System integrity is a basic requirement for operating system security. Presented are types of system integrity problems and their general solutions. Techniques used in OS/VS2 Release 2 to solve these problems are highlighted.

Operating system integrity in OS/VS2

by W. S. McPhee

System integrity is a major step in the direction of increased operating system security capability. This paper provides an explanation of the system integrity problem and how it relates to security. The general classes of integrity problems/solutions are discussed, and primary techniques used in VS2 Release 2 to correct or avoid integrity "exposures" are presented. User procedural requirements necessary to maintain system integrity, and the impact of system integrity support on the overall system are also addressed.

the need for security

During the past several years, concern over operating system security capability has grown significantly, motivated by the following basic facts:

- Data and the ability to process it correctly are valuable commodities. Both companies and individuals have a clear financial interest in being able to protect themselves from unauthorized disclosure or alteration of critical or sensitive data.
- Individual privacy is a socially valuable commodity. Much sensitive information concerning individuals is required to be kept private. In some cases this is now required by law; in other cases by growing social concern over the rights of the individual in our society.

To relate this growing concern to specifics, consider the following few examples:

- There has been increasing pressure over the past several years to bring the privacy issue as it relates to computers and other electronic equipment under the control of the law either through new laws, or through the courts in attempts to make existing privacy laws apply in this area. For example, in a recent suit in the state of California, an attempt was made to obtain a ruling to the effect that placing data in a particular computing system was equivalent to public disclosure of that data, due to the inadequacy of security safeguards in that system.

- The Department of Defense has issued rulings that generally prohibit defense contractors from simultaneous multiprogramming of classified and unclassified (or different levels of classified) jobs. This is of particular concern since it often results in inefficient usage of large computing systems.

- Many installations, in order to cost justify the large systems that are needed for peak-load environments, sell time to outside users in nonpeak periods. In addition, of course, there are installations whose business it is to sell time to outside users. Both of these cases give rise to concern over the capability of the system to protect accounting data and system accounting mechanisms from unauthorized alterations.

In general, the need for reliable operating system security capability is now well established, although there is still considerable disagreement over what type of security is appropriate for various types of operations, and so forth. However, there is another attribute of an operating system, called system integrity, that is a basic prerequisite for any security system.

The need for system integrity stems from the fact that the consideration of security introduces a new concept into operating system design. Security is by definition not only concerned with accidental exposure or damage, but also with deliberate, unauthorized attempts to access protected resources. However, the concept of a "malicious user," or adversary, is a concept that has historically not existed in a general-purpose operating system. Previous systems such as OS/MVT were not designed to prevent deliberate user (user program in the sense of a normal problem program, with no special authorization) tampering with the operating system. There was what could be called an "accidental error" philosophy which essentially said that the operating system would attempt to protect itself and other users on the system from common "accidental user errors," but there was no explicit attempt to protect against the user deliberately trying to interfere with the operation of the system. Consequently, in such systems a variety of ways did exist in which the functioning of the operating system itself could easily be tampered with.

The system integrity problem, then, is the fact that security controls, no matter how sophisticated, are not reliable if the operating system that administers those controls is not itself protected from user tampering. Putting security controls in a system not so protected simply protects against the "honest" user, and is somewhat akin to putting locks only on the entrance doors to a building on the assumption that no one would enter through an exit door. Thus any enhanced system security capability must

begin, at the most basic level, with solving the system integrity problem of eliminating the "back doors" to the operating system and its resources.

This leads to a more formal definition of *system integrity*—the ability of the system to protect itself against unauthorized user access, to the extent that security controls cannot be compromised. Specifically, for OS/VS2 Release 2, this means that there must be no way for any unauthorized program,[1] using any defined or undefined system interface, to:

- Bypass store or fetch protection.[2]
- Bypass password checking.
- Obtain control in an authorized state.[1]

Thus the system integrity support in VS2 Release 2 is not a goal in itself. Its objective is to create a system base, a foundation on which existing security features (for example, password protection) are reliable and effective, and a foundation on which user or future IBM security capability can be built.

reliability, availability, serviceability

Although the reason behind VS2 Release 2 system integrity support is security, this support also has the side effect of increasing system reliability and availability. While the "accidental error" philosophy mentioned previously has existed for some time, it has been an informal and discretionary guideline, and performance/design point trade-offs were often made with respect to the types of user errors that were protected against. In fact, many of the integrity problems found in previous systems could be caused by accidental, as well as deliberate, user errors; perhaps more significantly, many such problems turned out to be exactly the types of problems that are likely to result in intermittent "bugs" in the system.

OS/VS2 Release 2 integrity support by definition had to significantly improve system/user isolation capability, and in so doing, restrict to that user himself the scope of the damage that he could cause (accidentally or deliberately). In fact, the ability to cause an uncontrolled system failure at will was specifically identified as an integrity exposure in VS2 Release 2, thus assuring an increase in system reliability and availability. (Note the difference between uncontrolled system failure or crash as opposed to such things as system wait state or reduced system throughput caused by obtaining excess amounts of global resources such as System Queue Area or Direct Access Space. This differentiation was made because an uncontrolled system failure may directly impact security in that "secure" data may be destroyed or disclosed. The deliberate use of excess amounts of global resources, however, while affecting system availability, does not affect data security.)

System integrity problems and their solutions

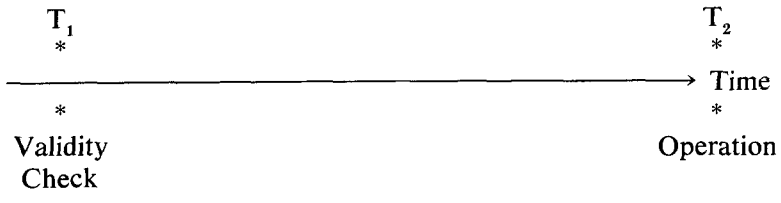
Over the past several years, various IBM and customer study efforts examining operating system integrity have aided the detection of integrity exposures in VS2 and previous systems. Furthermore, knowledge in the area of system integrity has progressed to a point where key validity-checking criteria essential to system integrity have been identified, and the general integrity problem has been broken down into seven classes or types of problems and their solutions.

In the discussion that follows, examples of validity-checking criteria are given and the classes of integrity exposures are presented. Each type of exposure is explained via examples and by relating it to the primary technique used for correcting or avoiding that type of exposure in VS2 Release 2.

validity-checking criteria

The new or changed validity-checking criteria for the most part result from the change in philosophy from "accidental error" philosophy to the "adversary" philosophy which says that nothing the unauthorized program can do, accidentally or deliberately, can be allowed to compromise system security controls. The examples below illustrate the types of changes that have taken place.

One example of the new validation criteria is the *Time-of-Check-to-Time-of-Use* (TOCTTOU) Problem shown below:



In a multiprogramming system, if there exists a time interval between a validity check and the operation connected with that validity check, the variables involved in the outcome of the validity check must remain unchanged from the time of the validity check until the operation is complete. If this cannot be ensured, then there is the possibility that, through multitasking, the validity-check variables can deliberately be changed during this time interval, resulting in an invalid operation being performed by the control program.

The TOCTTOU consideration is perhaps the most significant of the changes in validity-checking criteria. The requirement it imposes has considerably influenced the direction of the integrity support in VS2 Release 2. For example, it is no longer acceptable

to validate such items as a user parameter list at the beginning of supervisor call (SVC) processing and then assume validity throughout that processing. Steps must be taken to ensure that the variables on which the SVC depends do not change throughout its processing; otherwise repeated validity checks must be performed once each time such variables are referenced. Listed below are several examples of steps that must be taken in various cases to meet the TOCTTOU validity requirement:

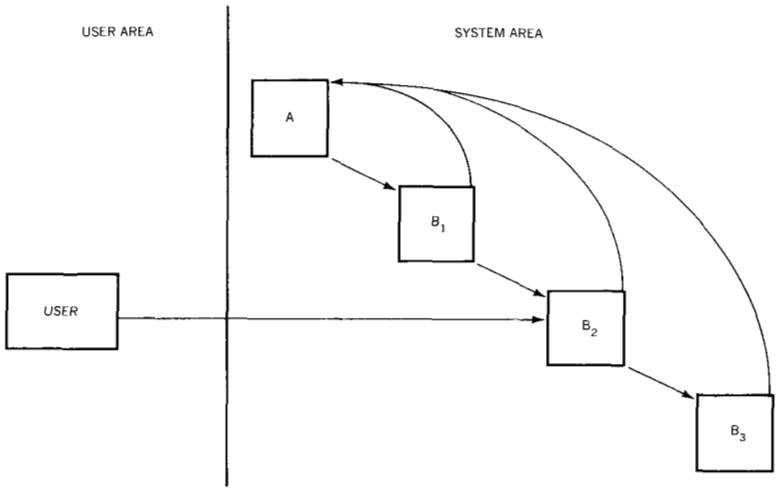
- Logical disablement (prevention of multitasking) for the duration of the validity check and related operation(s).
- Protection (in areas not accessible to user programs) of user-supplied addresses and other data for the duration of the validity check and related operation(s).
- Suppression (for the same duration) of user-initiated GETMAIN or FREEMAIN operations, in order to preserve the current status of main storage being accessed by the control program.

The key-switch technique (described later in this paper) also plays an important role in satisfying the TOCTTOU requirement for two particular types of integrity problems.

A second example of the changed validation criteria is the *valid-chain* concept of validity checking. Any validity check involves certain validity-check variables that are accessed and tested or compared with other variables to determine if a given operation is acceptable. The valid-chain concept simply says that such validity-check variables must be located through a chain of protected system control blocks that begins with a control block already known to be valid, or one known to be fixed at a given location at Initial Program Load (IPL) time, such as the Communications Vector Table (CVT) in OS/MVT. (This does not imply that a system routine must go back to the CVT or similar control block everytime it wants to establish a valid chain. Typically, a control block address not too far back on such a chain is available and already validated in a register. For example, in VS2 the first load of an SVC may receive control with a valid Task Control Block (TCB) address in a register.) The following is an example illustrating the valid-chain requirement.

Figure 1 shows a case where a user program provides the control program with the address of a control block B_2. Assume that the validation procedure that must be carried out before the address of B_2 can be accepted as valid is one of verifying the address to be on a chain of similar control blocks, located via block A. One way to perform such validation would be to note that the B-type control blocks on such a chain all point back to block A. Then one can take the user pointer to B_2, B_2's pointer to A, and

Figure 1 The valid chain concept



follow the chain from A to ensure that B_2 is in fact on the chain. However, this is a clear violation of the valid chain rule. A's address has been obtained by assuming the validity of the control block (B_2) that was still in the process of being validated. A user attempting to compromise system integrity could have provided a counterfeit B_2, which pointed to a counterfeit A, which in turn pointed to a counterfeit chain containing B_2.

The correct approach then is to separately locate control block A (through a valid chain) and then search the chain for a match on the user-supplied B_2. Note, however, that when only the "accidental error" is considered, the above USER to B_2 to A to B_2 validation is an acceptable mechanism because the chances are negligible that a user error would extend to a valid USER to B_2 to A to B_2 sequence. Such mechanisms were in fact used in prior systems, such as OS/MVT, to achieve accidental-error protection.

A final example of enhanced validity-checking procedures is the simple concept of a "full" check. This stems directly from the change in philosophy and refers to the fact that a previous validity check that would have detected most possible user errors before they caused system damage, must now be upgraded to detect all such user errors, deliberate or accidental. One common example of this type of change is in validating that a given control block is in a user-accessible area. In many cases in the past, the validation that would have been done would check just the first word of the control block; now consideration must be given to the fact that a page (4K) boundary might fall within the control block. (Validating a single word of storage in System/360

or /370 verifies only that the 2K/4K block containing that particular word is accessible to the user.) Since the part of the control block beyond the page boundary might not be accessible to the user in question, it also must be checked for accessibility. It should be noted that less than 100 percent complete validity checks and other integrity-related "omissions" in previous systems were not generally due to poor design or coding. In many cases they reflect valid trade-offs with respect to critical design-point/performance considerations relative to earlier releases of OS/360 systems.

classes of integrity problems

The following classes of integrity problems have been identified:

- System data in the user area.
- Nonunique identification of system resources.
- System violation of storage protection.
- User data passed as system data.
- User-supplied address of protected control blocks.
- Concurrent use of serial resources.
- Uncontrolled sensitive system resources.

Each of these classes of problems is described below in connection with the primary techniques used to solve the problem in VS2 Release 2. Examples are given in general terms as much as possible to avoid detailing specific problems that could be used to compromise systems prior to VS2 Release 2.

The first two problems in the above list are somewhat more general in nature than the remainder of the problems. The techniques, if they may be called that, of solving these problems can only be stated in very general (or imprecise) terms, or are specific to a given instance of the problem. Because of this, these two problems are addressed together before going on to the more explicit integrity mechanisms being used in VS2 Release 2.

system data in the user area

System data in the user area refers to the problem where sensitive system data, which should be located in an area of storage protected from the unauthorized user, is in fact located in a user-accessible area. In general, the types of system information that must be protected from the user are as follows:

- Code (and the location of code) that is to receive control in supervisor state or system key (0–7 in VS2 Release 2), or that is APF-authorized. (APF-authorization is described in a section later in this paper.)
- Work areas for such code, including areas where the contents of registers are saved/restored.
- Control blocks that represent the allocation or use of system resources.

Note that in VS2 Release 2, as in previous MVT and VS2 releases, system code and data is normally protected from user modification via protect/storage keys. Keys are only used to protect user programs from each other in the case of Virtual = Real (V = R) program. Segment protection is used otherwise.

Since the problem is very general, the solution also must be stated in somewhat general terms—that is, either relocate sensitive system data to areas protected from the unauthorized user, or modify the way in which the system uses the data such that it is no longer sensitive.

Probably the most common example of this type of problem is the case where the system uses an address, located in storage modifiable by the user, to branch to a program in supervisor state or key 0–7. The user can of course gain control in supervisor state or system key simply by modifying this address at the appropriate time. The solution to this problem is:

- If the routine being given control actually requires the privileged key or state, the address must be relocated to an area not accessible to the unauthorized user.
- If the routine being given control does not require the privileged key or state, the problem can be solved by replacing the branch with a SYNCH operation that gives control in user key and problem state, but allows control to be returned to the system in supervisor state system key.

nonunique identification of system resources

An operating system is essentially a resource manager. As such, if there is a case where it does not uniquely identify the resources it is dealing with, it becomes subject to integrity problems. These problems generally take the form of the ability of an unauthorized user program to counterfeit control program resources such as programs or control blocks, or to cause the system to intermix incompatible control program resources. *Nonunique identification of system resources* is the term used to refer to this problem.

The general solution to the problem can only be stated as the reverse of the problem; that is, the system control program must maintain and use sufficient data (protected from the user) on any sensitive control program resource, to uniquely distinguish that resource from any other control program or user resource. To be more specific than this, one must have a knowledge of the particular type of resource involved in the problem, as can be seen from the following examples:

- To be uniquely identified, a program must be identified by both name and library. Specifically, to prevent control program routines from being counterfeited, the VS2 Release 2

control program checks both name and library before satisfying a load request for an authorized program (a program to be executed in system key, supervisor state or APF- authorized). Such loads can be satisfied only from authorized system libraries.

- Certain types of resources such as copies of programs can be requested and used by both the user and the control program concurrently. In this case, the control program must identify the resource as belonging to both the control program and the user to ensure that the user is not able to delete the resource while the control program is still using it. This could result in an integrity exposure. Note that in this case, the "identity" of a resource is being extended to include the information that has become a part of the control program and as such is not deletable by the user.

The *key-switch technique* of changing from system key to user key is widely used as a validity checking mechanism in VS2 Release 2. Its purpose is to achieve simpler and more effective validity checking by making a system program, performing an operation in behalf of a user program, appear to be a user program for the duration of that operation. By switching from system key to user key, the system routine ensures that it will suffer the same validity-check failures as the user program would have suffered had it attempted to perform the operation itself.

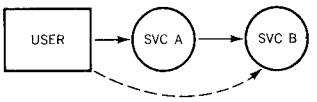
The ability to make use of this technique is dependent on the following two capabilities:

- The capability to define validity-checking mechanisms that are sensitive to key but not state (problem state versus supervisor state) as the difference between system and user. (Supervisor state must be used as the authorization mechanism that allows the key switch, and therefore must be able to be retained by the system routine in order to switch back to system key.) In general, this requirement may be described as the existence of a *state-switch* state on which validity checks do not depend, one or more other states on which various types of validity checks do depend, and the ability to hold the state-switch state simultaneously with any of the other states.
- A mechanism that allows the system routine to recover from the validity-check failure (for example, program check and ABEND) in order to cause the proper error messages, ABEND codes, and such to be given to the user. Note that since almost all VS2 Release 2 control program routines are protected by special recovery routines that do intercept such failures, this requirement is not a problem.

system violation of storage protection

Two types of integrity problems lend themselves particularly to the key-switch solution. *System violation of storage protection* is a problem where a system routine, operating in one of the privileged system keys (0–7), performs a store or fetch operation in behalf of a user routine without adequately validating that a user-specified location actually is in an area accessible to him. The key-switch technique is now widely used throughout VS2 Release 2 by system key routines performing store/fetch operations on such user areas (control blocks, buffers, and parameter lists and the like). It is a simple, effective technique that allows the hardware to do the validity checking. (A new key-switch instruction is available and used by VS2 Release 2 for this purpose. This reduces overhead and considerably simplifies the mechanics of key switching, which otherwise would have required LPSW instructions each time.) Also, it is also a low-overhead method of avoiding the Time-of-Check-to-Time-of-Use problem.

Figure 2 User data passed as system data



In addition, with respect to this particular integrity problem, there is a variation of the basic key-switch technique that provides an automatic solution to one of the most difficult aspects of system integrity support—namely, ensuring that a given solution has been applied in all cases of the problem. Thus, while the key-switch technique is effective as a solution, there is always the problem of ensuring that key switches have been inserted at all the appropriate places. The variation on the basic technique that provides this assurance is the change in VS2 Release 2 to run certain portions of the control program in a limited store/fetch state (that is, nonkey 0) instead of unlimited store/fetch state (key 0). This has the effect of forcing a key switch at the appropriate places (whenever a user area is referenced) because if the switch is not made, the control program's limited store state will cause the store/fetch operation to fail in the normal case, thus producing an error situation that must be corrected by a key switch. Large parts of the data management area (which is prone to the storage protection type violations), for example, will run in key 5 in VS2 Release 2.

user data passed as system data

User data passed as system data is a second type of integrity problem that lends itself to the key-switch technique. The potential for this problem arises wherever, as shown in Figure 2, it is possible for an unauthorized user program to use one SVC routine (routine A) to invoke a second SVC routine (routine B) that the problem program could have invoked directly. An integrity exposure occurs if SVC routine B bypasses some or all validity checking based solely on the fact that it was called by another SVC routine (routine A), and if user-supplied data passed to routine B by routine A either is not validity checked by routine A or is exposed to user modification after it was validated by routine A (the TOCTTOU problem). This problem does not exist if the

user calls SVC routine B directly because the validity checking will be performed on the basis of the caller being an unauthorized program. This confusion arises because of the various cases in the system where SVC routines operating in their own behalf invoke other SVC routines to perform operations that would not, and should not, withstand the normal validity checking applied to unauthorized programs. The problem is to identify the case where an SVC is operating in a user's behalf—that is, with unvalidated, user-supplied data that should undergo normal validity checking.

The solution to the problem requires that SVC routine A (which is aware of whether or not it has been called by an unauthorized program) ensure that the proper validity checking is accomplished. However, it is usually not practical for SVC routine A to do the validity checking itself because of the potential for user modification of the data prior to or during its use by SVC routine B (the TOCTTOU problem). The general solution, thus, is for SVC routine A to provide an interface to SVC routine B, informing routine B that the operation is being requested with user-supplied data in behalf of an unauthorized problem program (implying that normal validity checking should be performed).

In practice, in VS2 Release 2 most SVC B-type routines that could be subject to this problem use the key of their caller as a basis for determining whether or not to perform validity checking. Therefore, most VS2 Release 2, SVC A-type routines have simply adopted the convention of assuming the key of their caller before calling the SVC B routine. This solution is effective in the case of 2 levels of SVCs, as shown above, in *n* levels if necessary.

user-supplied address of protected control blocks

The *user-supplied address of protected control blocks* integrity problem can exist whenever the control program accepts the address of a protected system control block from the user. For most system control blocks this situation should not be permitted to exist. However, in certain cases it is permissible (with adequate validity checking), and even advantageous, to allow the user to provide the address of a system control block that describes his allocation/access to a particular resource (such as a data set) to identify that resource from a group of similar resources (for example, a user may have many data sets allocated). Inadequate validity checking in this situation creates an integrity exposure since the user program can provide its own (counterfeit) control block in place of the system control block and thereby cause a virtually unlimited array of integrity problems depending on exactly what sensitive data the system may be keeping in the control block involved.

The primary example of the potential for this type of problem in VS2 involves the Data Extent Block (DEB), which is effectively

the system's protected record of a user's ability to perform I/O to a data set. It serves to tell the system the location of the data set and any restrictions on the user's ability to access it. This avoids going through the open process each time I/O is requested to/from a data set. Since the DEB address is located in the Data Control Block (DCB) (a user-accessible control block), adequate validity checking must be done to ensure that the DEB cannot be counterfeited. The key to adequate validity checking in this and other cases of this type of problem is that the address of the control block in question should not be treated as an address, but rather as an identifier. The function of the address in the DCB should not be to tell the system where the DEB actually is located, but rather to identify which of the set of valid DEBs associated with the user is the one associated with the current I/O (or other) operation being requested. To meet this requirement, a DEB Check (DEBCHK) mechanism was implemented; and for each user I/O operation and certain other operations in which the DEB is critical to system integrity, the DEBCHK mechanism uses a protected, jobstep-related table of valid DEBs to ensure that the address provided is that of a valid DEB associated with the user in question. Also, in some cases, the type of DEB involved (for example, QSAM, ISAM) is verified.

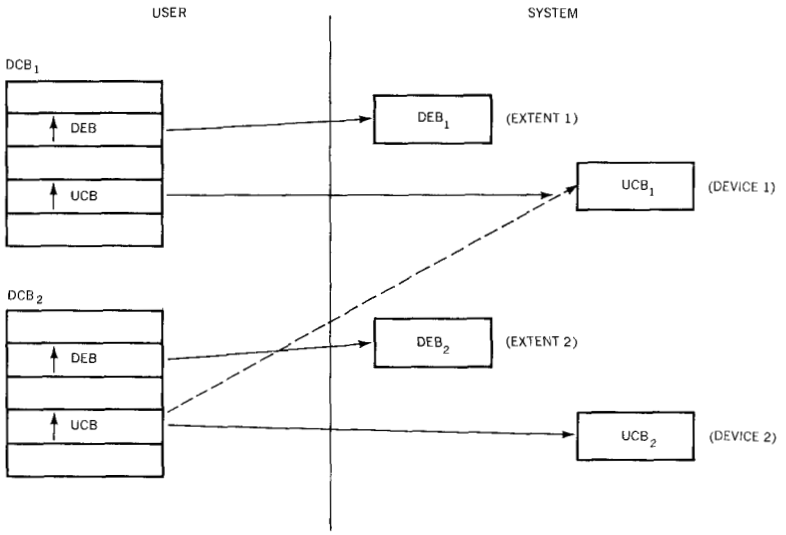
In order to minimize the performance implications of DEBCHK, a constant overhead mechanism that is not dependent on the number of entries in the DEB table has been designed. When a DEB is first created and its address placed in the valid DEB table, the offset of that address is then placed in the DEB itself. The validity check is as follows:

- From the DEB at the user-supplied address, obtain the offset into the valid DEB table.
- Verify that the offset does not exceed the size of the DEB table.
- Verify that the address at that offset matches the user-supplied address.

This mechanism eliminates the overhead of a table search and, more importantly, removes the possibility that DEBCHK could become a performance bottleneck in the case of user jobs having large numbers of simultaneous open data sets (large numbers of DEBs) which, by increasing the size of the DEB table, would increase the time needed to search the table.

Going back to the identifier concept, one must conclude that except for compatibility problems, the need for a DEB address in the DCB could be removed completely. The DCB could just as well contain only an offset into the valid DEB table, the same as that previously described for the DEB. After verifying that the offset was in the bounds of the table, the control program could use it to obtain the DEB address directly from the table.

Figure 3 A hypothetical DCB/DEB/UCB structure



While the validation mechanism as described functions well for the case where a single, protected control block address is used as the identifier of a resource allocated to a user program, complications can arise if various attributes of a user's allocation to a single resource are described by more than one protected control block. In such cases, the user must not be permitted to identify more than one of the set of protected control blocks describing that allocation unless there exists a mechanism whereby information contained in one of the blocks (such as unique keys, or chain pointers) can be used (after that block is validity checked) to verify that all other user-addressed blocks are actually associated with that user's allocation to the resource in question. If not, there is no validity check that can be performed to avoid incorrect combinations of resources or resource attributes being associated with that user. Why this is true can be seen in the following example involving the DEB. In VS2 the DEB is used to define the direct access extent limits of a user's data set, while the Unit Control Block (UCB) defines the device on which the data set resides. If the user were allowed to provide the address of both the DEB and the UCB and each was validity checked independently of the other, a method might be established to verify that (1) the user had access to the DEB-specified extent on some device, and (2) the user had access to some extent on the UCB-specified device. But the only way to ensure that the specified device and the specified extent go together with respect to that user is to have the UCB identified via information in the DEB (which is in fact the case in VS2) or vice versa.

Figure 3 shows a hypothetical DCB/DEB/UCB illustrating this problem. The solid lines indicate the correct pointers; the dotted line indicates one of several ways the data-set attributes could be confused (the extents of data set 2 are paired with the device of data set 1) because there is no interconnection between the DEB and the UCB. (Note that this is a hypothetical example, for in VS2 the DEB and UCB are interconnected.)

concurrent use of serial resources

While *concurrent use of serial resources* is a general problem for any multiprogramming system, there are two serialization considerations that specifically relate to system integrity. One is the TOCTTOU problem previously mentioned, which at times must be controlled with some form of serialization mechanism. The other concerns improper user manipulation of SVCs and again relates to the previous discussion on the "accidental error" philosophy. In VS2 Release 2, serialization mechanisms have been introduced in certain SVCs to prevent the user from utilizing multi-tasking to pass the same resource simultaneously to two parts of the system never designed to process that resource simultaneously. In general, the reason for the original lack of a serialization mechanism in such SVCs was the fact that only a deliberate user error would be likely to produce that situation, an event that did not have to be accounted for under the "accidental error" philosophy.

In VS2 Release 2, Enqueue/Dequeue (ENQ/DEQ) and a new hierarchical locking structure (developed primarily for multiprocessing serialization problems) are the primary methods used to control integrity problems relating to serial resources. The locking mechanism is used, for example, to prevent FREEMAIN from occurring on certain areas of storage for the duration of a given operation—a problem previously mentioned in connection with the TOCTTOU problem.

With respect to potential system integrity problems, it is critical that unauthorized programs not have access to the serialization mechanism. This requirement creates a problem with the use of ENQ/DEQ, not only for new areas of serialization control, but old areas as well, since it was determined that ENQ/DEQ was already being used in many areas where lack of serialization could cause integrity exposures. The problem was that ENQ/DEQ had never been a restricted function and, in general, if the user wanted to nullify a system ENQ by issuing the appropriate DEQ, there was no way to stop him. To correct this deficiency, a change was made so that ENQ/DEQ is now restricted to authorized programs for all major names of the form SYSZxxxx and for certain existing major names such as SYSDSN, SYSVTOC, and SYSPSWD.

The restriction problem did not exist with the new locking mechanism since it was by definition restricted to authorized programs from its inception.

uncontrolled sensitive system resources

In the ideal case, the system control program should maintain control over access to all system resources. However, there usually arises a need for certain special types of programs (for example; some system utilities) run as ordinary user programs (jobsteps), but that, because of the special nature of their function, must have the capability to manage certain system resources directly. This capability is provided through IBM-written and, in some cases, user-written special service routines (in the form of SVCs or special paths through SVCs) that bypass established resources-management controls normally imposed on user programs. Because there has been no way in the past for the control program to effectively differentiate the class of programs that require such special services from the totality of user programs, these special services have generally been made available to all user programs without restriction. The lack of restriction on such sensitive services results in system integrity problems. An example of this in OS/MVT is the unrestricted path through OPENJ that allows writing of the Volume Table of Contents (VTOC). To solve such integrity problems, yet allow this special class of programs to continue to exist, the *Authorized Program Facility* (APF) was introduced in VS2.

The following is a summary of the APF support provided in OS/VS2 Release 2. Two methods of restricting sensitive system resources/services are provided. Sensitive SVCs are restricted via a parameter on the SVCTABLE macro at system generation time. The SVC First-Level Interrupt Handler ensures that SVCs restricted in this manner are only accessible to programs having APF authorization, supervisor state, or system key (0–7).

For SVC's where only a part of the function they provide is sensitive, the capability of restricting a particular path through an SVC is provided (for example, the path through OPENJ that opens the VTOC for writing). This facility is provided through insertion of a TESTAUTH macro at the appropriate location in the SVC. TESTAUTH returns an indication that the program calling the SVC is either authorized or unauthorized; the SVC must then take appropriate action based on this return. TESTAUTH is capable of testing for supervisor state, system key, APF authorization, or any combination. Appropriate IBM SVCs are automatically restricted; however, the capability is provided for the security-conscious installation to restrict its own sensitive SVCs as well.

Nonsystem-key/nonprivileged-mode programs are authorized to access services, restricted as described above by being link-edited with an *authorized-state* indicator. This program authori-

zation is accepted by the system only from certain authorized system libraries and only on a jobstep basis. The following is a summary of how this functions.

An installation is given the capability at system generation or IPL time to define a list of authorized libraries from which the APF-authorized program attribute is recognized by the system. It is the installation's responsibility to control the contents of such libraries. (SYS1.LINKLIB and SYS1.SVCLIB are automatically considered authorized libraries.) The first load module in a jobstep basically determines the authorization for that jobstep. If the first load in the jobstep is not APF-authorized, it is impossible for any part of the jobstep to become APF-authorized. If the first load in the jobstep is APF-authorized, the jobstep remains authorized unless a LOAD request is satisfied from an unauthorized library, in which case the task goes to an ABEND. This is necessary to prevent such exposures as the use of the JOBLIB/STEPLIB facility to replace the second or subsequent loads of an APF-authorized program with a program of the correct name but from an unauthorized library. It is important to note the need for an ABEND. Simply turning off the jobstep's authorization does not suffice because of the possibility that an access path to a restricted resource (for example, a valid DEB allowing writing into a VTOC) may remain established after the authorization indicator is turned off. This path could then of course be utilized by the substitute, unauthorized module even though the authorization indicator had been turned off. It is the responsibility of the authorized program not to recover from the ABEND in a way that would allow the unauthorized module to execute authorized, or unauthorized with an established access path to a restricted resource.

Essential to the effectiveness of the APF-authorization mechanism is the fact that APF authorization is strictly program authorization, as opposed to user authorization. It is generally intended that any APF-authorized program should be executable as a jobstep by any user without damage to system integrity or security. The APF-authorized problem program is considered to be effectively an extension to the control program. As such, although it is allowed to bypass normal system controls on resources / services, it is responsible to provide the same or equivalent controls in any interface with the user. (The IEHDASDR program, for example, is allowed to bypass normal controls on access to direct access space; but before altering any data, it invokes the system password checking mechanism to ensure that its invoker is authorized to delete the data sets in question.)

In some cases, however, it may not be feasible for an APF-authorized program to apply the normal system control mechanism to its user interface. In this event, the use of the program must be

controlled. In VS2 Release 2, it is suggested that such programs be placed in a password-protected authorized library so that execution of the program is controlled (the password for the library is required when it is opened to allow fetching of the program).

As indicated above, it is necessary for the first load of an authorized program to be link-edit authorized—that is, with the authorized-state indicator. As long as all subsequent loads come from authorized libraries, the jobstep continues to run authorized. Second or subsequent loads should specifically not be marked APF-authorized since to do so would enable them to be executed as the first load of a jobstep which, because of their authorization, could cause unpredictable integrity problems.

If an APF-authorized program specifically wishes to LOAD a program from an unauthorized library and continue execution with authorization turned off (that is, not be terminated when the unauthorized program is loaded), the authorized program must turn off its own authorization indicator prior to issuing the LOAD request. In this case, it is the resonsibility of the authorized program to ensure that no access paths to restricted resources are still established when the unauthorized load is given control. It is also the authorized program's responsibility to ensure that there are no asynchronous routines running or yet to run that have a dependency on the program's currently authorized status.

The existence of multiple authorized libraries introduces a problem with respect to naming of authorized modules. It can be assumed that an executing, authorized program is aware of the correct name of a module it attempts to load. However, because an authorized program normally executes as a jobstep executable by any user, it cannot control the identity of JOBLIBs, STEPLIBs, and others since these libraries are identified via JCL. Therefore, there is an exposure that if two modules of the same name exist on different authorized libraries, an authorized program attempting to load one of these modules could get the other if the user executing the authorized program were to (deliberately or accidentally) improperly JOBLIB, STEPLIB, or concatenate the two libraries in question. The existence of this type of exposure requires the additional restriction that duplicate module names not be permitted across authorized libraries. Because this restriction must be enforced by the installation, it appears that a naming convention would be the simplest way to permit effective monitoring of this restriction.

miscellaneous mechanisms

While the previous portion of this paper described the techniques representing the primary integrity control mechanisms used in VS2 Release 2, there are several lesser-used techniques

that are useful to handle specific aspects of certain types of integrity problems. Two significant mechanisms are now described.

In connection with the TOCTTOU problem, it is frequently necessary for the control program to temporarily move user-supplied parameter lists and so forth to areas of storage that the user program is not allowed to modify. While there is no specific need relative to the TOCTTOU problem to fetch-protect such areas from the user program, it does no harm since there generally is no need for the user program to reference such areas for the duration of the move. Therefore, when moving such parameter lists and other items, the control program in many cases takes advantage of system-key fetch-protected subpool support in VS2 Release 2 to obtain storage for the moved data that is both fetch and store-protected from the user program. This successfully avoids additional validity checking that would otherwise be required to ensure that the user program did not pass a "parameter list" that was in reality an area of storage fetch-protected from that user program. (The user program could of course look at the parameter list if it were moved to a nonfetch-protected area.) This technique in fact turns out to avoid a fairly substantial validity-check overhead, since the key-switch technique does not work for the user-key to system-key type move operation.

The *protected copy technique* mentioned previously can also be useful with respect to problems other than the TOCTTOU problem. During OPEN/CLOSE/EOV, for example, the DCB (user control block) is involved in both the system-data-in-the-user-area problem and the system-violation-of-store-fetch-protection problem. Not only does the DCB have to be accessed many times by OPEN/CLOSE/EOV, but also, sensitive indicators are set and tested in the DCB. A straightforward total solution involves relocating the sensitive indicators to a protected control block and inserting the appropriate validity checks or key switches in OPEN/CLOSE/EOV. Instead, a solution was adopted whereby a second (protected) copy of the DCB is created and used throughout most of OPEN/CLOSE/EOV. The actual (user) DCB is updated at appropriate times, but otherwise the copy is used. This eliminated the need to relocate the sensitive indicator fields and also the need to validate the DCB each time it is referenced.

Installation responsibilities

VS2 Release 2 is provided with basic system integrity support. However, to ensure that system integrity is effective and to avoid compromising any of the integrity controls provided in the system, the installation must assume the responsibility for the following items.

The installation must be responsible for the physical environment of the computing system. Operations personnel and system programmers have, in effect, uncontrolled access to certain portions of the Operating System. These persons are considered to be under installation control and are presumed trustworthy as far as system integrity is concerned.[3]

The installation must ensure that their own modifications and additions to the control program do not introduce any integrity exposures; that is, all user-written authorized code (such as a user SVC) must perform the same or an equivalent type of validity checking and control that the VS2 Release 2 control program employs to maintain system integrity.

The installation must be responsible for the adoption of certain procedures that are a necessary complement to the integrity support within the operating system. Several examples of such responsibilities are now given. More detail on this topic can be found in VS2 Release 2 documentation.[4,5]

The installation must password-protect appropriate system libraries. System integrity clearly cannot be maintained if system code and data are exposed to arbitrary modification by any user on the system. For integrity purposes, it is generally sufficient to protect appropriate libraries from write access (no password is required for read access, but a password is required for write access). However, for security purposes, it is necessary to protect certain system data sets (for example, the PASSWORD data set itself) from read as well as write accesses. To improve the operational characteristics of such protection, password requests for data sets being opened by the system are suppressed during IPL and system task initialization.

The checkpoint data set produced by the Checkpoint/Restart facility contains sensitive system data normally protected from the user. Therefore, maintaining system integrity requires that such data sets be protected from modification (or from being counterfeited) prior to their use by the Restart facility. VS2 Release 2 implements a facility whereby the installation can adopt a set of special procedures/controls over checkpoint data sets that will eliminate their potential for compromising system integrity. The control mechanism involves a combination of:

- System/operator validation of checkpoint data sets.
- External labeling procedures for checkpoint volumes.
- Off-line control of checkpoint volumes.
- Prohibition of I/O to checkpoint data sets, except through the Checkpoint SVC (authorized programs excepted).

Concluding remarks

In conclusion, three areas of concern that continue to be brought up repeatedly with respect to system integrity support are now addressed. These questions concern the impact of integrity support on the system as a whole, the feasibility of integrity on other systems, and the questions of what level of integrity has been achieved in VS2 Release 2 and what in fact constitutes an adequate level of system integrity.

impact on the system

Of primary concern to any security-conscious installation is the overall impact to the VS2 Release 2 system and to installation procedures resulting from the introduction of system integrity support. While it is, of course, not possible to gauge the impact on any given installation without specific knowledge of that installation, the impact in general should be low. The following addresses some potential areas of impact.

With respect to performance, there are some extra CPU time and real storage use due to the enhanced validity checking and so forth. However, the information available to date indicates that this will not significantly degrade system performance. Insofar as possible, techniques have been used that minimize the impact of increased validity checking, as can be seen in the following examples. The DEBCHK type of validity checking, performed at each I/O operation, could have been a performance problem had it been designed such that the DEBTABLE search overhead increased as the number of DEBs (open data sets) grew larger. However, as previously described, the design is such that the search overhead is essentially constant no matter how many data sets are open, and in fact represents only a small increment to I/O validity checking in previous systems. The validity checking needed to ensure that the system does not violate store/fetch protection also could have become a problem were it not for techniques such as the key-switch technique which allows the mainline validity checking to be streamlined with the error handling done in recovery routines.

There are not expected to be any integrity-induced incompatibilities (with respect to user interfaces documented in Systems Reference Library manuals) that cannot be eliminated via a relinkedit of the program to make it APF-authorized. There is at present only one known integrity-induced incompatibility that would require such a relinkedit—the previously mentioned restriction that prevents all but authorized programs from doing I/O directly (that is, not through the checkpoint macro) to a checkpoint data set.

As with any new system there will be changes required in existing installation system specifications. The primary problem will

be the need (if concerned with security) to be aware of the nature of the new integrity support so as not to undo what has been done in VS2 Release 2. One example of a requirement in this area is that any user-written system modules executing with system key or state must be link-edited (into an authorized library) with the reentrant attribute. This must be done to ensure that such modules are loaded into the proper subpool protected from user access.

Probably the area with the greatest potential for significant impact is the existence of code that makes use of existing integrity exposures in the system (not necessarily malicious use of such exposures, but the casual use of a "hole" to get into supervisor state without writing an SVC). While no one has been able to assess the extent of such practices, there is the mitigating factor that most of the impact in this area should have been felt in OS/MVT Release 21 or at least by VS2 Release 1, since the "holes" most likely to be used for such purposes are eliminated in those releases. With respect to correcting the impacted programs, some will be able to be corrected via a relink-edit to APF-authorized; others will of course require code changes. However, for the future, the existence of APF support should make the creation of such special interfaces a simpler operation. With APF control, for example, it becomes safe to leave an SVC on the system that returns control in supervisor state and thus eliminates the need for authorized programs to use system exposures to accomplish this state switch.

other systems

At this point in the state-of-the-art of system integrity it is not clear if it is possible or reasonable to provide system integrity in all operating systems. There are at least two essential design concepts that must exist in order to provide system integrity:

- System/user isolation.
- User/user isolation.

As with most other operating system capabilities, these types of controls require increased storage and CPU overhead. OS/MVT already had these essential design concepts and while this isolation of the system from the user and the user from other users was not always strictly enforced in MVT, it was possible to carry these design concepts forward into VS2 and introduce the stricter controls necessitated by system integrity with a minimum of increased overhead. To introduce such basic integrity-design requirements in a system where such design concepts previously did not exist or existed only minimally would likely result in a final system very different from the original system in terms of main storage requirements, CPU overhead, and operational characteristics. In many environments this may simply not be practical or justifiable. There are simpler, more procedural methods of achieving the desired security control in such systems.

level of integrity

The question must always be asked as to what level of system integrity VS2 Release 2 will actually have and, perhaps more importantly, what level of integrity is adequate. VS2 Release 2 will not have total system integrity because total system integrity, or security, does not exist anywhere in the real world. If someone is willing to spend enough and risk enough, any security system can be broken.

Cost and risk are the key concepts. Security, or system integrity, does not have to be 100 percent foolproof. It only has to be at a level where the cost and risk involved in breaking that security exceed the benefits to be gained by doing so, or exceed the cost and risk of obtaining the same benefits in another way.

Thus, while the goal has been, and must be, 100 percent system integrity, it is sufficient to achieve a level of security integrity in VS2 Release 2 such that the cost/risk of "breaking" that system is significantly greater than the benefits to be gained in compromising such a system in the normal commercial environment. From limited user feedback, it appears that this level has been met with respect to our level of understanding of the problem and the level of subtlety of "exposures" now being corrected in the system. There of course always remains the possibility of the exposure that is easily understood and simply fixed, but somehow gets overlooked. Any such design exposure may be remedied via an Authorized Program Analysis Report (APAR). In addition, it is worth noting that even though such random exposures may exist, the difficulty (cost/risk) of finding one and successfully using it in a system of the complexity and size of VS2 Release 2 is by no means trivial.

Perhaps the single key factor in achieving this level of system integrity has been the "fix all exposures" approach adopted very early in the integrity effort for VS2 Release 2. This approach, in effect, says that any integrity exposure is to be fixed, no matter how unlikely it is that it could be used to violate system security. For example, many known integrity exposures have timing constraints that appear to make it very difficult to actually use the exposure to compromise the system. Such exposures have been fixed regardless, because it was learned also very early in the integrity effort that it is very risky to attempt to classify integrity exposures according to severity. Too often exposures that appeared very difficult to use turned out to be simple to use when more information came to light. This "fix all exposures" approach is essential in any serious effort to provide an adequate level of system integrity. The state-of-the-art in this area is not yet, and possibly never will be, at a point where a reasonable decision can be made that it is acceptable not to correct a known integrity problem.

ACKNOWLEDGMENT

The author would like to acknowledge P. Byrne, E. Cassorla, L. English, R. McAllister, and others who contributed to the IBM system integrity development effort.

CITED REFERENCES AND FOOTNOTES

1. An *authorized program* in VS2 Release 2 is a program running with system key (keys 0–7) and/or supervisor state and/or Authorized Program Facility (APF) authorization. In general, the concept of an authorized program can best be thought of as a "trustworthy" program in the sense that it is either part of, or a logical extension to, the control program, and therefore can be allowed certain privileges not granted ordinary user programs without fear of compromising system integrity.
2. Store and fetch protection correspond respectively, to preventing write or read access to portions of main storage.
3. *Considerations of Physical Security in a Computer Environment*, Form No. G520-2700, IBM Corporation, Data Processing Division, White Plains, New York.
4. *IBM System/370, Introduction to OS/VS2 Release 2*, Form No. GC28-0661, IBM Corporation, Data Processing Division, White Plains, New York.
5. *OS/VS2 Planning Guide for Release 2*, Form No. GC28-0667, IBM Corporation, Data Processing Division, White Plains, New York.