LANGUAGE-BASED PROTECTION MECHANISMS

NANCY ELAND

TR78-333

Department of Computer Science
Cornell University
Ithaca, NY 14853

This research was supported in part by an American Association of University Women-IBM Corporation Fellowship.

Language-based Protection Mechanisms

The control of access to information is one of many critical problems in a shared data base system. A wide variety of models have been proposed to guarantee the enforcement of security constraints in a shared environment. Consider the following four steps of user-data base communication:

(1) a user submits a request in the form of a program

(2) the program is translated by the language compiler

(3) the object deck is loaded by an operating system utility routine

(4) the program is executed (I/O with the data base system is performed through operating system routines)

Security controls could be interposed in any of the four steps above. The majority of existing systems have concentrated security controls either in the first or last step.

In the first case, a physical security check or password scheme may be used to permit or deny users the ability to submit program requests (without regard to what that request might be). This solution is absolute, and while it would definitely prevent unauthorized access to the data base, it is not sufficiently flexible. The system must have the ability to grant users access to the shared information, yet restrict their access to selected portions of the data base.

To interpose security controls at execution time, most existing systems choose to modify the I/O routines of the operating system to include security checks, so that each user access is verified at run-time by the operating system. These security checks greatly increase the overhead for I/O, which is already one of the highest expenses for data base systems.

We wish to consider systems which introduce security controls <u>into the user's programming language</u>. These language-based mechanisms may permit the security system to recognize and flag security violations during program translation, thus reducing the number of required execution-time checks. Obviously, some constraints, such as those dependent on the value of the data, cannot be tested until run-time, so some type of run-time file monitoring must also be provided by the host system.

In developing language-based access control mechanisms, four criteria will be considered important:

(1) The system must be flexible enough to provide for individualized user-views of the data base and read-only access to selected items.

(2) User containment must be enforced (the user must not be able to access restricted areas of memory).

(3) The system must be robust. Previously compiled user programs should remain unaffected by changes in the physical structure or organization of the data base.

(4) When feasible, security constraints should be checked at

compile-time rather than run-time.

One software system which provides an access control mechanism which satisfies these criteria is ASAP, a file maintenance and information retrieval system in which security controls were included as primary design objectives. (CON72a, CON72b, ASA71)

In ASAP a directory associated with each file is used to define user-views. This directory contains user identification information, a description of the file subset permitted to each user, and a list of actions permitted to each user. Two methods are available for specifying a particular file subset. First, the directory contains a list of Boolean expressions describing the contents of the records which are accessible to a user. These expressions are "anded" to every request submitted by that user. Hence the user program itself verifies at run-time that these value-dependent constraints are satisfied before giving a requested record to the user. Second, the directory specifies a security class for each data field, and designates for each user which security classes he is permitted to use. These classes are independent of the value of the fields, and thus a compile-time check is used to enforce these security class restrictions. In addition, the directory lists the type of actions a user may request (for example, read-only access, update records, update user directory). Associated with each type of action is a list of verbs which can be used if that action is permitted. A compile-time check is used to verify that no unauthorized verbs have been used in a program.

Rather than having the security monitoring task in the operating system's partition during execution, ASAP provides a file monitor in the user's partition. This file monitor routine resides side-by-side with the user routine in the user's memory partition and unobtrusively, yet unfailingly, enforces the security constraints at program execution time. Only the file monitor communicates I/O requests to the operating system. The information flow is as pictured in Figure 1. The user has no direct communication with the operating system I/O routines.

User containment is guaranteed because all data fields can only be referenced by name in the ASAP language. Every field in the user environment has a unique, unambiguous name bound to it, as specified in the internal directory, and no other means exists for accessing memory. Thus, unless the user goes outside the ASAP system, he is confined to his own memory space.

In a sense, ASAP meets the criterion of robustness by default. Since it is a compile-and-go system, every user program is recompiled each time it is run, and thus no object decks exist which might become outdated by physical reorganization of the data base.

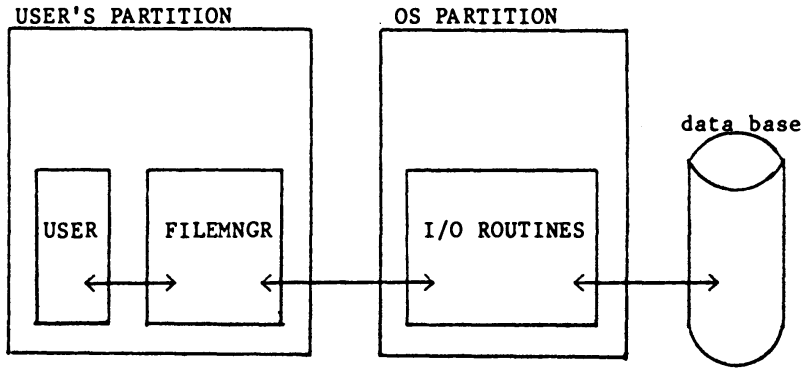

Figure 1

In summary, ASAP provides user-views by enforcing value-dependent constraints at the record level, value-independent constraints at the field level, and read-only access at the record level. A file monitor routine residing in the user's partition provides run-time enforcement of certain constraints. User containment and robustness are also guaranteed.

Thus ASAP provides a simple, yet fairly flexible security system. Unfortunately, the host system has a narrow range of application, due in part to its limited computational ability, and its restriction to sequential or ISAM file organizations. Its greatest flaw in terms of modern data base systems is the inability to express and manipulate interrelated entities in the language. (An ASAP request must deal with only one type of entity.)

Until data security enforcement can be guaranteed in a general environment, it is of limited value. Since a large number of data processing programs are written in general-purpose languages such as COBOL, FORTRAN and PL/I, it is important to consider security control using these languages. Using the ASAP model as a basis for our discussion, we wish to determine if it is feasible to provide comparable security control in a general purpose programming language.

In our discussion, we shall make several assumptions about the environment in which the security system will run. First, we assume the computer system is physically secure, and thus we will not be concerned with problems such as wiretapping or physical loss of secured files. Second, we assume that an adequate method exists for authenticating user identities. Third, we assume that users cannot by-pass the protection mechanisms using some "back-door" entrance to gain access to the data base. Finally, the software utilities used must also be secure. Since many of the security constraints are checked at compile-time, there must be no way to modify a user object program between the time it is compiled and the time it is executed. We are well aware that none of these assumptions are entirely realistic, but they at least serve to partition the various threats and allow us to concentrate on what might be called "frontal assaults".

We will now briefly describe two models which examine the possibility of enforcing access control constraints using only existing PL/I language facilities. Each is based on a different technique of communication between subprograms. The first method exploits the _static scope rules_ of a block-structured language (using nonlocal environments as a means of communication). The second method uses the _parameter-passing mechanism_ of procedure calls.

INTERNAL BLOCK MODEL

Let us review briefly those features of PL/I relevant to block structure and static scope rules. We need to define three data control operations concerned with identifier association:

> "_Naming_ is the operation of creating an association between an identifier and a program or data object.
>
> _Activating_ is the operation of making active an existing association between an identifier and a program or data object and thus making the association available for use in referencing.
>
> _Referencing_ is the operation of retrieving the data or program

object associated with a given identifier using the unique currently active association for the identifier." (PRA75)

The _referencing environment_ of a program at a particular point in time is that set of identifier associations which is presently active. Whenever a reference to an identifier occurs, it is the referencing environment which determines the appropriate association. References to identifiers are classified as either local or nonlocal. They are _local_ if they use an association active only within the block or subprogram currently being executed, for example, a reference to an identifier declared in the most recently entered block. A _nonlocal_ reference is any reference which is not local. _Scope rules_ determine referencing environments and can be classified as dynamic or static. A _dynamic scope rule_ defines scope in terms of program execution. (For example, the referencing environment may be determined by the most recent naming operation for each identifier, as in SNOBOL). A _static scope rule_ defines scope in terms of the structure of the program when written, at translation time. For example, in a block-structured language such as Algol or PL/I, each program or subprogram is organized as a set of nested blocks, and the referencing environment is determined by the pattern of declarations in the enclosing blocks in the original program.

A subprogram (or block) has a local environment, consisting of those identifier associations that are activated on entry to the subprogram (the identifiers declared in the head of the subprogram). In addition, a subprogram may have a nonlocal referencing environment, consisting of associations shared with other subprograms. In PL/I nonlocal references may include references to identifiers declared in intermediate-level blocks between the innermost and outermost block.

The rule for referencing nonlocal identifiers is as follows: the variable referenced by a nonlocal identifier in a block or subprogam definition is that variable declared within the innermost physically containing block or subprogram in the program as written. Thus at the time the program is written, we may create program blocks which take advantage of the static scope rules to enforce access control constraints.

Consider the program outline in Figure 2a and its associated block structure in Figure 2b. USER1's private variables can be declared locally in his user block. In order to retain these local variables from one entrance into the block to the next, the STATIC attribute can be included in the declaration. In addition, shared variables of the data base may be declared in the outer block, since USER1 has access to variables declared in that block.

If the data base administrator wishes to deny a user access to selected data items, those variable names must be hidden from the user block. One solution is for the DBA to create a new, protective block around each user block, and include in that block masking declarations of the variables denied to the user. In determining which declaration applies to a variable reference, the scope rules dictate that successive outer blocks are searched until a declaration is found. Thus from within USER1's block, the masking declaration in the protective block will be found first, rather than the data base declaration in the outer block. The binding made to the masking variable will prevent the user from getting access to the restricted field, but the error will not be detected until the program attempts to access an uninitialized

```
HOST:  PROC OPTIONS (MAIN);
     <declarations of data base structures>
     DO WHILE(...);
          CALL RECORDSELECT;
          BEGIN;
               <USER1 block>
               END;
          CALL RECORDUPDATE;
          END;

     FILEMNGR: PROC;
          RECORDSELECT: ENTRY;
          RECORDUPDATE: ENTRY;
     END FILEMNGR;

END HOST;
```

Figure 2a

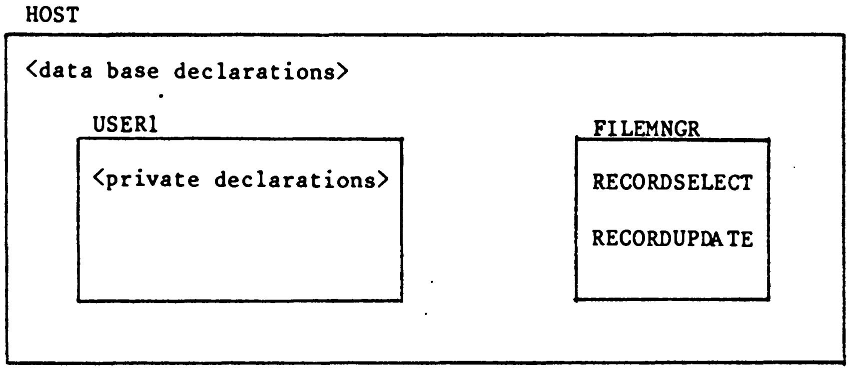

Figure 2b

variable at execution time. Of course, the compiler could be modified to flag this binding to a masking variable as an error. With the new protective block, the program structure would be as indicated in Figure 3.

Using this technique alone, we are still not able to enforce read-only access. For example, a user may be permitted to read a variable, but not modify it. One way to enforce this would be to declare masking variables for the read-only variables in the protective block to hide the data base names, provide new names that the user block can use, and then assign the value of the read-only data base variables to the new variables prior to entering the protective block. A new read-only protective block could be included surrounding the previous one. The program and block structure would be as pictured in Figure 4. For example, if USER1 could not modify the salary field, the DBA could add the declaration for SALARY in the read-only protective block, add the declaration for EMPSALARY in the protective block, and then include the

```
HOST:  PROC OPTIONS (MAIN);
     <declarations of data base structures>
     DO WHILE(...);
          CALL RECORDSELECT;
          PROT1: BEGIN;
               <masking declarations>
               U1: BEGIN;
                    <USER1 block>
                    END U1;
               END PROT1;
          CALL RECORDUPDATE;
          END;

     FILEMNGR: PROC;
          RECORDSELECT: ENTRY;
          RECORDUPDATE: ENTRY;
     END FILEMNGR;

END HOST;
```

Figure 3a

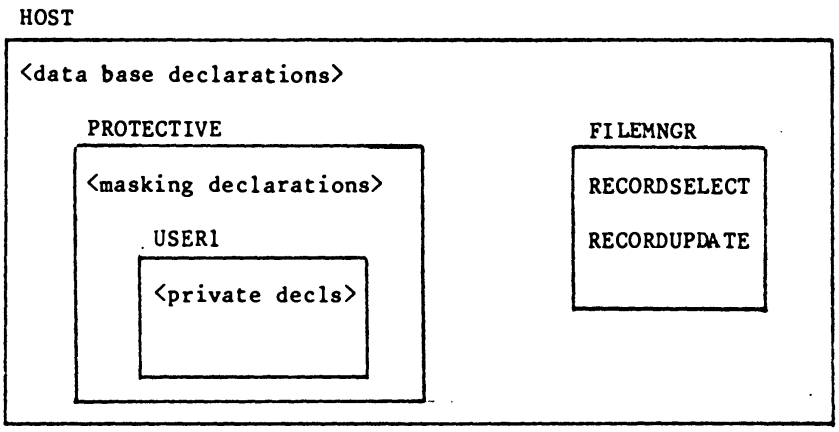

Figure 3b

assignment SALARY = EMPSALARY in the read-only block. (Assume that EMPSALARY is the name declared in the outer block for the salary field.) The user could then use SALARY anywhere he wished to use the variable for read access, but could not modify the data base field EMPSALARY, since it is masked from the user block. The compiler could be modified to flag as an error any attempt to change the value of SALARY.

In this system it is relatively simple to enforce value-dependent constraints at the record level to limit users to subsets of the file. A conditional statement could be added in the RECORDSELECT routine, and if the condition was not satisfied for a particular record, the routine would simply not release that record to the user.

In summary, the internal block model considers user code to be a program block which is surrounded by protective blocks used to hide restricted variables, and enforce read-only constraints. The file

```
HOST:  PROC OPTIONS (MAIN);
     <declarations of data base structures>
     DO WHILE(...);
          CALL RECORDSELECT;
          READ1: BEGIN;
               <read-only declarations>
               <read-only assignments>
               PROT1: BEGIN;
                    <masking declarations>
                    U1: BEGIN;
                         <USER1 block>
                         END U1;
                    END PROT1;
               END READ1;
          CALL RECORDUPDATE;
          END;

     FILEMNGR: PROC;
          RECORDSELECT: ENTRY;
          RECORDUPDATE: ENTRY;
     END FILEMNGR;
                .
END HOST;
```

Figure 4a

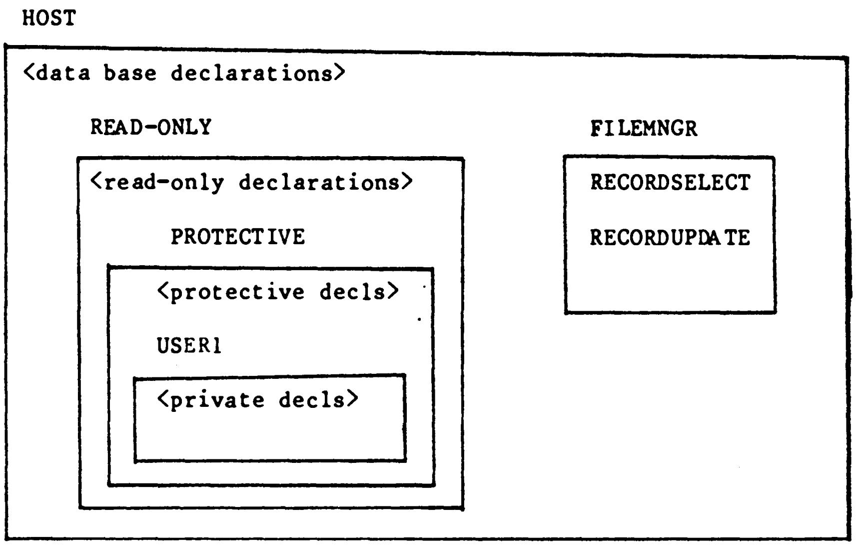

Figure 4b

manager is used to read (or write) records selected by the user, and to enforce value-dependent constraints.

However, several disadvantages are apparent in this method. One is the lack of independence from program modifications or file reorganization, due to the heavy reliance upon the block structure and referencing environments. The nesting of the blocks dictates that the entire program must be recompiled if a user subprogram is changed, or if the data base definitions are modified. The individual user routine cannot be compiled separately and stored for later inclusion in a program.

One of the most severe problems with this model is that it is not able to enforce user containment. With the available language facilities (such as ADDR, pointer variables, label variables, and array subscripting), a user may gain access to the address of a variable, then read restricted areas of memory, completely by-passing the name-binding facilities (and checks) of this model.

EXTERNAL BLOCK MODEL

The second method to enforce access control constraints is through the use of procedure calls and parameter-passing as a means of communication. In order to clearly delineate this method from the previous one (in which communication was through nonlocal environments), let us assume that all communication between program blocks is through parameters.

Let us review the language features associated with parameter transmission. First we must distinguish between actual parameters (or arguments) and formal parameters. Formal parameters are the identifiers used in a subprogram to name the items transmitted into the subprogram. Actual parameters are the expressions used in the calling program to specify the data or program items to be transmitted to the subprogram. By using parameters, a subprogram can access nonlocal data through strictly local identifiers. (Formal parameters are considered to be local identifiers in a subprogram.) A simple positional correspondence with the actual parameters specified at the point of call associates the formal parameters with nonlocal values on subprogram entry.

PL/I uses call-by-reference parameter transmission. In call-by-reference a pointer is transmitted (usually a pointer to a data location containing the value). Each reference to the formal parameter is treated as a reference to the location of the actual parameter. Thus call-by-reference allows data to be transmitted both into and back from subprograms.

Consider the program outline in Figure 5a, and the associated block structure in Figure 5b. The user can have local declarations for private data, and can access shared data via the parameters. In order to give each user a different view of the data base, the file manager can construct an individualized record before passing it to the user. In addition, record level value-dependent constraints can be enforced by the file manager routine. To enforce read-only access, the RECORDUPDATE routine of the file manager must ignore changes to fields which are supposed to be read-only.

Clearly, the parameterized user-views enforce value-independent constraints. If a user attempts to use a data name to which he does not have legal access, two possibilities exist. First, he may have declared that data item as a formal parameter. Since the file manager routine dictates what information is returned to the user through the parameters, the user has simply misnamed the data returned by the file manager, and will be misinterpreting the information (only the user himself will be harmed by this). Second, the user may be referencing a previously undeclared variable, attempting to use a nonlocal reference. Again, no harm is done, because PL/I simply creates a local variable declaration for that reference, and no protected information is obtained.

In summary, the parameter-passing mechanism is used to provide

```
USER1: PROC OPTIONS (MAIN);
     <user view declarations>
     CALL RECORDSELECT(RECORD);
          <user code>
     CALL RECORDUPDATE(RECORD);
     END USER1;

FILEMNGR: PROC;
     <data base declarations>
     RECORDSELECT: ENTRY(RECORD);
          <build user view>
          RETURN;
     RECORDUPDATE: ENTRY (RECORD);
          <update data base record>
          RETURN;
     END FILEMNGR;
```

Figure 5a

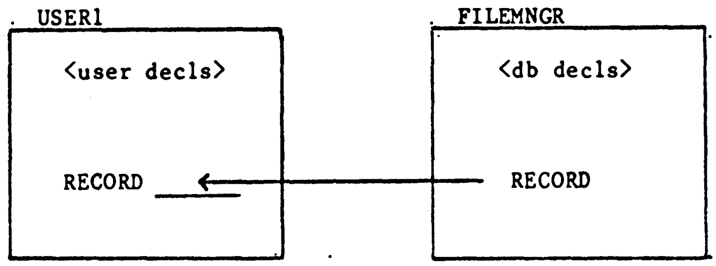

Figure 5b

individualized user-views of the data base. All communication of shared data is through the parameters, and the file manager subprogram can thus monitor and enforce the access control constraints.

The parameter-passing model is relatively independent of physical restructuring of the file. Since the file manager subprogram constructs the user's view at run-time, changes to the data base definitions would only require a modification of the view-building portion of the file manager routine. The user program can be separately compiled, and linked with the file manager just prior to execution.

While the delay of matching formal and actual parameters until run-time increases independence from data base changes, it eliminates the ability to perform compile-time checks for security constraints. This model will therefore have comparatively high run-time overhead for security enforcement.

Although the user and file manager routines are both external procedures and supposedly isolated from each other, user containment still can not be guaranteed. The same language facilities (ADDR, pointer variables, etc) which caused problems in the last model can be used to subvert the security control in this model.

CONCLUSION

Both models offer functionally adequate (although somewhat clumsy) means of establishing the necessary views. In comparing the two models, there appears to be a trade-off between compile-time security checks and system robustness. The internal block model provides for an early binding of names through the static scope rules and hence the ability to enforce certain constraints at compile-time. It fails, however, to be sufficiently robust, since data base reorganization requires recompilation of programs. On the other hand, the external block model relies on the run-time construction of user-views, and is robust, but has a high run-time overhead for security enforcement since no compile-time enforcement is possible. Apparently, robustness can not be obtained without some loss in efficiency due to reduced compile-time checking.

However, both models <u>fail to guarantee user containment</u>. This is due to the powerful language facilities available in PL/I, rather than to a particular fault in either of the models. The next step is to examine this question of program containment in detail, to enumerate and classify the threats, and to analyze possible approaches to overcoming them.

REFERENCES

[ASA71] ASAP System Reference Manual, Compuvisor, Inc., Ithaca, N.Y., 1971

[CON72a] Conway, R.W., Maxwell, W.L. and Morgan, H.L., "On the Implementation of Security Measures in Information Systems", CACM 15(4), April 1972, 211-220

[CON72b] Conway, R.W., Maxwell, W.L. and Morgan, H.L., "Selective Security Capabilites in ASAP", Proceedings of AFIPS SJCC, 1972, 1181-1185

[IBM76] IBM Corporation, OS PL/I Checkout and Optimizing Compilers: Language Reference Manual, Form GC33-0009-4, 1976

[PRA75] Pratt, Terrence W., Programming Languages: Design and Implementation, Prentice-Hall,Inc., Englewood Cliffs, N.J., 1975

[SUM74] Summers, R.C., Coleman, C.D. and Fernandez, E.B., "A Programming Language Approach to Secure Data Base Access", Rept. G320-2662, IBM Los Angeles Scientific Center, May 1974