To: Distribution

From: C. D. Tavares

Date: Monday, December 3, 1979

Subject: Changes to Design of MRDS security

This MTB addresses planned changes to the design of the security features of the Multics Relational Database System (MRDS), and the rationale for the changes. Intimate understanding of the organization of MRDS databases is not necessary for an understanding of the issues raised in this MTB.

Brief Description of the Problem Domain

Our problem is to protect information contained in MRDS databases, and to protect it to a level finer than the segment level. This finest level is not a physical entity (e.g., word or page) but a logical entity called an attribute.

A MRDS database is a directory containing a highly-structured configuration of files and subdirectories. Many of these files and subdirectories hold no user data, but are simply there to allow the MRDS to manage the information contained in the database. Although MRDS security will be using many of these ancillary branches, its main problem is to protect the user information. This is the problem we address.

Each MRDS database (e.g., "phone_book") contains one or more relations (e.g., "customer"), which can be thought of as logical groupings of data elements. Each relation contains two or more attributes (e.g., "name", "address", and "phone_number"), which describe the classes of data to be logically grouped. Any single instance of a relation (e.g., "V. Paoli", "745 Brunswick", "555-7672") is called a tuple. Any single item in an attribute (as well as any single element of a tuple) is called a datum. Our task is to manage the access control for relations, attributes, tuples, and data in a consistent manner.

Every datum belongs to one and only one attribute. However, although attributes can only belong to one "real" relation, there is a facility for constructing secondary relations (via submodels and temporary relations) to the result that for all practical purposes, attributes can appear to belong to more than one relation.

Those familiar with MTB-360 may notice that we have not mentioned "files" as part of the problem domain. This is because

their contribution to the is not as significant as MTB-360 would imply, and we have ignored them for reasons that will be later explained.

Brief Description of the Problem

There are several accessing operations to be controlled within MRDS. Data at the lowest level must be protected discretely with respect to both read and write attempts. Knowledge about the mere existence of attributes should be permittable or deniable. Access to relations also needs to be controlled with respect to certain operations, including the ability to know that they exist; the ability to use them; permission to add tuples to and delete tuples from the relation; permission to alter the structure of the relation (add or delete attributes from its definition), and permission to set access to the attributes making up the relation.

It is also desirable to give the user the option of deliberately limiting his access to an entity to which he would usually have greater access than he temporarily desires (a "capability" system of access). For example, even though a particular user has access to change anything in the phone book database, he might like to access the "customer" relation using a mode in which he is allowed to change the phone number but nothing else, so that he may prevent himself from mistakenly modifying the wrong item.

Corollaries and Deductions

The access allowed to various items at various levels must follow certain necessary relationships. The first is that in order to have a certain access to a lower-level object (where the database is considered the highest level and a datum the lowest) a user must have at least that access to the object(s) above it. For example, access to read an attribute is meaningless if the user has no access to determine the existence of the containing relation; and any access to relations or attributes is meaningless if the user can't access the database

The second is that in order for the previous constraint to make semantic sense, we must be able to order accesses in some manner so as to specify what accesses are subsets of what other accesses, especially since the meanings of modes differ subtly at different levels.

Design Problems with the Mechanism of MTB-360

It is advantageous for the MRDS to deliberately enforce these access relationships between successive levels of data organization; that is, to make sure that it never leaves the database in a state where a user has more access to a lower-level object than to a higher-level one. The next problem concerns what the MRDS does when it detects such an attempt. There are two

choices: propagate the necessary access in the upward direction (for access removals, propagate the revocation downwards); or refuse and require the user to manually and deliberately set the required accesses, starting from the proper end of the tree and working towards the desired level. Obviously the question here is not computability (anything the user can specify manually, the program can figure out how to do itself) but is one of user interface, and a balance between the law of intelligent defaults and the law of least surprise (i.e., does the user know what sweeping access changes may result from a simple, perhaps mistyped, request?) Since it does not affect the operation of access control proper, I will not address it.

What is important, however, is the concept that additions to access must be propagated (either automatically or manually) upwards, and revocations of access must be propagated downwards.

However, the design promulgated in MTB-360 is primarily concerned with minimizing the amount of space consumed by access control additions inside MRDS databases. Because of this, certain design decisions were made that tend to interfere with the basic security model outlined above.

Specifically, an "implied access control" facility is proposed whereby if any entity has no specific access control list of its own, it takes on the access control list of its superior entity. For example, if a relation has no ACL, the ACL of the database controls the access to the relation. It can be immediately seen that this runs counter to normal Multics convention, where if a user is not on the ACL of an object (because the object has no ACL) he has no access to it. This is actually a variation on the old CACL, with all its attendant problems.

For example, consider a database containing four relations. The ACL of the database is set up so all users have read permission. None of the relations or attributes have their own ACL. At some point, the data base administrator wishes to give Smith access to update one of the relations. By setting this access, a separate ACL is created for that relation, and it no longer shares its parent's ACL. (Presumably the parent's ACL is copied and becomes part of the relation's ACL.) Of course, Smith must now obtain write access to the database, or else his update access is worthless. The system automatically adds this ACL to the database. Now, however, Smith has the necessary access to change any relation in that database. Since none of the other relations have ACLs of their own, they "inherit" the database ACL, and Smith is now on that. Because of the implicit ACL references, we have just effectively propagated an increase in access downward by mistake. The alternative is for the software to detect this condition and create a new ACL for all other relations in the file, with the result that none of them now share the parent's ACL. By induction, it can be seen that very few access changes are required to force this process to reoccur to such an extent as to make the entire implicit-ACL strategy counterproductive.

Implementation Problems with the Mechanism of MTB-360

The concept of Access Control Segments (ACSs) being used to control access to the data (although dear to my heart) is not optimal for the MRDS. When compared with the strategy of maintaining internal software ACLs for relations and attributes inside some MRDS segment, it exhibits several inefficiencies. First, file system calls must be made each time an ACL must be read, changed, or compared; or a segment (ACL) must be created or garbage collected. Also, none of the standard reasons for using an ACS are present. There is no symbolic correspondence between any entity and its ACS, since all the ACSs have unique (shriek) names. (One could replace the unique names with names that had symbolic meaning, but given the ACS sharing and thus the need for multiple names, and frequent name changes each time an ACL is changed, we would quickly run into directory size limitations, implementation hair, and all sorts of other undesirable properties.) Even if one could identify a particular ACS as belonging to a particular entity, one could not use the standard ACL-manipulating commands to set ACLs (another standard reason for using ACSs) because along with the sharing and propagation requirements, the ACSs all reside in an inner ring.

It would seem more appropriate to maintain a software ACL for each element, and perform reference count manipulation, sharing, copying, garbage collection, and so on internal to the MRDS. The drawback to this plan is the high complexity of the software necessary to maintain storage-efficient, shared ACLS and to duplicate almost all of the operation of hardcore ACL maintenance code.

Unforseen Conflicts with Future Implementation Plans

Features planned for future versions of MRDS also invalidate some of the original design constraints for attribute-level security. The original design was based on the current property that each attribute belongs to one and only one relation, and that each relation belongs to one and only one file. Left unaddressed is the current notion of relations in data submodels-- views (actually masking templates upon actual relations, with some attributes hidden from the user and others rearranged) whose existence makes it possible to arrive at a particular datum from multiple paths. This means that an attribute no longer has only one superior relation to answer to, but many. Also left unaddressed is a planned future extension to MRDS whereby one of these views can address attributes belonging to more than one actual relation. This means that, short of all the attributes in an entire database, we can no longer partition attributes into distinct groups such that we can affirm that access control changes through any relation or view superior to any attribute in this group will have no affect on attributes outside the group. For example, if we have two real relations, a and b, each with its own (non-intersecting) group of attributes, we can construct a sub-

model view c that uses some attributes from a and some from b. Then when we perform any access control operation on "all the attributes belonging to" a, b, or c, we are almost sure to adversely affect the existing ACL setups of the other relations or views. Given these two properties of multiple-parent and multiple-child, any strategy that requires propagation of access in either direction is doomed to failure. (This conclusion is the reason why files were eliminated from our original discussion of access control, above-- relations and views are sufficient to show the problem, and the organization of the database into separate files should be transparent to all normal users.)

It could be argued convincingly that these views existing in submodels can be considered as analogs of Multics directories, with their references to other relations' attributes considered as file system links to other segments. From this, the access control requirement can be derived by analogy. However, users of relational database management systems DO want to specify attribute level access to attributes differently depending on what submodel or view is used to access it; an added fillip not available (or desirable!) in normal Multics access control. In fact, it is much easier to think of access to attributes by imagining that these attributes belong ONLY to this view, no matter how they are in actuality shared-- especially when otherwise, the administrator is tasked with arranging ACLs on attributes in such a manner that all the access permissions and denials are as he desires them, no matter what combinations of views referring to what combinations of attributes are used to reference them.

For instance, a submodel used to access an employee database may contain a view describing an employee strictly in terms of his administrative information. It may also contain a view describing him in terms of his status as a member of the employee's credit union. Both these views might include his address. It is reasonable to expect that the address would be a writeable attribute when accessing the administrative view, but would be read-only when accessing the credit-union view, REGARDLESS of who is doing the accessing. This helps idiot-proof the database against accidental changes, even by people who otherwise have access. The views themselves would be access-controlled in the normal manner (e.g., access to the credit union submodel and the administrative submodel would be limited by the appropriate file system ACLs).

The database administrator who creates these submodels must realize that the user's maximum access to any attribute is the maximum access granted him to that attribute by ANY view in the submodel; and that other views in the same submodel that restrict that capability limit the user only if he desires to be limited by explicitly using that view (since once a user has access to a submodel, he has access to all views in the submodel).

Proposed New Access Control Mechanism

A new syntax will be defined for submodels to be compiled with the create_mrds_dsm command. Each view will carry an access control field that will state whether the user of this submodel can add tuples to and/or delete tuples from the relation via this view. If both these access modes are missing, then users of this view can only access tuples that already exist. (Null and status access have no meaning in this framework, since it makes no sense to include a view in a submodel if you then either assign null access to it, or deny status access to it.)

In addition, each attribute field in the view description will also carry one of three modes: read, update, or null. These will give the effective access to the attributes of every view right in the definition of the submodel being used.

Certain constraints will be imposed by create_mrds_dsm, in the initial implementation that will be removed as improvements to MRDS are implemented. For example, without "null" (nonexistent) data values (not yet implemented in MRDS) it makes no sense to allow add permission to a submodel view that does not contain EVERY attribute of the relation(s) after which it is modeled (since creating a tuple by storing some but not all of its members cannot be done without null values). Similarly, cmdsm will not allow add permission to be specified for any view for which update permission is not also specified on all of its attributes. As new features like null values become implemented, these restrictions will be relaxed in cmdsm.

The database itself must reside entirely in a protected environment. Gates will be created to allow varying degrees of access into the protected environment by various people (database administrators and arbitrary users). All submodels for a database will also be contained in the protected environment. They will be constrained to reside within the database directory itself, not because of security requirements, but as an aid to database administrators who wish to keep track of existing submodels for their databases and perhaps be able to make wholesable changes to them if the main model is restructured in a fashion that would affect the submodels. Depending on the ultimate implementation of submodels as objects, this may restrict creation of submodels to database administrators; a limitation not now imposed.

The biggest change in the operation of current databases is that all secure references MUST be made through a submodel. The database model will contain no security information, nor will the data. Many customers currently use the model to directly reference the database. Fortunately, it is relatively simple to create a submodel that is isomorphic to the model, and to force all references to the data to be through that submodel. Since references through submodels are made exactly the same way as references through models, it can be made practically invisible to the user that the substitution has taken place.

References through the model will be allowed by a special

gate usable only by the database administrator and those he au-..orizes. Any user allowed to use this gate may completely by-pass attribute-level security. Authorization to use this gate is also necessary to restructure a relation; that is, to expand it (make it include more attributes) or contract it (make it refer to fewer attributes), since the actual data model must be changed to accomplish this. The same gate must be used to set and delete access to submodels in the database, as well as to create new submodels and delete old ones.

Several procedures (e.g., dsl_, dsmd_, and so on) will have to be changed in such a manner that they can determine whether a database is in a protected environment or not, and make the appropriate call; directly to the proper procedure (in the non-secure case) or to a gate which invokes the proper procedure in the protected environment.

Finally, it should be noticed that nothing in this access control mechanism precludes us later adding real ACLs to attributes (only; not relations, files, or anything else that would require propagation). The ACLs of these attributes then would be the first line of defense-- the final definitive word as to who can or cannot access any attribute. The access on attributes and views defined in the submodel could then limit, but not extend, the access defined by the attribute's ACL. This would be a convenience rather than a necessity; it would allow a DBA to feel secure that he knows who can and cannot access anything, and that he has not erroneously given someone access to a submodel that contains a view possessing more "power" than he realized (or rather, if he HAS done so, that no damage will actually be done).

And Now the Bad News

The suspicious reader has doubtless noticed by now that we have been taking almost unnatural care to use the term "protected environment" instead of "inner ring". There is indeed a reason for this circumlocution.

If the protected environment chosen is an inner ring, all databases sharing that inner ring share the same protected environment. In the general case, they are not protected from each other. For instance, suppose a person is authorized to use a database in a restricted manner-- to read a couple of attributes and update one, let's say, but nothing more. If he wishes to circumvent security, he may decide to create a dummy database of his own, ask the system to secure it, and thus become (to coin a phrase) a "banana republic database administrator". Now, since he has rw physical access (in the inner ring) to the database he wishes to subvert, and since he has access to perform database administrator operations on his own database, all he has to do is find some way to cause his database to somehow "indirect" various requests into the other database. There are two entities he can try to exploit: the database itself, and the associated programming. The use of an inner ring to enforce database security is

thus dependent on the database being a known quantity, operated upon by known programming. Unfortunately, this turns out not to be the case.

For starters, it is definitely possible to make clever alterations to the structure of the database itself that would cause indirection as described above. For example, a MRDS database contains canned attach descriptions naming some of its own subfiles. A clever user could patch these descriptions so that they reference any other file sharing the protected environment with the database. The patching would have to take place before the database is transferred into the protected environment; meaning that the command interface that transports a given MRDS database into this protected environment would have to perform extensive "gullibility checks" upon the format of the database, not only for this case, but possibly for others as well. It becomes a problem then of the validator being cleverer than the perpetrator; hardly a sound basis for a security implementation.

Worse yet, we cannot even rely on the validity of the programs operating in the protected environment. A common feature of database management systems (and one being added to MRDS presently as a customer requirement) is the ability for the database administrator to specify "editing" or "validation" procedures that can change or impose extra constraints on data elements. For example, a field which holds an automobile registration may be simply "char (7)" to MRDS; but to the customer, it must contain three letters, a dash, and three numbers or else it is invalid. As another example, perhaps the user wishes to log every change ever made by anybody to a certain attribute. All such programming must be executed within the protected environment.

Those who have ever been exposed to the classic "rings versus domains" argument will recognize this as a prime example. For those who have not, we regret that a full explanation of domains in this MTB is impractical. Very roughly explained, domains are like rings but nonconcentric; and their properties of intersection (or lack thereof) can be controlled by the creators of the domains. Thus, each secure database would operate in its own protected domain, separate from but equal to the domains in which other databases were kept. Any Trojan Horse programming thrown into a protected domain by a devious database administrator can then only subvert information within that domain--namely, the administrator's own database. Of course, database administrators who trust each other could arrange to have their private domains accessible from the other's. This would be the ideal technical solution to the problem at hand. The drawback is that, for the near future, no domain capability seems planned for Multics.

Other halfway measures are possible. Access on secure databases could be limited to one project only (like user-owned gates are today). This begs the question of information sharing. It also makes data dictionaries irrelevant: who cares what data items are available in other databases at your site when all the

data elements you will ever be allowed to know about are limited to the ones right there in front of you?

Alternatively, the System Administrator could issue database administratorships only to known "good guys". This would be enforceable by requiring creation of a secure database and ACL-setting thereon to be an exclusive power of the System Administrator. Also, the System Administrator would have to personally audit and install all user editing procedures. This operation is similar to the current operation of Multics system library maintenance. However, there is a big difference between maintaining system libraries (of which there is only one set per site) and user databases, which may run into the hundreds. The System Administrator's job in the latter case would be truly oppressive. This option actually does nothing but create a large corps of System Administrators who are glorified Database Administrators, thus effectively removing most direct administrative control over the database from the user who actually owns it (and may have a proprietary interest in controlling it).

Either of the constrained solutions severely limit the options open to user database administrators. If either is implemented, the implementation should be considered as at most a temporary stopgap solution.