Implementing Execution Controls in Unix

Todd Gamble
WilTel Network Services

Implementing Execution Controls in Unix

Todd Gamble – WilTel Network Services

ABSTRACT

Current implementations of UNIX offer security features in the form of discretionary access controls (DACs). DACs are implemented with file access permissions and access control lists (ACLs). Unfortunately, neither of these facilities provide for access control to active processes. In order to provide many users access to a process (and its associated data) the current practice at our site is to establish a group account, where members on a project team share the login and password for an application. This practice is both insecure [cur90][fer93], and a violation of our site's security policies.

This paper describes the implementation of a new tool, `medex`, which eliminates the need for group login accounts. `Medex` mediates the access of users to privileged accounts and executables. The history behind our use of group accounts and a complete methodology for UNIX application management are presented. Details of the implementation of `medex`, including its interaction with the existing security features of UNIX, are given. The tool utilizes execution control lists (ECLs) as a means to allow controlled execution of programs under accounts other than the current login. `Medex` also re-authenticates the user's password upon each instantiation and maintains an audit trail via log files or the use of the UNIX *syslog* facility. A complete project management example utilizing `medex` is given along with a comparison to related tools.

Site Description

WilTel is the forth largest telecommunications company in the U.S. The company's nationwide enterprise network has an installed base of approximately 200 UNIX platforms, including UNIX variants from DEC, HP, IBM, NeXT, SCO, Sequent, and Sun. The user base is in the thousands with the largest system supporting approximately 600 accounts. Typical UNIX application projects include distributed databases, device control and data acquisition, network monitoring, and customer access.

History

Many projects developed at WilTel have inherited support systems which rely upon the use of group accounts. Project teams share the password to a single login account that is used by all team members to manage an application. Team members login or *su*(1) to a special account that is used to start up the application or to modify data owned by the application. While logins or *su*(1) usage can be tracked via conventional accounting records on the machines, the commands executed by the user once in the account are limited only by the access authorizations for the special account, i.e., not those of the specific user. Delegation of authority has little granularity; either a user has the password to the special account or not. Propagation of the password from user to user is untraceable. Modification of the password (e.g., aging) is difficult since password changes must be scheduled with all users of the special account.

Another common practice is to establish ''check out'' logins. A special account is created that can perform some privileged function. Users ''check out'' the account by obtaining its current password. After each use, the password is changed. This eliminates the problem of password propagation, but still does not provide a means to limit the user's actions once access to the special account is given.

Requirements

Our objective is to eliminate the need for *group accounts*. They are difficult to maintain and are insecure. However, we still need the ability to define a *privileged* set of users who are capable of accessing an application project's data as well as its processes during execution. Additionally, the number of access points to the project data should be controlled while providing a complete audit trail of all accesses.

A separation of duties should be established such that authorizations to the system may be made by *security administrators* while system maintenance is performed by *system administrators*. The system should enable security administrators to define strict controls on precise commands that could be executed by a specific user with the account privileges of another user. The system should be portable across all the UNIX environments. Finally, any tools developed should be integrated with the vendor's existing security system.

Access Controls

Most of the UNIX versions at our site implement access controls using industry standard Discretionary Access Controls (DACs).

DACs are defined as follows:

> *The [system] shall define and control access between named users and named objects (e.g., files and programs) in the ... system. The enforcement mechanism (e.g., self/group/public controls, access control lists) shall allow users to specify and control sharing of those objects by named individuals or defined groups or both.* [dod85]

UNIX offers DACs in the form of file permission bits, i.e., the self/group/public controls. The permission bit field is located within the file's *inode*. An inode is a data structure that defines a file within the UNIX hierarchical file system. The permission bit field (see Figure 1) includes access flags for three classes of users: user (self), group, and other (public). Read, write, and execute permissions (or *modes*) for a file are specified by turning on (or off) bits within the bit field.

User			Group			Other		
read	write	exec	read	write	exec	read	write	exec
8	7	6	5	4	3	2	1	0

Figure 1: UNIX file permission bit field

Some UNIX implementations offer additional DAC mechanisms in the form of Access Control Lists (ACLs)[hew91]. ACLs offer more selectivity in access control than the standard file permissions. ACLs allow the file owner or superuser to permit or deny access to a list of users, groups, or combinations thereof. ACLs are supported as a superset of the UNIX operating system DAC mechanism for files, but not for other objects such as inter-process communication (IPC) objects.

An ACL consists of sets of (*user.group,mode*) entries associated with a file that specify permissions. Each entry specifies a set of access permissions for one *user.group* pair, including read, write, and execute (or search). In an ACL, user and group IDs can be represented by their mnemonic user and group names or by their numeric user ID number (UID) or group ID number (GID) as found in the `/etc/passwd` file. Two special symbols may also be used in the ACL entry to simplify the lists:

```
% — no specific user or group
@ — the current file owner or group
```

While DACs control access to file objects, no provision is made for objects of other types (e.g., active processes). Also, all access under the DAC model is centered around the user (or group). The level of granularity on controls does not extend to specifying which programs may be used to access a file object.

Execution Controls

Bacic [bac89] has proposed the Process Execution Control (PEC) model as a means to extend the security system in modern operating systems. PECs introduce additional steps in security by restricting the list of executables that have access to a file and by restricting which user can invoke which executable. These two restrictions are designed to preserve the *integrity* of system objects. Integrity is maintained if objects are manipulated only by *authorized* and *known* entities that leave the object in a *consistent* state after manipulation.

Using PECs, each object in the system is tagged with an Execution Control List (ECL), that describes *which users* may read/write an object and *which program* must be invoked to do so. This hides the object from the user while still allowing the user to manipulate the file with designated programs.

PECs view all files on a given system as *objects* whether or not they are executable. An *executable* is a binary encoded file that may be passed to the operating system for execution. A *subject* is the combination of a known system user and an executable that is accessible to the user. Objects are manipulated by subjects and subjects may in turn be manipulated by other subjects, making them an object from the new subject's viewpoint. Every object in the system defines a *domain* of control around itself. A domain is described by means of the ECL specifying which subjects may manipulate an object. An ECL entry is represented as a *<user,program,data>* triple authorizing a specific user to access a data object with a designated program.

PECs have been implemented in a version of Tunis[1] [bac90]. The kernel of Tunis was modified to consult ECLs when system calls are made for file access. The lists consist of *<user,program>* pairs that are attached to file objects via their inode in a fashion similar to ACLs. These pairs define the subjects that may access the object. The Tunis system kernel functions as the mediation point between subjects and file objects by consulting the ECL attached to the file's inode whenever a system call (e.g., *write*) is made to access the file.

The PEC model extends controls to restrict access to data to pre-defined user and program pairs, but does not include a specification for access control to processes.

[1] Tunis is a UNIX compatible operating system developed at the University of Toronto.

Medex Design

The existing DACs in UNIX and the PEC model provide most of the necessary foundation to develop a system which satisfies the original requirements. Since modification of the UNIX kernel as done for implementation in Tunis is not an option at our site, a solution based on a combination of an application program and careful configuration of the available control mechanisms is used. The application program is implemented as `medex`, an execution control facility that mediates access between users and privileged objects. The system configuration requirements are the implementation of a control policy for project resources on the system.

object: Either a file on the system or the invocation of an executable file as an active process.
method: An executable that is authorized to access a particular object, i.e., the executable is a *method* of the object[2].
subject: The combination of a known system user and a method that is accessible to the user.

Figure 2: Definitions for `medex` control facility

Figure 2 defines the `medex` control elements. Users on the system are authorized to access objects via defined methods. With proper configuration, the superuser is the only user able to subvert the controls imposed by the `medex` facility.

Normally, the objects of concern are those which are part of a particular application project. A *project* can be defined as the *class* of all objects that represent parts of the project on the system. In order to identify objects within a class, a project account is created on the system. The UID of the project account is used as a class identifier for the project's objects. All file objects within a project class should be owned by the UID of the project account. In addition, the project account should disallow direct logins, i.e., its password should be disabled. This eliminates access to the project's objects except for privileged intermediaries (e.g., `medex`), thus, hiding the objects from the user.

Methods are specified in the control file `medex.methods`. Entries in the file associate a UNIX executable with the UID of the project account. In order to be a method of a project object the executable must execute under this UID. Each entry is tagged with the method name that will be used to authorize access to the method (see Figure 3). Method names need not be related to the actual executable (i.e. they can be used to provide a some level of abstraction), but they must be unique. Executables are specified as UNIX pathnames followed by any command line options. The pathnames must be *absolute*, i.e., specified from the root of the directory hierarchy with a leading ‘‘/’’. This is done in order to reduce the possibility of a trojan horse being introduced in the execution path of the `medex` program.

```
METHOD-NAME:uid,pathname [options]
```

Figure 3: `medex.methods` file

Users are granted access to defined methods via authorizations in the `medex.ecl` file. Each line in the file represents an ECL entry that associates a *<uname.gname>* pair with a method. A user with a username of *uname* and group membership in *gname*[3] may invoke the executable defined in the method with the method's UID, i.e., the user may act as the project account to run the program given in the method's specification. For example if methods are defined as in Figure 4 and ECLs as in Figure 5 then user *jane* in group *programmer* can access objects in the project *bigapp* with the methods PRG1 and GUITOOL.

```
PRG1:bigapp,/usr/proj/bigapp/prog1
GUITOOL:bigapp,/local/guis/tool1
```

Figure 4: Example `medex.methods` file

```
jane.programmer:PRG1, GUITOOL
```

Figure 5: Example `medex.ecl` file

The ECL file syntax includes the special character `%` with a similar meaning as in ACLs. The percent sign ‘‘`%`’’ is used to indicate a wild card entry, i.e., any user or group. For example, with the ECL file in Figure 6, **all** users in the group *programmer* are given access to the method GUITOOL, while *jane* is the only user that may access the PRG1 method. Note that *jane* can still access GUITOOL since she is a member of the *programmer* group.

```
jane.programmer:PRG1
%.programmer:GUITOOL
```

Figure 6: Example `medex.ecl` revised file

Medex Implementation

`Medex` is implemented in Perl [wal91] for compatibility with all the required UNIX variants. The special features of Perl for analysis of tainted execution paths are used in order to minimize the possibility of trojan horses or other programmed threats based on executable imposters. Both the ECLs and the method specifications are stored in protected files owned by the superuser account.

Methods are stored internally using a Perl associative array which is keyed on the method name.

[2]This definition of *method* has similar semantics to the methods defined in Smalltalk [gol83].

[3]Group memberships are defined in the file `/etc/group` on most UNIX systems.

ECL entries are stored as an array of regular expressions. When a request is received by an invocation of `medex`, the requester's *<uname.gname>* pair and the method name given on the `medex` command line are compared to the ECL. If a match is found then the method is executed for the requester and a log entry is generated either to the file `medex.log` or by issuing a call to UNIX *syslog* facility. On each invocation `medex` queries the user for her login password in order to verify her identity. This eliminates the possibility of an unknown user accessing a privileged account via an abandoned login. Note in Figure 7 that the user *jane* calls `medex` by specifying its absolute pathname, a good idea when executing any `setuid` program.

```
jane% /usr/local/bin/medex PRG1
Password: xyz123
```

Figure 7: Example `medex` invocation

The `medex` program must operate as the privileged user `root` in order to switch its UID from the requesting user to that of the method's project. Care must be taken to create methods which do not allow the user to escape to a UNIX command shell and gain additional access not defined in the method. Therefore, method declarations for UNIX utilities that allow shell escapes (e.g., *vi*) are highly discouraged.

The reader will note that *user* and *group* names, instead of numeric UIDs and GIDs, were used in the `medex` control file examples above. In fact, the use of numeric UIDs and GIDs is not supported. Experience has shown that these are much less stable than the specific user and group names that are assigned. Since only the names are used in the `medex` files, the problems with UID and GID migration are avoided. Also, references to the user's *group* membership work for **all** groups in which the user is a member, not just his or her current one[4]. This eliminates the need for the user to issue a *newgrp* call in order to switch groups prior to invoking `medex`.

Medex Usage

The procedures for `medex` usage include the establishment of special accounts with disabled logins, creation of groups for application development or support teams, and delegation of access authority to special accounts via the ECL control file. This procedure provides a mechanism for enforcing access via `medex` to the special account.

Further implementation of PECs can be accomplished by changing the file permissions on executables and data to make them available only to the special account, thereby limiting the paths by which the data can be accessed. Management of the ECL file can be delegated to a security administrator for proper separation of duties by changing the file's access permissions to make the file writable by the security administrator. This can also be accomplished with the use of an ACL on the file, specifying access for the administrator's account.

Example 1: Application Project Management

In this section we present a small example to illustrate the use of `medex` when managing a database application. We often have the need to establish processes which collect data in real-time from devices and store the data in a database. Since the application is updating the database, it must operate under the login of a valid database user. Also, the application must be supported by a rotating support staff in order to provide for 24 hour data collection from the device.

To provide for this type of access, we first establish a login account for the application. For this example, we choose the account name *rtapp* to match the application's name, i.e., `rt`. The account *rtapp* is also given write access to the necessary database files on the database server. We disable logins on the *rtapp* account by placing an ''*'' in the password field of the account's line in the `/etc/passwd` file.

Two directory areas are created on the system, one, `/proj/rt/dev` is a development area, and the other, `/proj/rt` is a production area. All files in the production area are given the owner *rtapp*, essentially putting all production access under `medex` control.

Once the special account is enabled, we establish methods to control the application in the `medex.methods` file as in Figure 8. Methods are established to start the application, stop the application, reload a configuration file, and to update the production code from a development area. The RTSTOP method is implemented as shell script that issues a KILL signal to `rt`. The RTUPDATE method is a program that migrates development code from the directory `/proj/rt/dev` to the production directory.

```
RTSTART:rtapp,/proj/rt/init
RTSTOP:rtapp,/proj/rt/kill
RTRECON:rtapp,/proj/rt/config
RTUPDATE:rtapp,/proj/rt/update
```

Figure 8: `medex.methods` file for `rt`

We establish the group *rtappops* in the `/etc/group` file; filling the group with the usernames of the support personnel for the application. Authorizations for members of the group to access the *rtapp* methods are defined in the `medex.ecl` file as in Figure 9. All users in the group *rtappops*

[4] This is primarily an issue in UNIX System V where users may only belong to one group at a time.

are allowed to start and stop the application. Only the user *sally* is allowed to reconfigure the application. The user *tim*, the application department's code librarian, is allowed to move code from development to production. Note that it is not necessary for *tim* to be a member of the group *rtappops*.

```
%.rtappops:RTSTART,RTSTOP
sally.rtappops:RTRECON
tim.%:RTUPDATE
```

Figure 9: Changes to `medex.ecl` file for `rt`

With this configuration, all transactions to the application `rt` are logged with audit trail information indicating the specific user that modified or accessed the application. Code migration from development to production is also controlled and logged, making revision and quality control easier.

Example 2: Running DNS

In this section we present an example of using `medex` to control access to a pre-existing account, namely the `root` account. We operate the DNS (Domain Name System) at our site using the Berkeley Internet Name Domain (BIND) software. The system implements the program *named*(8) as a UNIX daemon which processes queries for the DNS database. The daemon is usually started by *root* from one of the system startup files. Since the daemon is constantly running, it accepts commands via *signal*(3) calls, rather than from the command line. Under normal operation a UNIX process will only accept signals from its owner, in this case *root*. For all system administrators or operators that we wish to have access to `named`, we must give them access to the *root* account. Rather than just hand out the password to *root's* account, we use `medex` to enable *root* access for the limited set of executables necessary to control `named`.

When `named` starts, it reads the current DNS database from a set of files called *zone* files. These files contain the necessary data for the DNS server to reply to queries about hostname to address mapping information. Since the syntax for DNS zone files is somewhat tricky, we automatically generate our files from a host table that is in the same format as the standard UNIX `/etc/hosts` file. The translation is performed by a tool called `h2n`[5] (hosts to named).

Once `named` is running we need our BIND operators to be able to:

- Update the `hosts` file and rebuild the DNS databases on the server with `h2n`.
- Issue a HUP signal to `named` telling it to reload the databases into its active memory.
- Restart `named` by issuing a KILL signal and running the server daemon by running `/etc/named` as *root*.

First we create a new group in the `/etc/group` file by adding a line like:

```
dnsops:*:100:charles,tony,david
```

This sets up the group `dnsops` with group id of 100 and puts our DNS operators (Charles, Tony, and David) in the group. We use the new group to authorize access the DNS methods. We add the lines in Figure 10 to `medex.methods` to support the required commands. The shell script `reload` issues a HUP signal to `named`. The file `restart` is a shell script that sends a KILL signal to `named` and then re-executes `named`. The use of these scripts illustrates using `medex` to control access to an active process, i.e., the `named` daemon.

```
DNSRELOAD:root,/var/named/reload
H2N:root,/var/named/h2n
DNSRESTART:root,/var/named/restart
```

Figure 10: Changes to `medex.methods` file

```
%.dnsops:DNSRELOAD,DNSRESTART,H2N
```

Figure 11: Changes to `medex.ecl` file

With the commands setup we can authorize the DNS operators with one line in `medex.ecl` (see Figure 11). Though this procedure implements all the necessary execution controls, the DNS operators still need to change the `hosts` file. This is done by making the file's group `dnsops` and enabling group write permission on the file. We also put the file under revision control using the revision control system (RCS) so that changes are tagged with the specific username of the DNS operator.

Comparison to Other Tools

Two tools exist with similar functionality to `medex`. One, `sudo`[6], allows controlled access to the superuser account. Commands (and command groups) may be aliased and access to them authorized in a control file called `sudoers`. `Sudo` satisfies many of the requirements of `medex`, such as a protected access mechanism to the privileged account and an audit trail, however it is limited to access of the superuser account. It includes a more complex syntax that is accessed with a parser written in `lex` and `yacc`. The remainder of the program is written in C. These features made it less portable than a utility developed in Perl.

A second tool, `su-someone`[7], allows a specified group of users to switch to a special user's

[5] `h2n` is presented by Albitz and Liu in their book *DNS and BIND* from O'Reilly and Associates. I highly recommend it for anyone considering running a DNS server.

[6] `Sudo`'s current incarnation was developed by Jeff Nieusma and David Hieb at The Root Group, Inc.

account. This stops the password to the special account from being propagated, but it doesn't restrict the actions of the user once they have made the switch. The access list is compiled into the program, i.e., a new executable is created for each application of the tool.

Conclusions

During the implementation of `medex` it became glaringly apparent that most of the work involved developing a model for application project management, rather than actually implementing a tool to provide a controlled access to privileged accounts. Along with the implementation were other procedural issues, such as procedures for setting up special accounts and for establishing adequate separation of duties. Questions raised in this area may lead to future extensions that allow delegation of method and ECL specifications for a particular project class to a project security administrator.

Since some of the ECL specification syntax relies on the use of group assignments in the `/etc/group` file, there is a possible limitation on the number of methods and projects that a user may access. This limitation is imposed by the operating system restricting the number of concurrent groups that a user may belong to at one time. An additional group membership file could be added, but the use of existing DACs in the form of the `/etc/group` was a desirable feature.

Availability

Release information for `medex` is available via anonymous FTP from `ftp.wiltel.com` in the directory `/pub/src/medex`. See the `README` file in this directory.

Author Information

After receiving his M.S. in Computer Science in 1991 from Washington University in St. Louis, Todd Gamble worked as a UNIX system administrator for the university until joining WilTel in July of 1992. He is currently project lead of Network Security for WilTel. His recent project areas include network firewalls, Internet customer access, and Unix security. He can be reached via electronic mail at the address `todd_gamble@wiltel.com`.

References

[bac89] Eugene Mate Bacic, "Process Execution Controls as a Method to Ensure Consistency", Fifth Annual Computer Security Applications Conference, pp. 114—120. 1989.

[bac90] Eugene Mate Bacic, "Process Execution Controls: Revisited", Canadian System Security Centre, Communications Security Establishment, 1990.

[cur90] David A. Curry, "Improving the Security of Your UNIX System", SRI International Publication, ITSTD-721-FR-90-21, April 1990.

[dod85] Department of Defense, *Trusted Computer System Evaluation Criteria*, DOD 5200.28-STD, National Computer Security Center, Fort Meade, Maryland, December 1985.

[fer93] David Ferbrache and Gavin Shearer, *UNIX Installation Security & Integrity*, Prentice Hall, Englewood Cliffs, 1993.

[gol83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, May 1983.

[hew91] *HP-UX Release 8.05 System Manuals*, Hewlett-Packard Company, June 1991.

[wal91] Larry Wall and Randal L. Schwartz, *Programming Perl*, O'Reilly and Associates, Sebastopol, CA, 1991.

[7] `su-someone` was developed by Wietse Venema at the Eindhoven University of Technology.