Fine Grained Object Protection in Unix

Technical Report No.130

Marie Rose Low

March 1992

FINE GRAINED OBJECT PROTECTION IN UNIX

Marie Rose Low
School of Information Sciences, Hatfield Polytechnic, Herts.

March 1992

Abstract

In this document we describe and discuss a solution, called UNIX Access Table Protection (UATP), to the problem of providing a secure environment for persistent object types with fine grained protection on a UNIX system. UATP requires no modifications to the UNIX kernel. User's on UNIX have discretionary access control over their directories and files. By using these controls in a particular way, this paper shows a way in which a hierarchy of protection domains can be set up to enable object methods and type instances to be managed securely and to be protected from unauthorised use. The document also describes how a particular object type, UATP - kernel, which executes in these protection domains (in the same way as all other objects) can be used to provide fine grained protection for all other objects which are installed and running in the manner described here.

1. Introduction

There is no automatic provision for fine grained protection on object oriented software under a UNIX system. This document proposes a means of using the existing UNIX features to provide an environment where an object instance can reside in a secure domain and the methods that operate on that object can only be applied if the user invoking those methods has the appropriate authority to do so.

Protection for an environment and its objects can be provided at two levels. There is 'coarse grained protection', which is the protection imposed on an object at file level, where files include directories. This is the general protection as provided by the system for all persistent data on that system. The second level is 'fine grained protection' and is tailor made for each different object. This protection is imposed on each of the methods which operate on the data, such that not only does a user need access to the files but the user also needs permission to invoke each function individually.

File protection in a UNIX system is achieved by using Read, Write and eXecute permissions at three different levels, the file owner, other users in the owners group and the rest of the world. The UNIX protection mechanism, therefore, is coarse grained.

This document describes a scheme which may be super-imposed on a UNIX system to provide a suitable secure environment for object oriented software.

The document is divided into the following main sections. Section 2 describes the problem which is being investigated. Section 3 defines a notation and uses it to describes how UNIX protection can be used to provide a secure environment. Section 4 proposes a solution to the problems of fine grained protection and describes the use of UATP. Section 5 discusses the decisions taken. Section 6 is the summary and discusses problems still unsolved and other avenues open for further investigation.

1.1 Object Oriented Software

In conventional terms, software is viewed as a set of programs and the data they produce. An alternative to this is to view software as 'objects'. An object is then defined as a set of data and the functions (or methods) used to manipulate that data.

The object methods must run in a secure environment and the data must be protected from tampering. By this we mean that the object is defined so that only the associated functions can operate on the data. Furthermore only authorised persons may invoke these functions. These authorised persons are the owner of the type instance and any other persons to whom the owner has granted permission to perform all or some of these functions. The object not only requires a secure environment so that no person can read or modify the information in an unauthorised manner, but it also requires selective protection for each of the functions.

The terms used in this document are taken to have the meanings described below. Thus, a program is specified in terms of the data it produces and the functions it performs on that data. Two objects are of the same 'object class' if they are characterised by the same abstract user-specification, i.e. they can both be expressed entirely in the same operational terms [EvKe90]. Individual objects which satisfy the same user specification are 'instances' of that class.

An 'object type' is the technique used for a particular implementation of a user specification. An object class has two object types when the same specification has been implemented in two different ways. Different objects arising from one implementation are 'instances' of that type.

An object type and its type instances have to reside in a protected environment so that the object type methods and its type instances are protected from unauthorised access.

For the purposes of this document, a user environment or domain in UNIX is a user area, where the user has discretionary control over the access to all the data in that area. A secure domain (protected environment) is a domain where the user has exercised his access rights to protect his data from manipulation by all other users in the system.

1.2 Secure Systems

Secure systems provide some form of protection for their users and their data. This is done by restricting access to users' data and resources to those users who have been granted appropriate use of the item. Some systems also impose a

security policy, whereby not only must a user have rights to an item before he can access it, but the user must also have the correct classification to own those rights. The UNIX system does not impose such a policy, but it enables each user to set his own protection on his data.

Most security models are based on Lampson's access matrix, in which the rows of the matrix represent the subjects (users) and the the columns represent the information containing objects [Kar88]. The rights that a user holds to an object are found at the intersection of the row and column belonging to the user and object, as shown in figure 1

	Object1	Object2	. . .	ObjectM
User1	null	R	. . .	RW
User2	RW	null	. . .	R
. . .				
UserN	R	RW	. . .	null

Figure 1. Lampson's Access Matrix

This matrix is quite simple as it just represents the coarse grained protection imposed upon objects by an operating system. It can be used to represent the UNIX protection scheme; the matrix would then include the eXecute, as well as the Read and Write, permissions that the users have on each object. It would, however, become more complex if the fine grained protection were to be included. The matrix would be required to include each user's access rights to every method and hence would need to be 3-dimensional to provide a true representation of such fine grained protection. In a real system, the matrix would also be very sparse as most users do not have any access to most objects. Therefore, computer systems do not keep an internal matrix as such, but store the above information in one of two ways, as columns, in Access Control Lists, or as rows as Capabilities. An access control list of users and their permissions is associated with each object; a capability system keeps a list of objects and the access rights to that object for each user. Thus, both systems perform the same function; it is the details of where this information is stored and how it is accessed that differ and may affect the performance of the system.

2. The Problem

Object security is a current topic of research, see [Oli91a]. The problem of providing a secure environment, within the UNIX operating system, for the processing of objects can be viewed on two levels. Firstly, the domain the object resides in must be made secure using the UNIX protection mechanism, and secondly fine grained protection must be available in order to control the use of object methods. Note that the solution detailed within this paper can be contrasted with the approach of Oliver [Oli91a]. Further reference to this point

will be made in section 5.

2.1 Object Protection

When a user creates a file in UNIX, the file is 'owned' by that user. The user also owns the permissions to share the file with other users. Files are addressed by an entry in a directory. A directory is a special file containing entries to other files and the owner of the directory owns the permissions to that directory (though not necessarily to all the files addressed by that directory). A user's 'area' on a system can be seen as the user's ownership of a directory; this is the user's domain. The domain need not be owned by a 'real' user, but may be a 'fictitious' user handled by the system manager. A user domain can be made secure by setting the Read, Write and eXecute permissions appropriately.

The protection provided must be such that if an attacker is able to pose as a legitimate user, then only the impersonated user and those who trust the impersonated user are at risk. The UNIX coarse grained protection must be used in such a way that the following requirements are met:

- no user and no program can modify an object's methods without 'due process'.
- object methods must be installed securely, following their verification.
- releasing a new version of the methods is to be as secure as installing the first version.
- object instances are protected from direct read and write access from any user, unauthorised programs and any other rogue methods; they are only accessible via the authorised methods and these can be executed only by authorised users.

The object methods and the object instances must, therefore, reside in protected environments. These requirements are met as described in section 3.

2.2 Fine grained protection

There must be a mechanism to provide fine grained protection for the object methods. This protection must restrict access to the methods to the creator of a type instance and to any persons authorised by the creator. In this implementation fine grained protection is provided using access control tables rather than capabilities. Where a system offers none of these features, such as UNIX, a centralised means to control access, i.e. a table, seems to offer a more efficient implementation of the solution, rather than a distributed approach using capabilities. It is the fundamental problem of propagating capabilities that makes them of limited usefulness in an application such as this. A fine grained protection mechanism using an access control table is defined in section 4.

3. A UNIX Secure Environment

In this section a language is defined and used to describe the UNIX protection mechanism. The problems of verifying the object methods and installing them are discussed. As always, human intervention has to be used. The trustworthiness of the system manager is the pivot of the security available; if this fails then there is no security. It is also assumed that the onus of verifying new methods falls on those who want to use the methods i.e. if a user wants to use new methods, then it is up to him to check that they are honest and he places himself at risk if he takes someone else's word for their validity.

3.1 Environment

The object methods and all instances of that object type must reside in a protection domain i.e. a user area where access to the files in that area is restricted. The UNIX user domain protection can be used to provide an area where only the owner of that area has access to the files. The *owner* of the methods is the user who owns the rights to set the method's permissions. The *permissions* are the ability a user or user-process has to Read, Write or eXecute a particular file. When a method is executing, it may have been invoked by a user other than its owner, in which case the process is owned by one user but runs with the userid and permissions of the calling user. The *set-uid bit* is a permission which can be set or unset by a file owner for an executable file. If set-uid bit is set, the process runs with the userid and permissions of the owner of the process, otherwise it runs with the userid and permissions of the caller.

Attention must be given to the set-uid bit. This bit can only be set by the owner of the file and no one else. If a user runs a process that is outside its domain, the set-uid bit determines what effective userid that process uses at run time. By default, a process will take on the userid and permissions of the calling user but if the set-uid bit is set then the process runs with the userid and permission of its owner. Thus, if a process A, in a secure environment calls a program, B, outside its domain, then B can take on the permissions of A and then B is in a position to compromise the files in A's secure domain. Clearly, such a situation does not meet the requirements outlined previously in section 2.1.

3.2 Notation

The notation given below defines a language which describes the permissions UNIX processes and their output take as they are invoked by different users.

The following notation is used to describe the current state of a process:

Z: is any user with discretionary R, W and X permissions over any files (program and data) that he owns.

A_Z^W: is a method owned by user Z and running with the userid and permissions of the user (W) who called it.

$^{1}A^{z}_{z}$: is a method A owned by user Z, running with Z's userid and permissions regardless of who called it because it has the set-uid bit set.

$A^{w}_{z}(B^{w}_{x})$: is a method, A, owned by user Z, called by user W, running with W's userid and permissions and which is calling another method B. B is owned by user X and is running with W's userid and permissions because A called it and A is running with these permissions i.e. it is running with the permissions of its caller.

D_{Z}: is a type instance owned by user Z.

A^{w}_{z}—> D_{W}: is a method A owned by Z, called by W and running with W's userid and permissions which operates on the instance D owned by W.

$A^{w}_{z}(^{1}B^{x}_{z}$---> $D_{X})$: is a method owned by user Z, called by user W, running with W's userid and permissions and which is calling another method B. B is owned by user X, runs with X's userid and permissions (it has set-uid bit set) and operates on the instance D owned by X.

3.3 Object Methods and Object Type Instance Protection

To satisfy the requirements in 2.1 the object methods, M, and object type instances, I, must reside in a secure environment. Assume that all the instances, I, reside with the methods, M, in the same protection domain, P. (If the methods, M, have been verified and **only they** have access to the instances then this proves to be satisfactory). Also assume that each object type has a protection domain for its methods and instances which is separate from other protection domains.

Now consider a situation in which there are two different object types, M and N. The methods of N may invoke the methods M. If M and N reside in separate protection domains, then N must perform a cross-domain call to invoke the methods M. If M's domain is compromised then N's domain is at risk as the methods, M, may then have access to N's instances i.e. when N invokes one of the methods of M, this method runs with the userid and permissions of N's domain and so it has access to N's instances; if the methods, M, have been tampered with then there is no guarantee that they will not perform unauthorised operations on N's instances.

In order to avoid the cross-domain call, now assume that the methods for M and N reside in the same protection domain, P. If we still assume that methods reside in the same protection domain as their type instances, then all type instances (for both M and N) also reside in P. If M and N are to create and maintain their type instances in P, then these must necessarily have Read and Write permissions for P. However, should N not have been properly checked and the author be malicious, then, a user, Z, who invokes the methods of M only, is now at risk as both methods have access to his instance. Furthermore, as

M and N both run with P's permissions, the methods themselves can be violated i.e. N could modify M.

A solution can be found by keeping all the methods in one domain and instances of each type in their own separate domains. The methods M and N reside in P, but the type instances for each of the methods reside in separate domains i.e. M's instances, I, reside in protection domain, X and N's instances, J, reside in protection domain,Y. This solves the cross-domain problem because now there is only one domain, P with methods and this alone has to be secure for both M and N to retain their integrity. Furthermore, as M and N do not operate on instances in P they do not run with P's permissions and so even if N has a malicious author these methods cannot corrupt M; neither can M or N operate on each others instances as these reside in separate protection domains.

3.4 Domain Hierarchy

Domains are set up as follows. The author of the object type I, releases the methods, M, to the system manager. This is done using a standard release procedure which may either be automated or manual. The system manager then places the methods, M, in a secure user area, P, where there is read access to all who wish to inspect the code. (If object releases are carried out by using copying functions, these functions will also live in P.) This domain is write protected so that the methods cannot be altered in any way. The methods are then open to inspection by all users that want to use them. This version of the object methods is no longer owned by the author and therefore may not be modified except by repeating the release procedure.

When the system manager is satisfied that the methods have been properly inspected, the object type is ready for use. The methods are then enabled by setting eXecute permission for the world for the executable file.

The executable object methods, M, remain in P and are owned by P, expressed in the notation as,

$$M_P$$

The system manager also allocates a free fictitious user, X, to the new object type ensuring that this is a secure domain for, in the simplest case, all instances of I. The system manager sets up stub routines, S, in X which simply call the methods M in P. The methods M are given execute permission for the world. Thus anyone can invoke the methods, but the only one who can change the permissions of the methods is the owner, P. Furthermore, anyone calling the methods must not be able to take on P's permission. To satisfy this, the set-uid bit for the executable methods remains unset so that they will always run with the calling userid. The stub routines S, are owned by X and also have execute permissions for the world. The stub routines, S, have the set-uid bit set so that they run with the permissions of the domain X and not the calling user i.e.

$$1S^{x}_{x}$$

The methods M, in P, are then called by S, in X, and also run with userid X. The object instances, I, are created under the userid of X and so the owner of I's

permissions is X, and only X has read and write access to them,

$$^{1}S^{X}_{x}(M^{X}_{p} \longrightarrow I_X))$$

In this way, access to all type instances in that domain is restricted to the appropriate object methods, and no user need ever have permission to access the object data directly. When a user, U, wants to invoke one of the methods (using a program or system call), he calls S in X which calls M which then runs with the permissions of X and not U's permissions,

$$U(^{1}S^{X}_{x}(M^{X}_{p} \longrightarrow I_X))$$

It is possible for M to be invoked directly, not using the stubs, by a user that does not follow the correct procedure. This could be overcome by the methods checking that they are only called by S in X. However this is not necessary, as U does not have direct access to the methods themselves or to the domain X so there are no undesirable consequences for legitimate users. Any type instances created in this way however, will not necessarily reside in a protected environment, for U does not necessarily have a secure area and is therefore open to attack i.e.,

$$U(M^{u}_{p} \longrightarrow I_u)$$

When a new version of the object type is ready for release, the methods in P are updated by the system manager: there is no need for any changes to the stubs in X's area. If X is violated the stubs and instances are at risk but there is no risk to the methods which reside in P as there is no way of assuming P's permissions because the methods always run with the caller's userid [Chri91a].

3.5 User Authentication

In the following situation, user A has asked for an instance of an object to be created on his behalf. Only A and authorised sub-users may invoke the methods, M, on A's object. User B, an outsider to the object instance, writes a program, Q, which he persuades user A to run. Q, however, also includes software which invokes M. If B persuades A to run Q, then Q takes on A's userid. This now leaves a clear path for B to pose as A, with A's userid and permissions, and execute the methods on A's instance, I, with no authority to do so i.e. A has not explicitly conferred such permissions upon B,

$$A(Q^{A}_{B}(^{1}S^{X}_{x}(M^{X}_{p} \longrightarrow I_X)))$$

When the user B takes on A's permissions he can call M to operate illegally on A's object, I, but he is not able to violate M in P as they run with the caller's userid and permissions i.e. X's userid. The stub routines in X only call the methods in P, they do not call any process outside these two secure domains, so B cannot take on X's permissions either. B is therefore only able to operate on A's type instances. If A runs unprotected code, then he is obviously at risk, but this

remains A's responsibility. The main requirement is that this does not put any other users at risk unless they share access to those instances also.

One way for the routines to check that the caller is really A is for the methods to have a call back mechanism, but because of the confinement problem [Lamp73] cross-domain calls are undesirable. Another way of tackling the problem is for the access control methods to include an extra level of user authentication so that the methods can check that the user requesting the invocation of the method is the person he claims to be. In this situation, a non-discretionary security policy could be implemented over and above the access control mechanism. If it is acceptable that if a user is compromised, only that user is at risk then no further user checks are required.

4. Unix Access Table Protection (UATP)

This section describes how a particular 'access table' object type (UATP) can be used to provide fine grained protection for all object types contained within the form of secure domains described in section 3.

The access table is, in this paper, customised for use with a specific object type (LIST). There is no reason, however, why this table should not be generalised so that the same UATP methods can be used to provide protection for any other object.

It seems most appropriate at this stage to describe the notion of UATP with reference to a specific example.

4.1 Example

A course organiser wants to keep a list of his students and related information about them. The list is an object type and the functions which can be performed on it are the methods. These functions may include Listing all the student records, Adding a new record, Editing the record for one particular student and Removing a record. The list has confidential information which the organiser does not want unauthorised people to have access to, but he wants to be able to delegate some of the work to others.

The owner of the instance of the object, the organiser, is able to perform all processing functions, but he wants to allow some of his staff access to only some of the functions, for instance Listing all the records and adding a new one, but not Editing or Removing a record.

In order to provide such a service, the methods and type instance need to be protected from unauthorised access. Each method must be invoked only by a user that has permission to do so.

4.2 Object Functions

There are two object types to implement the solution to the problem. The object type, LIST, includes methods that create and delete type instances and those that handle a list of students and the information associated with each of them. The

object type, UATP, is a type controller whose methods create and maintain an access table for the fine grained protection mechanism required by, in this case, the LIST methods.

The methods of both LIST and UATP reside in one protected domain, P. The type instances of UATP and LIST (the access table and the student lists) reside in the secure domains W and X respectively. The methods for both UATP and LIST are called via stub routines, denoted as TS and LS, which also reside in W and X respectively. The LIST methods invoke the UATP methods to impose the fine grained protection. Before these methods are released for use, **both** LIST and UATP are verified (as described in 3.2); without such verification the system cannot vouch for the security of the object instances. The UATP methods have a 'higher level' of verification, as they are checked by all users of object types that use this fine grained protection.

4.3 Type controller Object Structure - UATP

A UATP structure is created for each type instance of LIST. (The table may be a part of the instance itself or may simply be associated with the instance by its name.) The first entry in the table is for the user who requested the creation of the instance; the subsequent entries are for every sub-user that is allowed access to that instance. Each entry has a field for the userid of the user allowed to operate on that instance and a field for each of the methods available. The field for each method indicates whether or not the user in that entry has permission to invoke that function, as shown in figure 2

USER ID	LIST Function	ADD Function	EDIT Function	REMOVE Function
x_id	yes	yes	yes	yes
y_id	yes	yes	no	no

Figure 2. Type Controller Table for LIST

In the example, user x_id owns the LIST instance to which the UATP relates. Consequently, x_id has rights to all the methods. User y_id has been given permission to List and Add entries to x_id's instance.

The type controller functions which create and maintain the access table are the following:

TCREATE	create an access control table
TADD	add an entry to the table
TREMOVE	remove an entry from the table
TEDIT	edit an entry in the table
TREAD	read an entry

The UATP methods are called via the stub routine, TS, and stored in the secure domain W. Only the creator of an instance of that type has access to the type controller functions which modify this table i.e. the calling userid must match

the userid in the first entry in the table. Although this may appear unnecessarily restrictive, it is necessary to prevent access right propagation problems similar to those encountered with capability systems.

The methods TCREATE and TADD are invoked by the LIST CREATE method, which creates a new LIST type instance. TCREATE creates a new access table associated with the newly created LIST instance and TADD places the first entry in that table. The LIST methods have access to the TREAD function so that they can verify user permissions before applying any functions to a type instance.

4.4 Creating and Using an Object Type - LIST

Assume that the LIST methods reside in the domain P and that there is a stub routine, LS, in the domain X, used to call these methods. When a user, U, wants to create a new instance of LIST, he is told the path to the LIST stub, LS, by the system manager. U invokes the LIST CREATE method which runs with the secure domain X's permissions to create a LIST instance, *UL* in X, for U which he can then manipulate using the authorised methods. CREATE, in turn, calls TCREATE, in order to create an access table *UT*, and TADD to place the first entry in *UT* which grants permission to the owner, U, for all the functions. The user's *UT* resides in the protected domain W as the UATP methods execute under the userid and with the permissions of W. Only the protected domains, W and X, have read and write permissions for the entities owned by U. These entities are therefore protected from external tampering. U choses a unique name for each type instance he wishes to create. The userid of the user that requested the new instances to be created, may be included in the identifier for each instance, for convenience. Hence, each instance is easily associated with that user. The above creation process may be represented using the notation, thus,

$$U({}^{1}LS^{X}_{x}(CREATE^{X}_{p}({}^{1}TS^{W}_{w}(TCREATE^{W}_{p}\text{—>}UT_{w}))({}^{1}TS^{W}_{w}(TADD^{W}_{p}\text{—>}UT_{w}))\text{—>}UL_{x}))$$

The LIST methods have execute permission for the world. Protection is enforced by the user invoking the methods via the stub LS. The stubs run with set-uid bit set, but they never change the effective userid. However, as the LIST methods execute with the protected domain's userid, X, they have automatic Read/Write permission for the users' object type instance, *UL*. The LIST permissions may not be altered by anyone outside the protected domain even though a user may be able to view any type instances he 'owns' through a link (in his directory), merely so that he can see which instances he has created.

When a user calls the LIST methods, he must provide the name of the instances on which he wants to operate. Each of the LIST methods invokes the type controller function to check the access control table before operating on the data file. An example of an invocation of the LIST method, REMOVE, is shown below, using the notation

$$U({}^{1}LS^{X}_{x}(REMOVE^{X}_{p}({}^{1}TS^{W}_{w}(TREAD^{W}_{p}\text{—>}UT_{w}))\text{—>}UL_{x}))$$

4.5 UATP Access by Other Users

When a user first creates an instance of a type, he has access rights within UATP, as the "owner", to invoke all the functions available on his instance. This user can modify his own access rights and can enable some other user (a sub-user) to have access to some of the methods which operate on his instance. The "owner" has to identify the sub-user to the UATP methods and tell the controller which of the functions the sub-user may perform on his instance. The user, U calls the UATP TADD method thus,

$$U({}^{1}TS^{w}_{w}(TADD^{w}_{p} \text{---}{>} UT_{w}))$$

TADD places an entry in the controller access table for the named sub-user. The "owner" U, tells the controller the name of the sub-user and the access he is allowed to have to each method. U then gives the sub-user a link to the stub, LS and possibly a directory link to the instance. U may provide the sub-user with the path to LS or the user may pass the link manually using the UNIX 'ln' system command. U may place a link in the sub-user's directory provided that the sub-user allows him write access to his directory, or the sub-user may take a copy of the link if the user allows him read permission. When the sub-user calls the methods, his userid is checked against the access table before he can perform any functions. Thus even if a link is 'stolen' from a user, an attacker will not be able to perform any functions on the instance for which he is not authorised, because his userid will not appear in the access table. Thus propagation of the link providing access to the stubs or instances is not a security problem.

5. Basis of Reasoning

The following decisions were taken to provide the proposed solution.

a The solution proposed and implemented has used access control tables and not capabilities. This seems best suited to a single UNIX system. A protection system which is based on capabilities needs to address the following issues; storing and addressing a user's capabilities, protecting the capabilities, revocation and propagation of capabilities. This last issue is one which is always hard to resolve. However, in a distributed environment capabilities would have a number of beneficial properties.

b The protection mechanism proposed in this paper does not require any changes to the UNIX operating system. Fine grained protection can also be provided by modifying the UNIX kernel and extending the UNIX permissions as proposed in [Oli91a] with further developments in [Oli91b] and [Chri91b].

c All the stubs run with the set-uid bit set (see section 3.3). The data files must not be modified in any way other than by using the the associated methods. By having these files belong to the protected domain, then even the user who requested their creation has no direct access to them. Once this decision has

been made, the methods cannot run with the calling userid as this has no R/W access to these files.

d The object type instances reside in the secure domain and not in the creator's user area (section 3.3). The instance is owned by the secure environment, therefore the user who requested the creation of the instance has no permissions to it. However, a link to the instance is placed in the users directory so that the user is able to list the object instances that he has created. The user has no direct access to the instance, although it is listed in his directory. Only in this way can the instance be protected from direct manipulation by any user.

e It is the responsibility of the owner of the instance to allow sub-users to have access to his instance (section 4.3). The decision to allow another user access to some or all of his instances is taken deliberately by the owner. No sub-user can propagate access rights to another sub-user. This decision was taken to minimise the problems associated with propagation such as the two described below. The owner of an instance may want to allow access to only a sub-set of the methods to be propagated without his deliberate consent; the owner may want to remove a sub-user from the access table and so also any other user that has been given access by that sub-user.

6. Summary

The above implementation is based upon a number of assumptions:

- that the secure domain is in fact protected from users outside the domain and from legitimate users themselves.
- that Read, Write and eXecute permission cannot be violated in UNIX.
- that the userid and password of the protected domain are known only to root.
- that the userid and password are enough to prevent an attacker from posing as a legitimate user.

The requirement to keep the methods themselves protected from attack has been met by keeping the object methods in a separate secure domain from the object type instances and stubs and by making use of the UNIX set-uid bit facility. It is important that no external software is invoked from inside the protected domain unless it has been checked and found to be trustworthy as this would allow the external software access to all the files within that domain.

An advantage of this implementation, is that different object methods can be held in the one secure domain provided that instances of only one type are held in each instance secure domain (instances of the same type may reside in different domains).

If instances of different object types reside in the same domain then the instance for one object is open to direct manipulation by the methods of another object, as all object methods that operate on type instances in that domain have

Read/Write access to all the instances. If users U and V use different methods M and N to operate on type instances I and D respectively and if I and D reside in the same protection domain (X), the type instances I and D are both then owned by X. Any methods running with X's userid and permissions then have access to both I and D (as shown by the notation).

$$U({}^{1}S^{X}_{x}(M^{X}_{p} \longrightarrow I_x)) \quad \text{and } V({}^{1}Q^{X}_{x}(N^{X}_{p} \longrightarrow D_x))$$

U may have verified M and be satisfied that M is trustworthy, but he may never use N and so has not checked them. User, V, however, may not have been very thorough in checking methods N and these may have rogue code. If both I and D are owned by X, then both M and N have access to them. This places U, who has carefully checked what he uses in a vulnerable position. This is avoided by having two domains, X and Y, then

$$U({}^{1}S^{X}_{x}(M^{X}_{p} \longrightarrow I_x)) \quad \text{and } V({}^{1}Q^{Y}_{y}(N^{Y}_{p} \longrightarrow D_y))$$

An object must invoke only software from a domain that is as least as secure as the one in which the object resides. In a secure domain hierarchy, such as P and X (outlined in section 3.4), it is safe for software in X to call methods in P. But in general, when a function in another domain is invoked, that function may take on the permissions of the calling domain which leaves open access to the data in the calling domain. Hence, cross-domain calls are not secure in this implementation. The only way a program can ensure that it is unable to violate another user's space is by running with set-uid bit set i.e. its effective userid is its own, and by never using the system setuid() function. This, of course, will not be a principle adhered to willingly by those wishing to violate another user's data and neither is it one which can be imposed upon the users of a system.

The requirement for fine grained protection is met by UATP. The functions provided by any object, for example, LIST, can be protected by using UATP. Once this feature is available, minor modifications to the access table, or a generalisation of the format described, can satisfy the need for protecting individual methods for any object.

6.1 Further Work

This report has not addressed the issue of security protocols for the installation of methods in the secure domain. The system manager is trusted to install the methods and stubs correctly, therefore it may be necessary to have a means of verifying that it really is the system manager who is performing the function.

The situation where there is a requirement to have objects invoking methods in another domain has not been addressed. It seems that there is no direct way of doing this as there is always the risk that if a domain has been compromised, then calls to any methods within that domain could also compromise the calling domain. The following are some ways of dealing with cross domain calls, each with their disadvantages.

a All object types reside in the same domain and there are no cross domain

calls. There is no real fine grained protection, as all objects have direct access to all instances, and a single security breach compromises everybody.

b If cross domain calls are allowed, then fine grained protection can be applied, but a breach of security in one domain puts all other domains which make such calls at risk also.

c The domains can be arranged in a lattice and calls can be allowed to a higher, but not a lower domain. This works provided there is no call back required from a higher to a lower domain where the lower domain may have corrupt code which is then executed with the higher domain's permissions. This does allow for fine grained protection and protection from corruption.

d The safest way to call one domain from another would be via a 'message' sending mechanism - instead of invoking the method directly a message is sent to a server which in turn invokes the required method. This solves the problem of any user taking on any other user's permissions. It may be achieved using daemons (see [Dew89]), FIFOs or sockets. If daemons are to be used to solve the cross domain problem, then using daemons to call all the methods must also be considered. This could provide a unified solution to restricting access to methods and object instances.

The problem and solutions described in this document are for a single machine and not for a distributed environment, this has not been considered; it may be that a capability solution is more appropriate to a distributed system than an access control table. The distributed environment needs further investigation.

6.2 Conclusion

The fine grained protection mechanism for UNIX described in this document makes use of the existing UNIX features and function and does not require any changes to the UNIX kernel. It does however, place some user domains, e.g. X and P, in a special position of trust. Provided the secure domains are fictitious users, then this level of trust is acceptable. The hierarchy of two secure domains satisfies the need for different object methods to protect themselves from each other, as the methods do not run with the userid and permissions of their own protection domain.

In some circumstances special fictitious users may not be acceptable to system users; they may not trust any user other than root and may wish to impose their own discretionary access controls on their object type instances. In this situation it may be necessary to have a solution which requires changes to the UNIX kernel [Oli91a].

The need for a security policy for user authorisation has been noted but the actual security policy in use has had no bearing on the actual solution to the problem and so the policy itself is not an issue, so long as there is one implemented.

There is a need for code inspection and the ways of doing this have not been discussed. This could be very costly, but there seems to be no better way of ensuring that code is trustworthy. However, in this implementation, inspection

is the responsibility of the actual user group.

The fine grained protection mechanism used is a matter of implementation and UNIX itself does not influence what choice is made. The main issues are in implementing and executing the methods in a secure manner within the confines of a UNIX system.

References

[EvKe90] Evered M., Keedy J.L., A Model for Protection Object Oriented Systems In: *Security and Persistence Bremen 1990.* Bremen W.Germany: Springer-Verlag in collaboration with BCS

[Kar88] Karger, P.A., 1988, *Improving Security and Performance for Capability Systems.* Cambridge: Computer Laboratory. Thesis (Phd)

[Oli91a] Oliver R., *An Object Oriented Protection Mechanism for UNIX.* Staff seminar presentation. March 1991

[Chri91a] Christianson B., Personal Communications, Nov. 1991

[Lamp73] Lampson B.W., *A Note on the Confinement Problem* Communications of the ACM, October 1973 Volume 16 Number 10

[Oli91b] Oliver R., *Post Seminar Thoughts on the Object Oriented Protection Mechanism for UNIX.* Technical Note. March 1991

[Chri91b] Christianson B., *Thoughts on Object Oriented Protection Mechanism for UNIX.* Technical Note. April 1991

[Dew89] Dewan P., Vasilik E., Supporting Objects in a Conventional Operating System In: *Proceedings of Usenix Winter '89 meeting.* pp273-285, February 1989