to authenticate the source of a request. The access rights specify the authorization data for the proxy ticket. The format of the access rights is the same as in Safe-Tcl implementation above. The access rights are represented by a principal, a set of groups, a set of object access rights granted, and a set of object access rights revoked. Message 3 in Figure 15 sends this request to the ticket-granting service (TGS). Unless there is an error, message 4 returns a new proxy ticket that contains the command script access rights in its `authorization data` field. The proxy ticket is encrypted using the application's session key from its ticket-granting ticket, so only the application can use the proxy.

Once the proxy ticket is obtained, `krb5_exec` uses this ticket to authenticate the untrusted interpreter to the application. In message 5, the encrypted proxy ticket and an authenticator is sent to the application. The proxy ticket contains the authorization data for the proxy and session key for the application (different than the session key between the reader and the TGS). The application uses the session key to verify that the authenticator is from the reader. The application authenticates itself to the untrusted interpreter in message 6. If both authentications succeed, then the application is run. Using the authorization data

```
/*
* Execute a kerberize service. Delegate the proxy to
* the kerberized service.
*/

krb5_exec(appl, args, s_key, proxiable_tkt, access_rights)
{
   /* Create KRB_TGS_REQ msg to obtain proxy ticket */
   KRB_TGS_REQ from application, proxiable_tkt,
        access_rights, etc.
   Encrypt KRB_TGS_REQ using the session_key
   /* Msgs 3&4: Get proxy from TGS */
   send_receive(KRB_TGS_REQ, TGS, resp)
   /* If the resp is OK, create KRB_AP_REQ */
   if (resp == KRB_ERROR) reporterr ()
   else {
      Decrypt resp using the session key
      Obtain proxy_ticket for appl from resp
      Create authenticator from sub-session key
      KRB_AP_REQ from appl,args,proxy_ticket,
         authenticator,etc.
      /* Msgs 5&6: authenticate w/ appl */
      send_receive(KRB_AP_REQ,application,resp)
      /* Handle responses from the application */
      if (resp == KRB_ERROR) reporterr()
      else {
         Decrypt resp using the sub-session key
         if (resp is OK) perform application
      }
   }
}
```

Figure 17: The **krb5_exec** Function

in the proxy, the application can limit the access rights of the command script's application appropriately.

The benefit of integrating Safe-Tcl's untrusted interpreter with Kerberos is that kerberized applications could be executed that enforce the command script access rights and the command script access rights are transferred safely (i.e., with strong authentication). However, a requirement on the kerberized application is that they must be able to interpret the access rights data. At present, there is neither a standard representation for access rights data in a proxy nor a significant body of applications that are kerberized. Therefore, this implementation provides only a slight improvement over the first implementation. We do think that a secure operating system that is integrated with Kerberos version 5 may be a viable solution in the future, however.

6.2.3 Safe-Tcl and Taos

To enforce the command script access rights for any arbitrary user application, the operating system must perform the access rights enforcement. In this implementation, we use Safe-Tcl's untrusted interpreter to execute the command script and the Taos distributed operating system [15] to authorize the command script's I/O. We select Taos because it provides

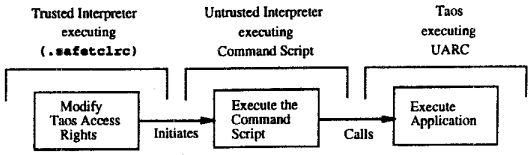

Figure 18: Safe-Tcl and Taos script execution

an extensive model for authentication in a distributed environment. Taos' representation of process access rights does not include all the fields necessary to store the information from our access control model, however. Therefore, we suggest a new data type to represent our access control model's information and a primitive that manipulates this data type. Also, some modifications to Taos' authorization mechanism are necessary to use this new data type.

The mechanism for executing a command script using Safe-Tcl's untrusted interpreter on the Taos distributed operating system is shown in Figure 18. In this implementation, all authorization is performed by the operating system. First, the Safe-Tcl initialization function `.safetclrc` calls the Taos primitives to alter the access rights of the command script process to those obtained from the reader. All processes spawned by the command script process inherit the command script process' access rights. In addition, any services that are requested by the command script or its applications are delegated the same access rights. Therefore, as long as these access rights are unchanged, the access rights of the command script are enforced by this implementation even when applications are executed. `.safetclrc` also adds the functions `safe_open` and `safe_exec` (as well as `safe_gets` and `safe_puts`), but these functions simply call `open` and `exec`, respectively. Safe-Tcl does provide other security capabilities, such as CPU time limits, that are useful for this application.

We now examine the details of the implementation. At the application-level, Taos represents the access rights of a process using a set of `Auths`. Each `Auth` represents a principal (possibly compound) that the process has access rights for. Taos provides primitives for obtaining and modifying the set of `Auths` for a process:

- **Self**(): *Auth*: Returns the `Auth` of the current process.
- **Inheritance**(): *ARRAY of Auth*: Returns the `Auths` that the process inherits from its parent.
- **New**(*name, password:TEXT*): *Auth*: Create a

new credential for **name** if the **password** is correct.

- **AdoptRole**(*a:Auth; role:TEXT*): *Auth*: Restrict the authority of the process to that of the **role**.
- **Delegate**(*a:Auth; b:Prin*): *Auth*: Create a credential that can be used by the principal **Prin**.
- **Claim**(*b: Auth; delegation: Prin*): *Auth*: Claim a credential that has been created by the principal **Prin**.
- **Discard**(*a: Auth; all: BOOL*): *void*: Make an **Auth** invalid. If **all** is true then the **Auth** becomes invalid in all the processes that inherited it.

Access rights of a process are restricted in Taos using the primitive **AdoptRole**. This primitive weakens the authority of an **Auth** by associating it with a role that has fewer access rights. A role corresponds to an identity that has a special access control policy, usually some semantically meaningful identity such as system administrator [5, 16]. Roles are static entities that are managed by system administrators. If the number of operations in our access control model is fairly small, we find roles to be a reasonable model to represent the access rights for the operation access right values: **public**, **none**, and **all**. For example, a role can be created for each combination of **public**, **none**, and **all** with the **read**, **write**, or **execute** operations. In the UARC-WWW server example, the operation access rights value of $\{(public, read, none)\}$ can be implemented by a role that only has **read** access to any file that **anyuser** has access and negative rights for **write** and **execute**.

Roles are not an appropriate representation for implementing operation sharing that requires an intersection of two principals' access rights or the object access rights of our access control model, however. The access rights granted by intersection also depend on the writer of the script. Since the reader may execute several scripts with different writers, we may need several different roles. In addition, object access rights cannot always be predicted, even if we know the application. The reader may want to use different objects for different runs of the same application or may provide more access rights to the application as the reader's trust in the application and the writer increase.

The generation of a role on demand would be the cleanest solution to this problem, as it would be consistent with theory upon which the Taos authentication mechanisms are built [9]. However, the creation

```
/*
* set_cmd_script_rights: Reset the Auths of
* the current process (i.e, the Safe-Tcl interpreter) to be
* those specified for the command script by the reader.
* Called from .safetclrc (trusted interpreter).
*/

set_cmd_script_rights(role, grps, objRights, negObjRights)
{
   /* Get default Auth */
   a:Auth = Self();
   /* Create DynPrin from input data */
   AdoptDP(a, role, grps, objRights, negObjRights);
   /* Remove other Auths from process */
   foreach auth in Inheritance() {
      unless (auth ≠ a)
         Discard(auth,TRUE);
   }
}
```

Figure 19: The **set_cmd_script_rights** function

of a role would require the modification of a large number of access control lists (ACLs), so the performance would be poor. In lieu of this, we propose to create a subtype of the Taos principal data structure **Prin**, called **DynPrin**, which represents the information that would be used to create a role dynamically. Below, we define the data structure for **DynPrin**:

- **principal name**: name of base principal (e.g., a role)
- **groups**: a set of group names to which **DynPrin** belongs
- **obj_rights**: OBA_g of the command script access rights
- **neg_ object_rights**: OBA_n of the command script access rights

Since a **DynPrin** defines the access rights of the process to each object, a role could be created in a straightforward manner from a **DynPrin**. Therefore, a **DynPrin** is functionally equivalent to a role. As a result, it seems reasonable that **DynPrin** is a consistent extension of the Taos authentication model. However, a formal proof is beyond the scope of this paper.

We modify the definition of the **Auth** data structure to permit a principal to be represented by either a **Prin** or a **DynPrin** data type. A new primitive called **AdoptDP**, is added to restrict the access rights of an **Auth** using a **DynPrin**. **AdoptDP** takes the following arguments:

1. **a**: *Auth*: The **Auth** whose access rights are being restricted.

2. **role**: *TEXT*: Name of the role being assumed. The value of the `principal` attribute in `DynPrin` is set to `role`.

3. **groups**: *ARRAY of TEXT*: A set of group name strings. The value of the `groups` attribute in `DynPrin` is set to `groups`.

4. **objRights**: *ARRAY of* oba_g*'s*: A set of object access rights granted. The value of the `objRights` attribute in `DynPrin` is set to `objRights`.

5. **negObjRights**: *ARRAY of* oba_n*'s*: A set of object access rights revoked. The value of the `negObjRights` attribute in `DynPrin` is set to `negObjRights`.

`AdoptDP` replaces the value of the `Auth`'s principal with the `DynPrin` created from the `role`, `groups`, `objRights`, and `negObjRights` arguments. In our implementation, `.safetclrc` calls the function `set_cmd_script_rights` shown in Figure 19 to create the appropriate `DynPrin` given the command script access rights selected by the reader. In the UARC-WWW server example, the following argument values are used:

1. **a**: The default `Auth` for the process returned by a call to the primitive `Self`.

2. **role**: "PUBLIC-READ" (public access rights for only the read operation)

3. **groups**: NULL

4. **objRights**: OBA_g value for the UARC-WWW server (see the Access Control Model Section)

5. **negObjRights**: OBA_n value for the UARC-WWW server (see the Access Control Model Section)

In general, the authorization mechanism is more complicated for the Taos implementation because: (1) of the ability to specify cascaded delegations in an `Auth` and (2) a process may have multiple `Auths`. In Taos, an access is granted if there exists a principal in the ACL with the rights requested that one of the process' `Auths` speaks for [9]. Principals and groups in a `DynPrin` can be checked using the Taos authorization mechanism, but we must add the ability to check object access rights. The semantics for checking an object access right is that if any `DynPrin` in a chain of principals contains a `negObjRight` that precludes any requested operation on the current object, then the access is denied. Otherwise, if any `DynPrin` contains an `objRight` that grants access to perform all the requested operations on the current object, then access should be granted. Therefore, a modified version of the function `safe_authorize` that first checks `negObjRights` and `objRights` and then applies the Taos mechanism to authorize compound principals can be used to authorize access. To make this test more efficient, we may like to cache information about an `Auth`'s constituent `DynPrins` on the `Auth` rather than having to examine an entire chain of delegations.

Since this authorization mechanism is used by all calls to `open` and `exec` in Taos, we obtain a greater guarantee that the command script access rights will be enforced. However, if we use an application on a machine running a different operating system, the access rights data is of no value.

The security of this implementation depends on preventing the computation from obtaining any additional access rights (`Auths`). The primitives available for obtaining additional access rights are `New` and `Claim`. If the writer has access to the machine, then the writer can create additional `Auths` for the command script computation by using either: (1) `New` to generate a new `Auth` or (2) `claim` to obtain an `Auth` from a writer process that is executing concurrently. Although these `Auths` only enable the writer to gain access to objects that the writer could already access through normal means (i.e., public objects), the access control model should be able to prevent this access if specified by the reader. The representation of a process must be modified to indicate that its set of `Auths` are immutable to prevent obtaining these new `Auths`.

7 Conclusions and Future Work

We defined a discretionary access control model to represent the access rights available to the command script when it is executed. This model permits the reader and writer of a command script to flexibly restrict the access rights that are authorized to the command script's processes. This is important because the access rights needed for executing a command script can vary greatly depending on the application and the writer. We use this model in a prototype implementation of a system that executes command scripts that are obtained using Mosaic. The prototype implementation only supports the reader's ability to grant access rights at present.

We examine three separate implementations for enforcing the command script access rights in the prototype. We see some synergy between the three im-

plementations that leads us to believe that a combination of the three implementations may be appropriate in the future. A command script execution language, such as Safe-Tcl, is necessary to execute the command scripts. We expect that as people try to evolve scripts into agents (e.g., Telescript) more demands will be placed on the interpreter, so an untrusted interpreter that can ensure that these constraints are upheld in an environment of mutual distrust becomes more important. Kerberos' restricted proxies provide an effective mechanism for specifying and transferring the command script access rights, but Kerberos must be integrated with a secure operating system to enable script execution to be trusted. Taos provides an extensive authentication model, but we needed to add what amounts to restricted proxies in Taos (the `DynPrins`) to implement our access control model. Therefore, some combination of the functionality of the three systems should be examined. We also plan to enable writers to delegate access rights safely to the command script.

Acknowledgements

Nathaniel Borenstein has provided us with valuable guidance on the use of Safe-Tcl. Discussions with Aviel Rubin and Peter Honeyman also contributed to this paper. This work is supported in part by the National Science Foundation under the cooperative agreement IRI-9216848.

References

[1] Trusted Mach kernel primer. Trusted Information Systems, Inc., 1991.

[2] N. S. Borenstein. Computational mail as a network infrastructure for computer-supported cooperative work. In *CSCW 92 Proceedings*, pages 67–74, 1992.

[3] N. S. Borenstein. Email with a mind of its own: The Safe-Tcl language for enabled mail. In *ULPAA '94*, 1994. Available via anonymous ftp from ics.uci.edu in the file mrose/safe-tcl/safe-tcl.tar.Z.

[4] R. et. al. Clauer. A prototype upper atmospheric collaboratory (UARC). AGU Monograph: Visualization Techniques in Space and Atmospheric Sciences. In press.

[5] 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.

[6] Y. Goldberg, M. Safran, and E. Shapiro. Active Mail – a framework for implementing groupware. In *CSCW 92 Proceedings*, pages 75–83, 1992.

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

[8] J. T. Kohl and B. C. Neuman. The Kerberos network authentication service, September 1993. Internet RFC 1510.

[9] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10(4):265–310, 1992.

[10] N. R. Manohar and A. Prakash. The session capture and replay paradigm for asynchronous collaboration. Submitted to ECSCW'95.

[11] B. C. Neuman. Proxy-based authorization and accounting for distributed systems. In *International Conference on Distributed Computing Systems*, pages 283–291, 1993.

[12] B. C. Neuman and T. Ts'o. Kerberos: An authentication service for computer networks. *IEEE Communications*, pages 33–38, September 1994.

[13] J. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.

[14] A. Rubin. *Nonmonotonic Cryptographic Protocols.* PhD thesis, University of Michigan, Ann Arbor, 1994.

[15] C. Thacker, L. Stewart, and E. Satterthwaite. Firefly: A multiprocessor workstation. *IEEE Transactions on Computers*, 37(8):909–920, August 1988.

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

[17] J. E. White. Telescript technology: The foundation for the electronic marketplace. General Magic White Paper.