PROJECT MAC — Originally published August 5, 1974
Restored October 15, 2016
Computer Systems Research Division — Request for Comments No. 59

PATTERNS OF SECURITY VIOLATIONS: MULTIPLE REFERENCES TO ARGUMENTS

by Harry C. Forsdick and David P. Reed

Restoration Note: Because of concern for revealing security vulnerabilities in running Multics systems before patches were distributed, CSR-RFC-59 was withdrawn immediately after it was first circulated. As a result, neither an original paper copy nor its word processing source file in RUNOFF format have been located. However, the contents of RFC 59 were later reprinted as part 5, pages 34–49, of Project MAC Technical Memorandum TM-87, *Ancillary Reports: Kernel Design Project*, edited by David D. Clark and published June 30, 1977.

As a (hopefully temporary) measure pending location of an original copy of RFC 59, this cover page was fabricated and the attached content pages were extracted from a scanned copy of TM-87.

A date disagreement raises a question about whether RFC-59 was revised. An RFC index published in May 1980 reports the publication date as “8/5/74”. However, the table of contents of TM-87 reports the publication date as “11/8/74”. The earlier date reported by the index is more consistent with its position in the RFC numbering scheme, and also with David Reed’s recall of the timing of the reported work, and that earlier date appears at the top of this page. However, until original paper copies have been located, one should keep in mind that the content reprinted in TM-87 may be a later revision.

J. H. Saltzer

Patterns of Security Violations: Multiple References to Arguments

by Harry C. Forsdick and David P. Reed

1. Introduction

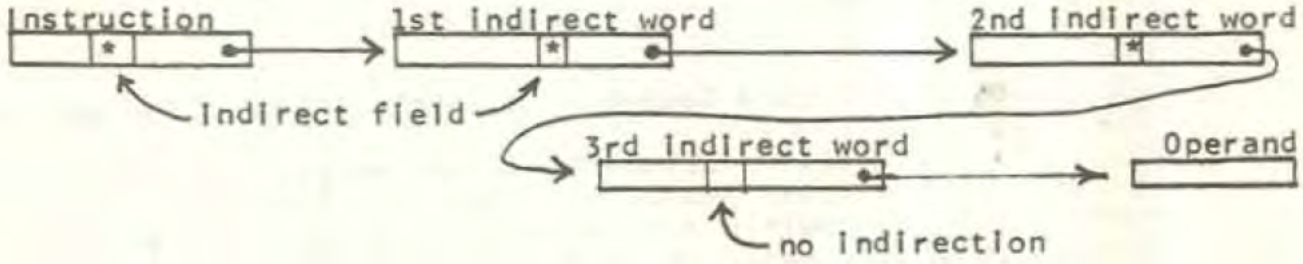
A large class of potential holes in the security of an operating system is characterized by the use of an argument more than once. On the surface, this situation appears to be harmless: multiple references may be inefficient, but they seem to be functionally equivalent to a single reference. But, are they? If the value of an argument could change between one reference and the next, the possibility of an error in the logic of the program using the argument exists. The assumption made by the author of the program that an argument could only be altered by the program or agents of the program is violated. How could an argument change in this invalid way? A simple conceptual scheme on a multiple process system is for one process to execute the call, supplying the arguments and a second process which has access to the values of the arguments, to perform, at the appropriate time, the alteration on the arguments. Whether or not a multiple argument reference leads to a breach of security depends on how the information gained from each reference is used. If the results of a test on one reference to an argument determine how the information of a second reference is used, then a exploitable hole in the system probably exists. More specific conclusions on the correctness of multiple references to an argument depend on the semantics of the particular program under analysis. Richard Bisbey of the Information Sciences Institute of USC brought this subject to our attention. He described the multiple referencing of arguments as a general pattern for a class of security holes and cited several instances of this pattern in Multics.

With these ideas as motivation, the Multics gate entrances to ring 0 were examined to determine if such multiple references to arguments were being made and if so, the implications of such flaws. Of the approximately 170 entrypoints to ring 0 through the hcs_ gate, about 50 were found to make multiple references to their arguments. Nine of these instances were potentially serious breaches of security in the Multics system. All of these breaches are easily fixed by copying arguments and then referencing the local copies.

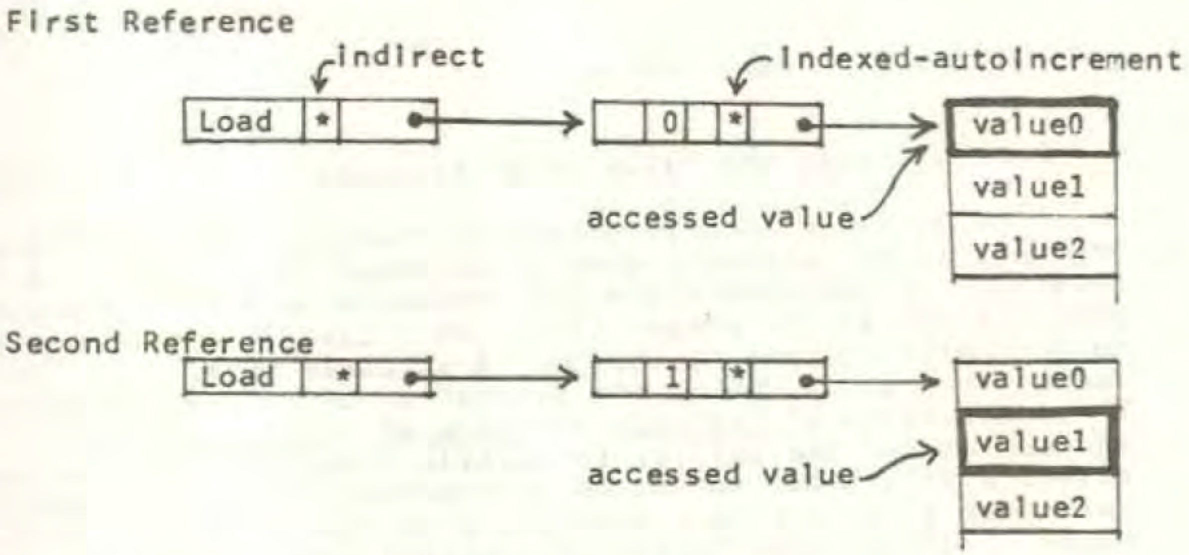
2. How to Change the Value of an Argument

The multiple process method of changing the value of an argument is conceptually simple, although in practice, it is necessary to coordinate the two processes so that the argument gets changed at the proper time. This task is often impossible to accomplish except by chance. A slightly more complex mechanism however, makes the alteration of an argument trivial. The combination of <u>indirect</u> and <u>indexed - autoincrement</u> addressing and the ability to cascade these modes of addressing allows a programmer to set up an argument list so that <u>each</u> reference to an argument accesses a different value. On the H6180, Indirect then Tally (IT) address modification is one of the kinds of indirect addressing and the Increment Address - Decrement Tally - Continue (IDC) variation on the IT modifier is an example of indexed - autoincrement addressing.

First, consider indirect addressing. Typically, there is a field in an instruction which can specify that the operand address points to a cell (the "indirect word") which contains the actual address of the operand. In addition, with cascading, a field in the indirect word can specify that the indirection process should continue at least one more level. For example, the diagram below depicts three levels of indirection:



For the indexed-autoincrement mode, there are two additional fields in indirect words: the indexed-autoincrement field and the count (tally). When an indirect word with the indexed-autoincrement addressing mode is accessed, the count is added to the address and used as the effective address of the indirect word. In addition, the count field is incremented by 1. Thus, each time an indirect reference is made through an indirect word with the indexed-autoincrement addressing mode, the effective address is one location higher. This is very useful in accessing tables -- in our particular case, tables of values for a single argument. For example, the diagram below depicts two consecutive references to an argument. The indirect word is part of the argument list set up by the calling procedure. In the first reference, the count is zero and thus the value accessed is the first value in the array of values.



The count is automatically incremented by one so that on the second reference, the value accessed will be the second member on the array.

If arguments are accessed by indirection (as they are in Multics) it is quite easy for a (malicious) programmer to set up an argument list so that each reference to an argument accesses a different cell. A number of machines (for example H6180, UNIVAC 1108) have the addressing features similar to the ones described above and thus systems running on these machines are susceptible to the problem of arguments changing values at unexpected but predictable times.

3. Classes of Errors Caused by Multiple Argument References.

The last Section established that multiple argument references can cause problems. There are four types of errors that arise from multiple references to arguments that are characterized by patterns of reading and/or setting the arguments. The illustrations below are stated in terms of double references, although the discussion applies equally well to any number of multiple references.

1. Double Reads: In this class of error, an argument is read twice. The value read the first time is tested and the result of that test determines how the value read the second time is used. The following program fragment illustrates this type of error:

```
if argument = 'pds' then switch = 0;
:
:
if switch = 0 then ....;
           else .... (reference to argument) ....;
```

The value obtained by the second reference to argument could very well be 'pds', a state that is inconsistent with the original _If_ statement.

2. Setting then Reading: Another common class of error occurs when a procedure initializes an output argument to a certain value and then relies on the integrity of that value. The scheme outlined in Section 2 works equally well for reading or setting arguments. Thus, it is possible for a user to cause a called procedure to use a value that is outside of its control. The following program fragment illustrates this type of error:

```
          a_ename = 'mailbox';
          :
          :
          call delentry$dfile(dirname,a_ename,code);
```

Between the points a_ename was set and used, its value could be changed to any value the user desired.

3. Setting twice: A slightly less obvious, yet potentially equally damaging error arises when an output argument is set twice. Damage results in situations where the value to which the argument is first set is to be hidden from the calling procedure by storing the second value. Again, since the scheme of Section 2 works equally well for reading and writing a history of argument values can be developed. This history is a potential privileged information leak. The following program illustrates this point:

```
  argument_code = error_table$entry_not_found;
  :
  :
  argument_code = error_table$no_access_to_file;
      /* Hide existance or non-existance of file from user. */
```

4. Passing an Argument: A "delayed" error can arise when an argument to one procedure is passed directly without copying to another procedure. This is because the value of the argument resides in an address space that is not protected (the user's address space). In Multics, the scheme described in Section 2 does not cause a problem because an entry in an argument list for an argument to the calling procedure points directly to the value of the argument. Thus, there can be no malicious addressing modifiers in the argument list. The more general multiple process scheme, however, is still effective in changing the value of the argument. For example, if procedure A is called with argument X by a user procedure, and A in turn calls B supplying X (without copying) as an argument, then the value of X can be changed by the multiple process scheme during the time B is running. This problem is made more serious by the tendency for argument validations to be dropped (for efficiency reasons) in

procedures that are internal to the protected part of a system.

5. Multiple References to Pointer Qualified Arguments: Quite often a pointer to an argument is passed to a procedure when the actual argument is a complex data structure. Again, the multiple process scheme can cause the actual data item to be altered during the running of a called routine. Copying the pointer into a local variable and performing references through this local copy does not solve the problem since the actual <u>value</u> of the argument can be changed by the multiple process scheme.

4. Methods of Recognizing Multiple References

In a large system it is very difficult to discover instances of the errors ourlined in Section 3. Two alternative methods of attack were taken in out study of Multics. One technique is to perform an analysis of the text of all procedures that are interfaces between the critically sensitive part of the operating system (ring 0 gates in Multics) and user programs. This analysis is aided by the cross reference listing produced by the PL/I compiler. Certain patterns in the cross reference listing for arguments indicate that multiple references are being made. The main advantage to this approach is that if done correctly, it will yield <u>all</u> instances of multiple argument references. The main disadvantage is that it is a time consuming task.

There are two defects in the cross reference technique. First, all references are listed together; thus it is impossible to tell by looking at the list which kind of reference (read, write, appearance in an argument list) occurred. The inability to distinguish in the cross reference listing between argument list appearances and reads and writes makes the analysis more difficult. The second defect of the cross reference technique is more serious. The appearance of a reference to a name in the text of a PL/I program does not guarentee that there will be a corresponding reference to the value of the name in the instructions emitted by the compiler. There could be zero or more references depending on optimizations performed by the compiler and the form of the actual reference. As an example of the last exception, the statement

```
x = convert(argument,z);
```

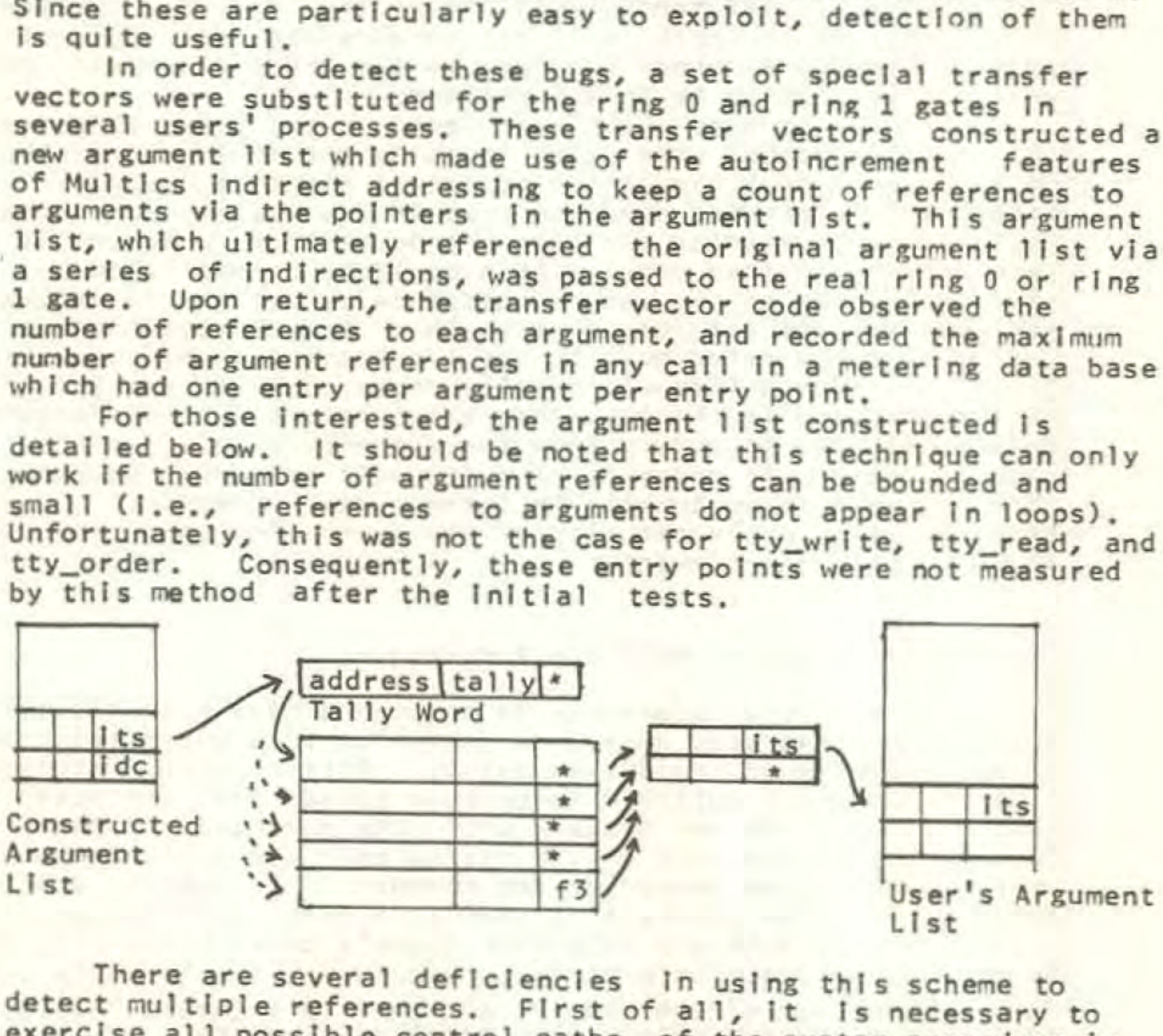
doesn't actually reference the value of the argument. The value of z is converted to a value whose type is the same as the <u>type</u> of argument and stored into x. Similarly, a reference to the length of a string does not reference the string, but rather the descriptor of the string. Thus, searching the cross reference list for multiple references can cause false alarms. On the other hand, the cross reference list provides no help in spotting references to arguments that are contained within loops.

It is conceivably possible to mechanize this process so that multiple references to arguments could be discovered by an automatic analysis. This task would fit in easily in the framework of the PL/I compiler since all of the necessary information is already available within the compiler.

The second technique for discovering multiple argument references involves monitoring the actual use of arguments passed to interfaces and noting any arguments that were referenced more than once. The mechanism used to exploit multiple references to arguments noted in Section 2 can also be used to detect multiple references to arguments at runtime. While all multiply referenced arguments cannot be detected in this way, many which can be exploited via the autoincrement mechanism will be found. Since these are particularly easy to exploit, detection of them is quite useful.

In order to detect these bugs, a set of special transfer vectors were substituted for the ring 0 and ring 1 gates in several users' processes. These transfer vectors constructed a new argument list which made use of the autoincrement features of Multics indirect addressing to keep a count of references to arguments via the pointers in the argument list. This argument list, which ultimately referenced the original argument list via a series of indirections, was passed to the real ring 0 or ring 1 gate. Upon return, the transfer vector code observed the number of references to each argument, and recorded the maximum number of argument references in any call in a metering data base which had one entry per argument per entry point.

For those interested, the argument list constructed is detailed below. It should be noted that this technique can only work if the number of argument references can be bounded and small (i.e., references to arguments do not appear in loops). Unfortunately, this was not the case for tty_write, tty_read, and tty_order. Consequently, these entry points were not measured by this method after the initial tests.



There are several deficiencies in using this scheme to detect multiple references. First of all, it is necessary to exercise all possible control paths of the system procedure in order to find all of the cases of multiple references (so some holes may pass unnoticed). Secondly, this technique produces many false alarms, since the code produced by the PL/I compiler may produce multiple indirections through the argument for one logical reference (this may or may not be a bug). Also, structure or array arguments may have subparts, all of which are singly referenced, but through the same argument pointer. Another problem is that PL/I sometimes copies argument pointers by indirection upon entry to a multiple entry point procedure (the case occurs if the same name appears in different positions in several formal parameter lists). This results in only a single reference being detected by this technique, even though multiple references may be made. The last problem is that arguments which are passed on to internal routines will not be caught, since PL/I indirects through the argument list once to get the address of the argument which is passed on. Even if the argument is referenced multiply by the internal routine which receives it, this will not be done via the indirect chain provided to the external routine by the transfer vector, and will not be counted by this technique.

Most of the bugs which were found in the current system by the auditing method were also found by the monitoring method. This suggests that the latter technique might be useful in attempting to prevent possible bugs in the system from being exploited, by crashing the user's process if an argument is referenced more than once. (This could be accomplished by causing a fault on the second reference by using a fault tag 3 indirect word as the second entry in a two element array of indirect words referenced by the idc autoincrement mode.) Certainly, such a firewall has its costs, both in runtime efficiency, and in the fact that all innocent multiple argument references must be purged from the system, as well as the security holes, in order for the firewall to work. Nevertheless, this may well be worthwhile in attempting to prevent retrogression in the security of the system for some users with high security requirements.

5. The Semantics of Multiple References

Once multiple references to arguments have been discovered, there is a final step needed to determine if a potential breach of the security of the system exists. This requires matching the information about multiple references gained from the essentially syntactic check on the program with the semantics of the program in relation to the rest of the system and the basic assumption that arguments can change at any moment. This step is quite difficult. To be complete, a similar effort would be required to justify that a multiple reference doesn't cause a security hole as to justify that the program _is_ secure. But, shortcuts can be taken: knowledge of the meaning assigned to arguments helps in isolating serious problems from harmless mistakes.

Of all of the steps in the technique for discovering errors due to multiple argument references, this is the most difficult step to mechanize. A very large amount of knowledge about the operation of the system must be used to determine whether or not a multiple reference is a serious error. The major benifit of searching for the pattern of multiple references is that areas of the program text which deserve close analysis are isolated.

6. Results of Applying this Approach to Multics.

An analysis of the Multics ring 0 gate entrances was performed. First, multiple references to arguments were discovered using both the cross reference listing technique and the monitor technique. Next, each entrypoint that had arguments that were multiply referenced was analyzed to determine the effect of the multiple reference. A list of the entry points tested and the results of those tests are found in Appendix 1. Numerous multiple argument references were uncovered. In most of these cases we were able to conclude with a high level of confidence that no errors result from these references. In a number of other cases, however, serious breaches in security were discovered.

The simplest and most glaring error was due to a multiple argument reference in "stop_process." By exploiting the multiple reference in the manner previously described, <u>any</u> process in the system could be stopped (including the initializer process). A less selective denial of service existed in "status_" and "status_long"; by setting up a certain form of argument list, these routines could be made to lock a lock that would never be unlocked. This would eventually cause the system to crash. It is possible to direct "tty_write" to send an unending stream of characters to a terminal. This has the effect of tying up the entire system and causing the appearance of a crash.

Other errors were found that were either deemed less serious or less obvious how to exploit. Because of a multiple reference to an argument in "add_inacl_entries" it is possible for a user to specify the initial access control list <u>for any ring</u> on any directories that he may create. This seems like a serious error, but it is difficult to see how to exploit it. In "printer_dcm" it seems possible, once a printer has been seized, to address any other printer. In "tdcm_message", multiple argument references make it possible to print inconsistent messages on the operator's console. Finally, assuming that it is possible to get past the "hphcs_" gate, it appears possible to set up inconsistent information in tables that record the state of tape drives by a call to "tdcm_add_drive".

One additional error due to a multiple argument reference is now known. At first we had classified the entrypoint "sfblock" as being in the class of entrypoints that did not have multiple references. A subsequent communication from Richard Bisbey pointed out a fairly subtle error in this entry to the supervisor. A portion of one of the arguments contains an index into a bit string stored into the PDS (an important ring 0 data base), and is first validated to be within range. It is then used to select a bit in the bit string to be set to one. If the second reference gives an out of bounds index, then any bit in the PDS may be set. Both of the multiple reference detection techniques had failed to find this error. The monitor technique failed because the argument is referenced via a generated pointer; the auto-increment technique for exploiting such holes will not work for this instance. The cross reference listing technique probably failed due to human error.

Several direct conclusions come out of our experience with Multics. First, each of the multiple reference detection techniques discovered multiple references that the other did not uncover. In addition, both missed at least one instance of a multiple reference. Tedium accounted for the missed occurances in the cross reference listing technique; an automated version of this method would presumably not suffer from this limitation. In the monitor method, multiple references were missed because some program paths were not taken. Second, even when all multiple references have been uncovered, one must be <u>very</u> conservative in analyzing programs for correctness. Further, when such programs are modified, there is a strong chance that harmless multiple references may lead to serious holes; such programs will need to be audited on each new installation. In many cases this is an extremely tedious task for which people are not well suited. To be entirely sure that a multiple reference is harmless, <u>all</u> paths that a program may take must be traced. Clearly there is a need to develop algorithms which would perform the analysis mechanically.

All of the security holes reported above have been fixed in the current Multics system.

7. Solutions to the Problem.

In the past there have been a number of different reasons for copying arguments. Most of these are characterized by the need to avoid a fault (directed faults: segment, page, no access, ring violation; or indirect address fault: linkage, f1, f3, illegal procedure) while a lock is locked. In May, 1967 a protocol similar to the one described below was detailed in MSPM BD.9.02. The suggestion was made that all arguments to a procedure be copied and that only these copies should be used in the procedure. As various improvements in the system have occurred, some of the reasons for copying arguments have been eliminated and some programmers have ceased to copy arguments. The results of this work show that because of the difficulty in analyzing the effect of multiple references to arguments, <u>all</u> arguments should be copied and validated upon procedure invocation. To be entirely safe, the following pattern of coding should be followed for all ring 0 interfaces:

```
F:  procedure(a_arg1, a_arg2, ... , a_argn);

    copy the values of all input and input/output arguments
    into local variables.

    validate local copies with respect to semantics
    associated with them in this procedure.

        :
        :   use local copies
        :

    set output arguments to values of corresponding
    local variables.

    return

        end;
```

By using this conservative coding style, a procedure can be more strongly isolated from its callers. In effect, we are making a better (by no means perfect) simulation of separate domains by following suitable restrictions in programming style. It should be noted that there are situations where it is difficult to adhere to this style because of efficiency considerations. For example, it would be very inefficient to copy an argument that is a large structure occupying many words of storage. Just as there are syntactic patterns for recognizing bugs in programs, the inverses of these patterns appear to be guides for secure programming.

The general idea of patterns of errors seems to be a powerful tool that can be used in an analysis of a system. In a very short time we have discovered several serious holes in the security of Multics. The success of this error pattern resulted from its simplicity. The main obstacle in discovering other patterns is not so much the nature of the error but rather the suitable simple pattern for which to search. For example, one of the recurring types of errors reported in RFC's 5, 46 and 47 and in the Multics Change Requests is overflowing the capacity of a table. Because of the flexibility of the PL/I language, there are many ways to implement tables. It would be difficult to come up with a general pattern that matched all of these ways because of the many degrees of freedom in the PL/I language. The conclusion is obvious: What we need are more highly structured languages which require a programmer to identify the objects being used (for example the language "CLU" being developed in the Computation Structures Group of Project MAC at MIT). In this way, simple patterns for complex errors can be developed.

Appendix

Classification of Entry Points in hcs_

Of the 170-odd entrypoints in the hardcore gate hcs_, some 50 have multiply referenced arguments which were found by the auditing and online monitoring techniques. We may classify these further into five classes:

1. Those which are probably not security holes. To the best of our knowledge, with the way the system is currently structured, these multiple references do not cause any problems. Of course, we would feel even safer if all arguments were copied and the copies referenced.

2. Multiple references which cause the procedure to be fragile, but which probably do not cause security violations. By fragile, we are trying to dramatize the fact that the multiple references to arguments cause the procedure to be very dependent on the current order in which tasks are carried out. Alterations in the procedure are very likely to upset this delicate balance.

3. Multiple references that have not been explored to the depth necessary to assign them to one of the other classes.

4. Multiple references which look as if they produce holes in the system, but we can't think of a way to exploit the hole.

5. Multiple references which cause holes which we know how to use to penètrate the system.

The following list of entrypoints tells which arguments, if any, are multiply referenced. The notation 'entrypoint (1,3)' means that the first and third arguments of entrypoint are referenced more than once. If any arguments are referenced more than once, remarks are made about which of the above five classes the references belong to.

Summary of Results

A summary of the results obtained in our study is presented in the following table.

Number of entry points examined in hcs_.	170
Number of entry points with multiple references.	51
Classification of multiple references:	
Type 1 -- Probably O.K.	23
Type 2 -- Fragile, but probably O.K.	8
Type 3 -- Don't know, lack of information	3
Type 4 -- Hole without obvious exploitation	8
Type 5 -- Hole with known exploitation	9
Untested entry points	3

Entrypoint -- Args referenced more than once	-- Type, Remarks
accept_alm_obj (1, 2)	1 -- Probably O.K.
acl_add	
acl_addl (3, 5)	1 -- Probably O.K. Arg 3 validated after 2nd reference, arg 5 is an array whose elements are referenced once each.
acl_delete	
acl_list	
acl_replace	
add_acl_entries	
add_dir_acl_entries	
add_dir_inacl_entries (5)	4 -- Hole, without obvious exploitation. Can operate on any ring initial acl, since argument is validated before copying.
add_inacl_entries (5)	4 -- See add_dir_inacl_entries.
append_branch	
append_branchx	
append_link	
appendl	
assign_channel	
assign_linkage (1)	1 -- Probably O.K. This program could run in the user ring.
block	
chname	
chname_file	
chname_seg	
cpu_time_and_paging_	
del_dir_tree	
delentry_file	
delentry_seg	

delete_acl_entries	
delete_channel	
delete_dir_acl_entries	
delete_dir_inacl_entries (5)	4 -- See add_dir_inacl_entries.
delete_inacl_entries (5)	4 -- See add_dir_inacl_entries.
ex_acl_delete	
ex_acl_list	
ex_acl_replace	
fblock (1, 2)	2 -- Fragile, but probably O.K.
fs_get_brackets (3)	1 -- Probably O.K. Array whose elements are referenced once each.
fs_get_call_name	
fs_get_dir_name	
fs_get_mode	
fs_get_path_name	
fs_get_ref_name	
fs_get_seg_ptr	
fs_move_file	
fs_move_seg	
fs_search_get_wdir (1)	1 -- Probably O.K. Referenced twice in copy of pointer using old version 2 pointer copy.
fs_search_set_wdir	
get_alarm_timer	
get_author	
get_bc_author	
get_count_linkage	
get_defname_	
get_dir_ring_brackets (3)	1 -- Probably O.K. Array elements referenced once each.
get_entry_name	
get_initial_ring	
get_ips_mask	
get_link_target (4)	1 -- Probably O.K. Return value, insensitive.
get_linkage (2)	1 -- Probably O.K.
get_lp (1, 2)	1 -- Probably O.K.
get_max_length	
get_max_length_seg	
get_page_trace	
get_process_usage (1)	1 -- Probably O.K.
get_rel_segment	
get_ring_brackets (3)	1 -- Probably O.K. Array elements referenced once each.
get_safety_sw	
get_safety_sw_seg	
get_search_rules	
get_seg_count	
get_segment	
get_usage_values	
get_user_effmode (5)	1 -- Probably O.K.
high_low_seg_count	

initiate	
initiate_count	
initiate_search_rules (7)	1 -- Probably O.K. Twice referenced in copy operation.
initiate_seg	
initiate_seg_count	
ioam_list (1)	3 -- Don't know, haven't looked at it close enough.
ioam_release	
ioam_status	
ipc_init (6)	1 -- Probably O.K. Twice referenced in copy operation.
level_get	
level_set	
link_force	
list_acl (3)	2 -- Fragile, but probably O.K. User can cause fault, but no locks locked.
list_dir	
list_dir_acl (3)	2 -- Fragile, but probably O.K. See list_acl
list_dir_inacl (3)	2 -- Fragile, but probably O.K. See list_acl.
list_inacl (3)	2 -- Fragile, but probably O.K. See list_acl.
make_ptr	
make_seg (1, 2, 5)	2 -- Fragile, but probably O.K. Can cause strange KST state with blank name.
makeunknown	
mask_ips	
pre_page_info	
printer_attach (2)	4 -- Hole without obvious exploitation. Event channel saved in user area, then referenced.
printer_order	Not checked. No listing available.
printer_write_special	Not checked. No listing available.
printer_detach (1)	5 -- Hole. Can cause inconsistent attachment states, since device index is validated, then used.
printer_write (1, 2, 3)	5 -- Hole. Can write on different printer than the one assigned.
proc_info	
quota_get (2)	1 -- Probably O.K.
quota_read	
quota_move	
read_events (1, 2)	1 -- Probably O.K.
replace_acl	
replace_dir_acl	
replace_dir_inacl (6)	4 -- Hole without obvious exploitation. See

		add_dir_inacl_entries.
replace_inacl (6)	4 --	Hole without obvious exploitation. See add_dir_inacl_entries.
reset_ips_mask		
reset_working_set		
rest_of_datmk_		
set_alarm		
set_alarm_timer		
set_automatic_ips_mask		
set_backup_dump_time		
set_backup_times		
set_bc		
set_bc_seg		
set_copysw		
set_cpu_timer		
set_dates		
set_dir_ring_brackets (3)	1 --	Probably O.K. Array elements referenced once each.
set_dtd		
set_ips_mask		
set_lp		
set_max_length		
set_max_length_seg		
set_pl1_machine_mode		
set_safety_sw		
set_safety_sw_seg		
set_ring_brackets (3)	1 --	Probably O.K. See set_dir_ring_brackets.
set_timer		
sfblock (1)	5 --	Hole. Uncopied value used when copied value available!!
star_		
star_list_		
status		
status_ (4, 5)	5 --	Hole. User's argument controls whether lock is locked, and then whether it is unlocked. Can leave lock locked.
status_long (4, 5)	5 --	Hole. See status_.
status_minf		
status_mins		
status_seg_activity		
stop_process (1)	5 --	Hole. Can stop any process. arg used after validation.
tdcm_attach	All	tdcm entries use a segment as argument. It is not clear whether changes to this segment can cause problems.
tdcm_detach		
tdcm_iocall		
tdcm_message (2)	4 --	Hole without obvious exploitation. Can possibly

	cause message inconsistent with system's idea of tape name.
tdcm_promote	
tdcm_reset_signal	
tdcm_set_signal	
tdcm_mount_bit_get (1)	1 -- Probably O.K.
terminate_file	
terminate_name	
terminate_noname	
terminate_seg	
total_cpu_time_	
trace_marker	
truncate_file	
truncate_seg	
try_to_unlock_lock	
tty_abort (2)	3 -- Don't know effect of multiple reference. Not sure whether this is a problem or not.
tty_attach (2, 4, 5)	2 -- Fragile, but probably O.K. Finally copies second argument inside second level call to ioam_. Other args O.K.
tty_detach (3, 4)	1 -- Probably O.K.
tty_detach_new_proc (3, 4)	1 -- Probably O.K.
tty_event (2, 3, 4)	2 -- Fragile, but probably O.K. See tty_attach.
tty_index (4, 5)	5 -- Hole. Code is referenced twice in dn355$get_devx. Could return information which might be sensitive about allowed device id's.
tty_order (2, 3)	3 -- Don't know whether this multiple reference is a hole or not.
tty_read (3, 5, 6)	5 -- Hole. Perhaps hard to exploit.
tty_state	
tty_write (3, 4, 5, 6, 7)	5 -- Hole. Arg 3 referenced in a loop. Can cause the system to appear crashed.
unmask_ips	
unsnap_service (1, 2, 3)	1 -- Probably O.K. This program need not be in ring 0.
usage_values	
virtual_cpu_time_	
wakeup (4)	1 -- Probably O.K.