PROJECT MAC   10 June 1975

Computer Systems Research Division   Request for Comments No.79.

DYNAMIC LINKING AND ENVIRONMENT INITIALIZATION IN A MULTI-DOMAIN PROCESS.

by Philippe A. Janson.

22 May 1975.

Submitted to the ACM
Fifth Symposium on Operating Systems Principles.

Abstract.

In an effort to make the security kernel of the Multics system easier to certify, it was recommended that the dynamic linker be removed from the protection environment of the security kernel. This task prompted a thorough analysis of the function of the dynamic linker. The study revealed that the dynamic linker implemented a function far more complex than inter-procedure linking: it was also exploited to help generate and initialize the workspace for domains and procedures. This implementation of dynamic linking and environment initialization was shown to be incorrect: it violated the least privilege principle as it failed to consider the multi-domain aspect of the Multics processes.

Dynamic linking and environment initialization in a multi-domain process is analysed and decomposed into its elements: dynamic inter-procedure linking, dynamic domain working storage generation and dynamic procedure workspace initialization. In order to respect the least privilege principle, all these components should not be implemented by one single program because this program would have to have the combined access privileges of several domains and could therefore be exploited as an unauthorized information channel between these domains. Instead each component should be implemented by a separate program running in a specific domain. The paper concludes by describing the problems and proposing a method of bootstrapping the elements of the distributed dynamic linking and environment initialization mechanism in a system like Multics.

Keywords and phrases: operating system security, security kernel certification, protection domains, protected subsystems, inter-procedure linking, inter-domain linking, multi-domain process, least privilege principle.

CR Categories: 4.3.

This report is based on a thesis [5] submitted in May 1974
to the Department of Electrical Engineering of M.I.T.
in partial fulfillment of the requirements for the degrees
of Master of Science and Electrical Engineer.
Research reported here was performed
in the Computer Systems Research Division of Project MAC,
an M.I.T. Interdepartmental Laboratory,
and was sponsored by:
the Advanced Research Projects Agency
under ARPA Order No.2095
monitored by ONR Contract No.N00014-70-A-0362-0006;
the Advanced Research Projects Agency and
the Air Force Information Systems Technology Applications Office
under ARPA Order No.2641 monitored by ISTAO;
Honeywell Information Systems, Inc.;
the Harkness Fellowships of the Commonwealth Fund of New York.

Author's address: MIT Project MAC, 545 Technology Square,
Cambridge, Massachusetts 02139.

1. Introduction.

Research reported in this paper was carried on in 1973-1974 in the Computer Systems Research Division of M.I.T. Project MAC. The Division has been concentrating its efforts on a project aimed at producing a certifiably secure operating system for a computing utility. The approach taken was to improve the certifiability of an existing system -the Multics system- rather than to design an entirely new system from scratch. Two tasks were identified as necessary to produce a certifiable system: the definition of a security kernel for that system and the simplification of that security kernel to the point where an individual can easily audit it to establish confidence in its correctness. One of several ways to make a security kernel both better defined and simpler is to make it smaller by removing from it functions which "do not belong there". Removing a function from the security kernel of an operating system simply means guaranteeing that the function is never used by the security kernel itself and preventing the function from ever being executed in the protection environment of the security kernel on behalf of some other protected subsystem. Dynamic inter-procedure linking [2,3,10,11,12] is a typical example of a function which "does not belong in the security kernel" of an operating system. First, the modules of the security kernel can be linked together by some static linkage editor prior to being used; thus the security kernel does not need the help of a dynamic linker to operate. Second, on any single invocation, a dynamic linker operates on data pertinent to only one domain; therefore, it is not part of the "common" mechanism implemented by the security kernel: it does not need the privileges normally granted to security kernel primitives to operate on data pertinent to several domains, during one invocation. Finally, a dynamic linker is a very complex program whose operation is driven by information derived from user code; therefore, it is highly susceptible to malfunction and could cause malfunction of the whole security kernel if it were in the same protection environment.

Removing the dynamic linker from the security kernel of Multics called for a deep analysis of its function to understand all the implications of the new design. In addition to satisfying the above goal, the analysis revealed some most interesting facts about the dynamic linker. It was found that the dynamic linker implemented not just a dynamic linking function, but a more complex function including domain and procedure environment initialization. More importantly, this dynamic linking and environment initialization function was incorrectly implemented. In a multi-domain process, dynamic linking and environment initialization were shown to require access privileges ranging over several protection domains. Granting all these privileges to a single program -as was done for the Multics dynamic linker- violates the least privilege principle: it forces that program to execute in a privileged environment -e.g. the security kernel- where it can potentially be exploited as an unauthorized information channel between the above domains. A correct design of dynamic linking and environment initialization required a much better understanding of the issues at stake in a system supporting multi-domain processes. The results of the analysis of dynamic linking and environment initialization and a discussion of the implications of the multi-domain problem are the topic of this report.

For each outbound reference appearing in the source code of a procedure, language translators generate an item, called a link, associated with the object code of that procedure. The link contains the symbolic name denoted by the outbound reference in the source code. Linking a procedure P to a program module P' means enabling P to reference P' by translating the link between P and P' from its symbolic form (symbolic name of P') to a processor interpretable form (address of P'). <u>Dynamic linking</u> is different from linking in that links are translated one at a time, on demand, rather than all together at load time. This design delays the binding of a procedure to the modules it references until that binding is actually needed. Delayed binding is a feature encountered in the design of many operating systems, and we will mention it again several times in this report. It buys flexibility and avoids having to establish bindings universally and unconditionally, when some of them may be needed only under certain circumstances. Dynamic linking includes allocating/retrieving the address of P' and using it to link P to P'. Dynamic environment initialization denotes the operations necessary to generate and set up the environment of P' when control is transfered from P to P'. This includes allocating/retrieving the workspace for P' and binding it to P'.

Systems have been described in the past which support only one domain per process [6]; in such systems, inter-domain communication is implemented by inter-process messages. In the present report, we consider systems supporting multi-domain processes; in these systems, inter-domain communication is implemented by inter-procedure calls.

Since in a normal sequence of events, a transfer of control from one procedure to another is preceeded, at some point in time, by a demand to link the caller to the callee, it seems very appealing to let the dynamic linker take care of all dynamic linking and environment initialization operations at once instead of just dynamic linking operations. This was the approach taken in the original Multics dynamic linker. It is a wrong approach to take in a multi-domain process system because, in the case of a cross-domain call, it requires the dynamic linker to have more privileges to handle all of dynamic linking and environment initialization than it would need to handle just dynamic linking. This is because dynamic linking concerns only the caller domain while environment initialization concerns the called domain. Thus the compound dynamic linker-environment initializer needs access to objects in both domains and therefore becomes a potential unauthorized information channel between the two domains. This report will show why dynamic linking and environment initialization operations should be distributed and how they can be distributed among several programs executing in different domains so that the security of neither domain is compromised during a cross-domain call.

The next section of the report will analyse dynamic linking and environment initialization in a multi-domain process. Relevant features will be described for a model system to be used as an illustrative example in the report. The action of a procedure P calling a procedure P' across a domain boundary will be decomposed into elementary operations. It will be shown that different sets of elementary operations require different sets of access privileges. Compounding all operations under the responsibility of one program forces that program to have the union of the sets of access privileges and violates the least privilege principle.

The last section of the report will sketch the principles of one possible correct design for dynamic linking and environment initialization in a multi-domain process. The elementary operations composing dynamic linking and environment initialization will be grouped into five sets. Three sets are implemented by three separate programs requiring different access privileges: dynamic working storage generator, dynamic workspace initializer and dynamic linker. Of these, only the dynamic working storage generator cannot be removed from the security kernel. Two more sets of operations are implemented respectively by the hardware call machinery and by the entry sequences inserted by language translators at the entry points of any procedure. The design proposed in the last section of the report has the additional advantage of making inter-domain linking look identical to intra-domain linking from the point of view of the dynamic linker. This was not the case in the original design where the dynamic linker had to distinguish between linking procedures within a domain and across domain boundaries. The design proposed in this report has been demonstrated viable: it has been implemented in Multics. The details and quantitative results of the installation are not discussed in this report: an evaluation of the simplicity and the performance of the new design is given elsewhere [5]. In summary, the size of the Multics security kernel and the complexity of its interface were decreased by ten percent while the performance of the system was not noticeably affected.

Throughout the whole report, the reader is assumed to be familiar with some information protection concepts (e.g. ring, domain, security kernel, least privilege principle [4,8]) as well as with some general design features of Multics (e.g. segmented virtual address space, dynamic linking [1,7]).

2. Problem Statement.

2.1. Framework for Discussion.

Before we can describe the meaning and the implications of dynamic linking and environment initialization in a multi-domain process, we need a short description of relevant features of an illustrative system to be used as an example in the later discussion. This system is very much like Multics but it is simpler and more general. It is simpler in that any program module has only one entry point; thus the symbolic name space for procedure entry points has only one parameter instead of two as in Multics. It is more general in that each process can go into (a finite but large number of) distinct protection domains which are not subject to any ordering of privileges as are the Multics protection rings; in this respect, the model is similar to the system described by Schroeder [9].

Apart from these differences, the system is very similar to Multics as far as dynamic linking and environment initialization are concerned. There is one structured virtual address space per process. The address space of a process is materialized by a descriptor segment. For each segment in the address space of a process, there is one entry in the descriptor segment, called a segment descriptor word (SDW); it contains the physical address of the segment. For each domain where a process can go, there is one entry in each SDW, describing the access privileges of that domain (1) to the segment denoted by the SDW. The current process and domain of a processor are defined by its descriptor base register (DBR) which denotes a descriptor segment, and a domain number to be used as an index into SDW access privilege fields. A security kernel primitive can be invoked to assign a segment number to a segment. When presented with a file system name uniquely identifying a segment, the primitive returns the segment number assigned to the specified segment. If the segment does not yet exist, a segment number is assigned to it, but, according to the delayed binding policy mentioned earlier, the segment and its SDW are created only when actually referenced.

During execution, a procedure requires two kinds of working storage: local storage (2) which is allocated in the _push-down stack segment_ for the current domain of execution, and

---

(1) We assume our Multics-like system supports enough domains per process to host all possible protected subsystems. If there were not such a one to one correspondance between domains and protected subsystems, a domain management primitive would have to be used to allocate/deallocate domains to protected subsystems. This problem is briefly discussed by Schroeder [9]. It is irrelevant to our discussion of dynamic linking and environment initialization.

(2) PL/I equivalent of local and own storage are automatic and internal static storage.

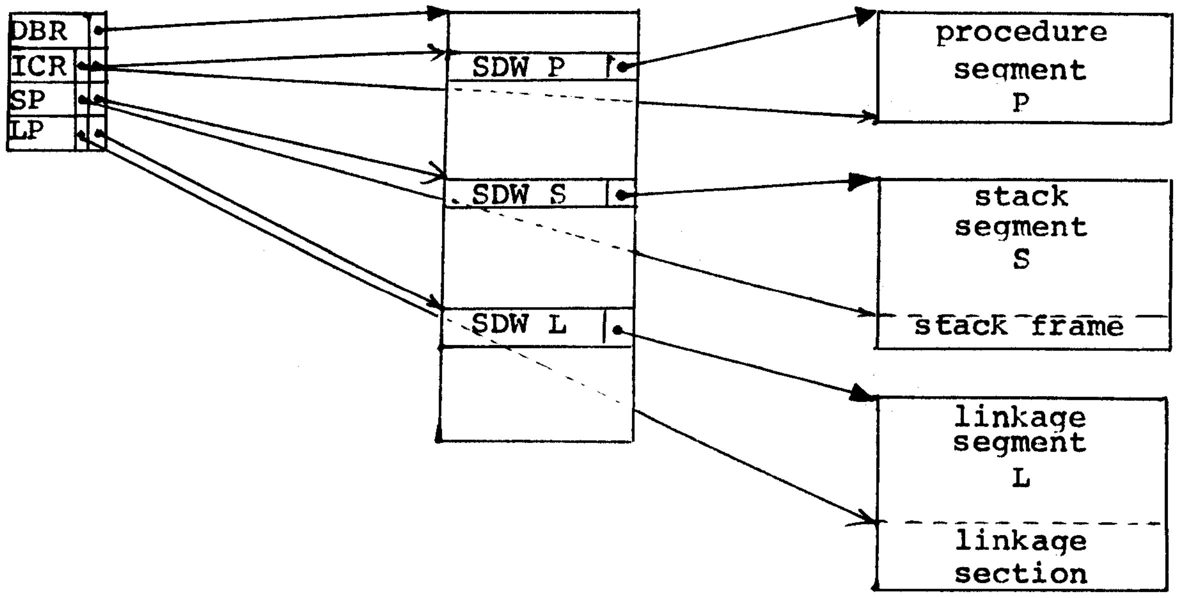
own storage (2) which is allocated in the "own" segment for the current domain of execution.

It is often desirable to have executable code be non-writeable: as a measure of self-protection, this prevents a user from accidentally damaging his procedures; as a measure towards recursion and sharing, it allows direct conflict-free access of all existing invocations of a procedure to the code of that procedure. On the other hand, the links between a procedure and the segments it references must be writeable by the linker which translates them and should not be shared across domains and processes because each process has a different address space and each domain may have a different name space (for mapping symbolic names into file system names). From the previous two remarks, we deduce that in a system in which access control is enforced on a per segment basis, the code and the links for a procedure cannot be stored in the same segment. In our Multics-like system, associated with each procedure is a non-writeable prototype linkage section appended to the procedure code; in that prototype linkage section, each link contains the symbolic name of a module referenced by the procedure; during execution by a given process in a given domain, the prototype linkage section must be copied into a linkage section private to that domain in that process; the symbolic names may then be unambiguously interpreted in the domain name space and translated into processor interpretable addresses based on the process address space. Thus, the scope of the private linkage section of a procedure instance and the scope of the own storage of that procedure instance are identical, viz. one domain in one process. Hence the two kinds of storage can be allocated on the "own" segment -which we therefore rename <u>linkage segment</u>.

Figure 1 helps visualize the system. It summarizes all we need to know about the system to understand dynamic linking and environment initialization. The entire report will be based on it.

Figure 1. Description of the environment of the processor during execution.

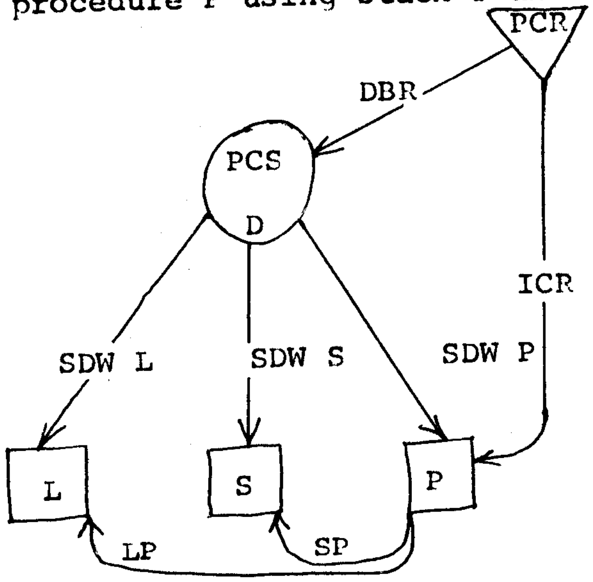
DBR	=	descriptor base register defines current domain and process
ICR	=	instruction counter register defines current instruction and program
SP	=	stack pointer defines current stack frame
LP	=	linkage pointer defines current linkage section



2.2. Analysis of Dynamic Linking and Environment Initialization.

Figure 1 summarized the functional description of the system. Figure 2 is just an abstraction of figure 1 in terms of the bindings required by the system to operate. Processor PCR, dedicated to process PCS, in domain D, executes procedure P using the workspace denoted by SP and LP. All of this is expressed by the bindings of figure 2 as well as by the registers of figure 1.

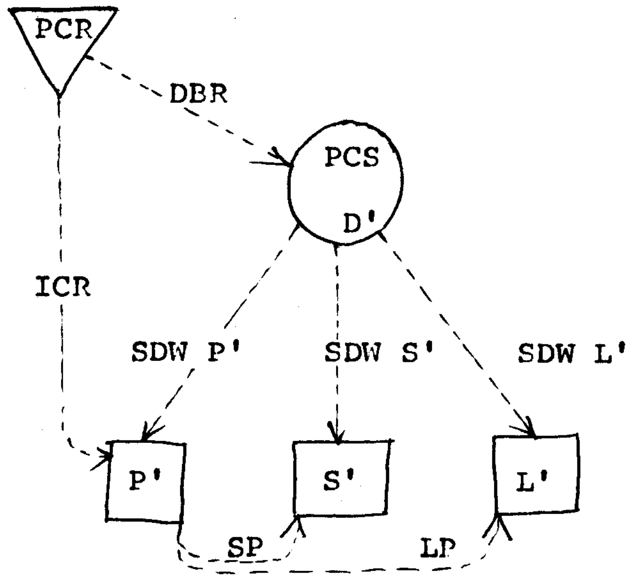
Figure 2.
Processor PCR dedicated to process PCS in domain D executes procedure P using stack S and linkage L.



To analyze dynamic linking and environment initialization in a multi-domain process, we consider a concrete situation. Assume procedure P calls procedure P', P' is a gate into domain D' and this is the first time process PCS goes into domain D', so that nothing is set up for executing in D' yet.

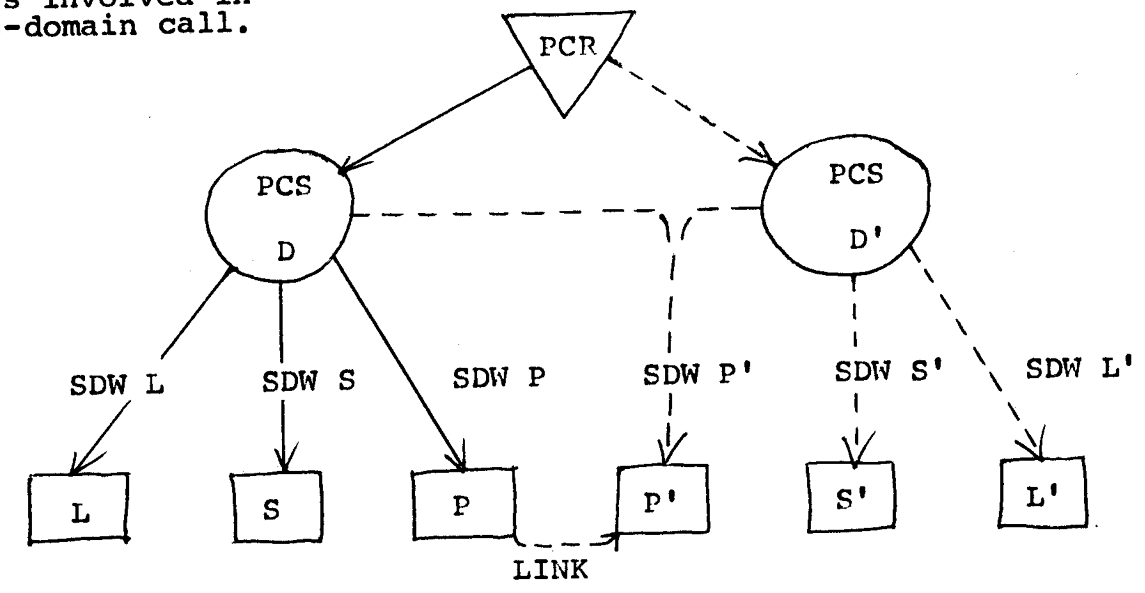
After the call, we will have moved from the system status described in figure 2 to a system status described in figure 3.

Figure 3.
Bindings to be established to support a cross-domain call.



While all bindings represented in figure 2 did exist at the time of the call, the corresponding bindings do not exist in figure 3 at the time of the call because the stack and linkage segments do not exist yet for D' and P' has never been referenced. One more binding not shown in figure 3 is still missing to support the transfer of control: we need a connection to go from figure 2 to figure 3 as expressed in figure 4 by the link required by P to call P'.

Figure 4.
Bindings involved in a cross-domain call.



We propose to examine the missing bindings of figure 4 case by case. The first binding which is required is the link from P to P'. We assume the existence of a dynamic linker whose action is triggered by <u>linkage</u> <u>faults</u>. A linkage fault is a processor exception occuring when a procedure tries to call another procedure via a link which has never been translated previously and still contains a symbolic name. When P calls P', it causes a linkage fault. Performing the translation of the symbolic name of P' into a processor interpretable address is the strict definition of dynamic linking. It is done in three steps: first, using the interpretation rules of the name space of D, the dynamic linker maps the symbolic name of P' into a file system name uniquely identifying P'; second, as explained in the description of address space management in our Multics-like system, a security kernel primitive may be invoked by the dynamic linker to obtain the segment number of P' given its file system name (to bind P' to a segment number); finally, the dynamic linker uses that segment number and a relative offset of zero as the virtual address of the single entry point (3) of P'. That virtual address is then stored in the link in place of the symbolic name of P', and control is restored to the instruction of P which caused the linkage fault. At this point, the LINK and SDW P' of figure 4 are established.

P can now call P' without causing any linkage fault in the current domain of the current process. As suggested by Schroeder [9], the SDW for a gate must define the domain number of the protected subsystem behind the gate. Thus the hardware call machinery will notice P' is a gate into D'. The DBR and ICR bindings of figure 4 will be established accordingly to express that PCR is now dedicated to PCS executing P' in D'.

Four bindings remain to be established: two SDWs must be created for S' and L', and SP and LP must be loaded to bind P' to its workspace.

The case of S' and SP is relatively easy to solve without using extra machinery. Suppose we adopt the following two conventions: first, eventhough the stack segments for all the

(3) In the real Multics, non-zero offsets are permitted, and additional mechanisms are used to determine their value.

domains where a process can go do not exist when the process is created, we reserve a given segment number for the stack segment of each given domain; second, in any stack segment, we reserve the first word to save the address of the next available stack frame on the stack segment. The cross-domain call machinery may be wired to then load SP with the conventional segment number S' corresponding to the stack segment of D'. And a suitable entry sequence into P' can further obtain the offset of the next stack frame from (S',0). According to the earlier description of the management of the descriptor segment in our Multics-like system, referencing S' when SDW S' is reserved will cause S' to be created and SDW S' to be filled to describe access to S'. The security kernel may distinguish stack segments from other segments and initialize their first word appropriately upon creation. While this mechanism may seem ad hoc, we will see its actual significance later. Thus establishing SDW S' and SP can be done conveniently with the existing machinery and two conventions.

Unfortunately, the same is not true for L' and LP. If it were just a matter of creating L' and finding empty space on L', a method similar to the one used for S' and SP would be adequate. However, more operations are required. The scope and lifetime of a stack frame is one invocation of one procedure in one domain; thus a new stack frame must be pushed on each procedure call. But the scope and lifetime of a linkage section is all invocations of one procedure in one domain; thus a linkage section must be allocated on L' only the first time P' is invoked and its address must be remembered for all subsequent invocations of P' in D'. In addition, the newly allocated linkage section must be initialized to the prototype linkage section from P' so that each symbolic link from the shared prototype linkage section appears in the unshared local linkage section and can later be translated locally into a meaningful address. The cross-domain call machinery and the entry sequence of P' are not sufficient tools for our needs as they cannot distinguish between the first and subsequent invocations of P'. A program knowing about P' and L' is required to initialize the linkage section of P' and save its address for later use.

2.3. Unsafe design.

This is the very part of the problem which was mishandled in the original implementation in Multics. The creation of L' as well as the initialization of the linkage section of P' were left for the linker to handle ahead of time, when it is invoked to establish the link between P and P'. It was conjectured that establishing a link between P and P' would always be followed by calling P'. Thus the dynamic linker also checked for the existence of L' and the linkage section of P' in D', and eventually created them. This task required the dynamic linker to have access to D'.

In trying to remove the dynamic linker from the security kernel of Multics, it became obvious that the dynamic linker would no longer have access to D', and in fact that it should never have had access to D' in the first place because this violated the least privilege principle and eventually created exploitable flaws in the protection mechanism. The original Multics dynamic linker had unnecessary privileges and unintentionally misused them so that at least two methods existed to deceive the protection of D'. First, without ever intending to call into D', P could reference links to each gate into D' and trigger the linker to initialize the linkage of all these gates into D'. The fact that an action of P in D could cause something to happen in D' without control is unacceptable and particularily dangerous if there is any chance that P might cause L' to be overflowed by too many linkage sections. Second, without ever triggering the linker, P could find out the segment number of P' by invoking the suitable security kernel primitive, and call P' directly using that segment number instead of a symbolic name. As a result, P' would start executing without its workspace being initialized since the dynamic linker was by-passed by P. This is even more unacceptable than the previous trouble as a process crash and perhaps some damage could be caused in D' because of the action of P in D: any initial own variable P' depends on or outbound reference it makes would almost certainly be handled incorrectly.

3. Implementation Proposal.

3.1. Basics of the Design.

The previous section presented a description of dynamic linking and environment initialization in a multi-domain process in a Multics-like system. Some mistakes in an early design of dynamic linking and environment initialization were pointed out. This section will abstract the operations described earlier from their Multics context and propose a correct design for implementing dynamic linking and environment initialization in a multi-domain process.

First, consider the link between P and P' in figure 4. The link defines a <u>binding between the caller and the callee</u>. <u>It must be established by the dynamic linker</u>. This is the only task assigned to the dynamic linker. It does require access only to domain D and to a security kernel primitive. When the linkage fault occurs, the linkage section of P and most information needed to translate the link are readily available in D. A security kernel primitive can be invoked from D to bind P' to a segment number. Consequently, the dynamic linker can and should execute strictly in D, and may be totally unaware of the fact it is translating a link to different domain. Dynamic linking is one of the three functions which need to be supported by an appropriate program running in a specific protection environment, the faulting domain or caller domain in this case. The creation of the link includes assigning a segment number to P' and thus yields the creation of SDW P', as explained in the description of the address space management of our model system.

Second, consider the DBR and ICR bindings. These two bindings are _defining respectively the protection and execution environments_ of PCR. In any system, setting these bindings _is the task of the environment switching machinery_. ICR is loaded by the "jump" instruction machinery; DBR defines the protection environment and must be set by the cross-domain call machinery as it is a protected register.

Third, consider the case of the SDWs. After a new protection environment D' is entered (as expressed by a change in DBR), the environment needs a _binding to some working storage_ out of which each procedure will subsequently be able to carve its own workspace. (This is what SDW S' and SDW L' express in a Multics-like system.) Allocating working storage in the address space accessible from D' _requires the help of the security kernel_ because one or more objects must be created and added to the address space, and because D' cannot create working storage for itself since no program can run in D' without its workspace. In a system in which dynamic environment initialization is a design feature, working storage should be created dynamically, i.e. when it is first referenced. To enable any process entering D' to reference the working storage allocated to that domain, we proceed as suggested earlier, by adopting the convention that the working storage of a domain be assigned reserved virtual addresses (segment numbers) which are left by the cross-domain call machinery in dedicated registers (SP and LP) when the domain is entered. (Notice that this design departs from the original Multics design: L' is now created on demand by the security kernel and not ahead of time by the dynamic linker.) Working storage creation is the second function requiring an appropriate program running in a specific protection domain, the security kernel. Notice that the above design required the cooperation of the call machinery as well.

Fourthly, when working storage is created for D', the executing procedure P' has to be associated with the workspace allocated to it in the working storage of D'. (This is expressed by the SP and LP bindings in our Multics-like system.) The retrieval task is the responsibility of the entry sequence of the procedure. Again, as suggested earlier, conventions are needed to make the retrieval possible. (For instance, in a Multics-like system, word 0 of each stack segment should be reserved to save the relative offset of the next available stack frame on the stack segment. Similarily, in each linkage segment, there might be a linkage offset table, located at the base of the segment, which for each segment in the address space gives the relative offset of the corresponding private linkage section - if any.

Thus the entry sequence of procedure P' can load SP from (S',0) and LP from (L',P'), where S', L' and P' are the segment numbers left in SP, LP and ICR after the cross-domain call.) The reader may have noticed that one problem was swept under the rug: in a Multics-like system, the prototype linkage section of P' must be copied into the private linkage section allocated to P' in L' prior to letting P' load LP; this is an example of the general problem of initializing the workspace of a procedure prior to the first invocation of that procedure.

Initialization of the workspace of a procedure was supported by the dynamic linker in Multics. Because it concerns stictly domain D', we declare that it _must be performed from within D'_, on demand, by a dynamic workspace initializer. (In a Multics-like system, we propose that the linkage offset table be initialized with zeroes when L' is created. Thus loading LP with a null linkage offset can be recognized as an _own storage fault_ whereupon the dynamic workspace initializer must be invoked. After the missing linkage section is copied and its relative offset is saved, the faulting procedure P' can successfully load LP.) In general, a processor exception is needed to identify the need of a procedure entry sequence to retrieve its workspace when it does not yet exist. Dynamic workspace initialization is the last function requiring a program executing in a specific protection domain, the faulting domain or called domain in a cross-domain call.

3.2. Summary of the Design.

What the above recommendations have achieved is a total separation of the components of dynamic linking and environment initialization, and particularily of the three main components requiring software support: dynamic linking, dynamic working storage generation and dynamic workspace initialization. Dynamic linking deals exclusively with linkage faults; it is supported by the dynamic linker which operates in the domain where a link must be translated. Dynamic working storage generation handles the bindings associating a protection environment to its working storage; this has to be supported by the security kernel as it requires creation of new objects. Dynamic workspace initialization deals with bindings defining the workspace of a procedure; it requires the help of a dynamic workspace initializer operating in the domain where initialization is required.

3.3. Side Effects of the Design.

The last problem which had to be dealt with in the new design is the initialization of the distributed dynamic linking and environment initialization mechanism. As long as the dynamic linker and the dynamic workspace initializer were one single program executing in the security kernel domain, they were initialized as parts of the kernel itself, i.e. some system generation mechanism or bootstrapping procedure was used for the entire security kernel and dynamic binding was no special case. Now that the dynamic linker and the dynamic workspace initializer have been removed from the security kernel and may be invoked at any time in any domain, both programs must be made operational in any domain as soon as that domain becomes used. For the programs to be operational in a domain, they have to have a private linkage section in that domain and all links in the linkage section must be translated prior to operation of the programs, because neither program can count on the other to bootstrap it, short of going into a recursive initialization problem.

First, notice that the dynamic linker and the dynamic workspace initializer are likely to contain symbolic references (calls) to security kernel primitives (e.g. to bind a segment to a segment number) but they will never contain symbolic references to user programs. Second, since security kernel modules and dynamic linking and environment initialization modules are vital to any process, they must be mapped into the virtual address space of any process; thus, they can be assigned a fixed set of addresses in the address space of any process. Because of the above two remarks, we can assert that, once translated, the links in the private linkage sections of the dynamic linker and the dynamic workspace initializer could be identical in all domains of all processes. Thus all it takes to make the dynamic linker and the dynamic workspace initializer operational is to submit them to a static linkage editor after system initialization but prior to system operation and to have the linkage editor produce a template translated linkage section for each program. Then the security kernel can either make the template translated linkage sections public, read-only, or copy them into each linkage segment it creates. If the latter approach is chosen, the operation can perfectly well be performed at the time a linkage segment is created (SDW L') as the creation of a linkage segment coincides exactly in time with the earliest moment when the dynamic linker and the dynamic workspace initializer may be needed in a new domain. This again requires the security kernel to distinguish the creation of linkage segments from the creation of any other segment, as was the case for stack segments. Linkage and stack segment creation are not just ad hoc mechanisms but should be regarded as the two operations implementing domain generation. Such an operation must necessarily be carried on by the security kernel.

4. Conclusion.

This report has presented the conclusions of a study of dynamic linking and environment initialization in a multi-domain process. Dynamic linking and environment initialization was first decomposed into elementary operations. Elementary operations were then grouped into five sets of operations each implementing a different aspect of dynamic linking and environment initialization. In particular, three aspects require the help of software programs. It was shown that each of these three aspects should be implemented by a separate program executing in a specific protection domain to preserve the integrity of the two domains involved on a cross-domain call. Dynamic linking is implemented by the dynamic linker operating in the faulting domain (caller domain of a cross-domain call), dynamic working storage generation is implemented in the security kernel, and dynamic workspace initialization is implemented by the dynamic workspace initializer operating in the faulting domain (called domain in a cross-domain call). This distribution of functions insures a safe design of dynamic linking and environment initialization: programs implementing the functions cannot be exploited as unauthorized information channels between two domains as they never execute in more than one domain during any single invocation.

Acknowledgements.

Many people have contributed ideas to the research described in this report and many people have commented on the successive versions of the report itself. Professor M.D. Schroeder deserves special gratitude for his guidance, his encouragements, his dedication and his interest in the research project, and for his pertinent comments on the report. Professor J. H. Saltzer has made very detailed remarks and highly relevant suggestions on the later versions of the report. The helpful contributions of Dr. D.D. Clark were appreciated in the early phases of the project. Thanks are due B.S. Greenberg, R.K. Kanodia and E.L. Thomas for the ideas they provided during the design of the dynamic binder and for the many hours they spent to help implement it.

References.

1. ---
   Introduction to Multics.
   MAC TR-123 - MIT Project MAC - 1974.

2. Donovan J.J.
   Systems Programming.
   P166 - McGraw Hill - 1972.

3. Fabry R.S.
   Capability-Based Addressing.
   P403 - CACM 17 7 - 1974.

4. Graham R.M.
   Protection in an Information Processing Utility.
   P365 - CACM 11 5 - 1968.

5. Janson P.A.
   Removing the Dynamic Linker from the Security Kernel
   of a Computing Utility.
   MAC TR-132 - MIT Project MAC - 1974.

6. Lampson B.W.
   Protection.
   P437 - Proc. 5th Princeton Conf. on Info. Sc. and Syst. - 1971.

7. Organick E.I.
   The Multics System: an Examination of its Structure.
   MIT Press, Cambridge, Mass. - 1972.

8. Saltzer J.H., Schroeder M.D.
   The Protection of Information in Computer Systems.
   to be published in Proc. IEEE.

9. Schroeder M.D.
   Cooperation of Mutually Suspicious Subsystems
   in a Computing Utility.
   MAC TR-104 - MIT Project MAC - 1972.

10. Shaw A.C.
    The Logical Design of Operating Systems.
    P158 - Prentice Hall - 1974.

11. Tsichritzis D.C., Bernstein P.A.
    Operating Systems.
    P91 - Academic Press - 1974.

12. Watson R.W.
    Timesharing System Design Concepts.
    P68 - McGraw Hill - 1970.