Operating Systems

C. Weissman
Editor

A Note on the Confinement Problem

Butler W. Lampson
Xerox Palo Alto Research Center

This note explores the problem of confining a program during its execution so that it cannot transmit information to any other program except its caller. A set of examples attempts to stake out the boundaries of the problem. Necessary conditions for a solution are stated and informally justified.

Key Words and Phrases: protection, confinement, proprietary program, privacy, security, leakage of data

CR Categories: 2.11, 4.30

Introduction

Designers of protection systems are usually preoccupied with the need to safeguard data from unauthorized access or modification, or programs from unauthorized execution. It is known how to solve these problems well enough so that a program can create a controlled environment within which another, possibly untrustworthy program, can be run safely [1, 2]. Adopting terminology appropriate for our particular case, we will call the first program a *customer* and the second a *service*.

The customer will want to ensure that the service cannot access (i.e. read or modify) any of his data except those items to which he explicitly grants access. If he is cautious, he will only grant access to items which are needed as input or output for the service program. In general it is also necessary to provide for smooth transfers of control, and to handle error conditions. Furthermore, the service must be protected from intrusion by the customer, since the service may be a proprietary program or may have its own private data. These things, while interesting, will not concern us here.

Even when all unauthorized access has been prevented, there remain two ways in which the customer may be injured by the service: (1) it may not perform as advertised; or (2) it may leak, i.e. transmit to its owner the input data which the customer gives it. The former problem does not seem to have any general technical solution short of program certification. It does, however, have the property that the dissatisfied customer is left with evidence, in the form of incorrect outputs from the service, which he can use to support his claim for restitution. If, on the other hand, the service leaks data which the customer regards as confidential, there will generally be no indication that the security of the data has been compromised.

There is, however, some hope for technical safeguards which will prevent such leakages. We will call the problem of constraining a service in this way the *confinement* problem. The purpose of this note is to characterize the problem more precisely and to describe methods for blocking some of the subtle paths by which data can escape from confinement.

The Problem

We want to be able to confine an arbitrary program. This does not mean that any program which works when free will still work under confinement, but that any program, if confined, will be unable to leak data. A misbehaving program may well be trapped as a result of an attempt to escape.

A list of possible leaks may help to create some intuition in preparation for a more abstract description of confinement rules.

0. If the service has memory, it can collect data, wait for its owner to call it, and then return the data to him.
1. The service may write into a permanent file in its owner's directory. The owner can then come around at his leisure and collect the data.
2. The service may create a temporary file (in itself a legitimate action which cannot be forbidden without imposing an unreasonable constraint on the computing which a service can do) and grant its owner access to this file. If he tests for its existence at suitable intervals, he can read out the data before the service completes its work and the file is destroyed.
3. The service may send a message to a process controlled by its owner, using the system's interprocess communication facility.
4. More subtly, the information may be encoded in the bill rendered for the service, since its owner must get a copy. If the form of bills is suitably restricted, the amount of information which can be transmitted in this way can be reduced to a few bits or tens of bits. Reducing it to zero, however, requires more far-reaching measures.

Author's address: Xerox Palo Alto Research Center, 3180 Porter Drive, Palo Alto, CA 94304.

If the owner of the service pays for resources consumed by the service, information can also be encoded in the amount of time used or whatever. This can be avoided if the customer pays for resources.

5. If the system has interlocks which prevent files from being open for writing and reading at the same time, the service can leak data if it is merely allowed to read files which can be written by its owner. The interlocks allow a file to simulate a shared Boolean variable which one program can set and the other can test. Given a procedure open (file, error) which does **goto** error if the file is already open, the following procedures will perform this simulation:

```
procedure settrue (file); begin loop 1: open (file, loop 1) end;
procedure setfalse (file); begin close (file) end;
Boolean procedure value (file); begin value := true;
  open (file, loop 2); value := false; close (file); loop 2: end
```

Using these procedures and three files called data, sendclock, and receiveclock, a service can send a stream of bits to another concurrently running program. Referencing the files as though they were variables of this rather odd kind, then, we can describe the sequence of events for transmitting a single bit:

```
sender:   data := bit being sent; sendclock := true
receiver: wait for sendclock = true; received bit := data;
            receive clock := true;
sender:   wait for receive clock = true; sendclock := false;
receiver: wait for sendclock = false; receiveclock := false;
sender:   wait for receiveclock = false;
```

6. By varying its ratio of computing to input/output or its paging rate, the service can transmit information which a concurrently running process can receive by observing the performance of the system. The communication channel thus established is a noisy one, but the techniques of information theory can be used to devise an encoding which will allow the information to get through reliably no matter how small the effects of the service on system performance are, provided they are not zero. The data rate of this channel may be very low, of course.

Confinement Rules

We begin by observing that a confined program must be memoryless. In other words, it must not be able to preserve information within itself from one call to another. Most existing systems make it quite easy to enforce this restriction. In ALGOL, for example, a procedure which has no **own** variables and which references no global variables will be memoryless.

Taking this for granted, it is easy to state a rule which is sufficient to ensure confinement.

Total isolation: A confined program shall make no calls on any other program.

By a "call" we mean any transfer of control which is caused by the confined program. Unfortunately, this rule is quite impractical. Example 5 above shows that supervisor calls must be forbidden, since quite innocuous looking ones can result in leaks. Example 6 shows that even the implicit supervisor calls resulting from input/output, paging or time-slicing can cause trouble.

To improve on this situation, we must make a distinction between programs which are confined and those which are not. Recall that being confined is not an intrinsic property of a program but a constraint which can be imposed on it when it runs. Hence if every program called by a confined program is also confined, there can be no leakage. This is still not good enough, since the supervisor, for example, cannot be confined. It is at least conceivable, however, that it is *trusted,* i.e. that the customer believes it will not leak his data or help any confined program which calls it to do so. Granting this, we can formulate a new confinement rule.

Transitivity: If a confined program calls another program which is not trusted, the called program must also be confined.

Examples 5 and 6 show that it is hard to write a trustworthy supervisor, since some of the paths by which information can leak out from a supervisor are quite subtle and obscure. The remainder of this note argues that it is possible.

A trustworthy program must guard against any possible leakage of data. In the case of a supervisor, the number of possible channels for such leakage is surprisingly large, but certainly not infinite. It is necessary to enumerate them all and then to block each one. There is not likely to be any rigorous way of identifying every channel in any system of even moderate complexity. The six examples above, however, were chosen to illustrate the full range of possibilities which I have been able to think of, and for the present plausibility argument they will be regarded as a sufficient enumeration. The channels used there fall into three categories:

Storage of various kinds maintained by the supervisor which can be written by the service and read by an unconfined program, either shortly after it is written or at some later time.

Legitimate channels used by the confined service, such as the bill.

Covert channels, i.e. those not intended for information transfer at all, such as the service program's effect on the system load.

All the examples except 4 and 6 use storage channels. Fairly straightforward techniques can be used by the supervisor to prevent the misuse of storage as an escape route for a confined program. (For instance, example 5 can be taken care of by making a copy of a file which is being read by a confined program if someone else tries to write it; the confined program can then continue to read the copy, and the writer can proceed without receiving any indication that the file was being read.) The main difficulty, as example 5 illustrates, is to identify all the kinds of storage which the supervisor implements. This class of channels will not be considered further.

The following simple principle is sufficient to block all legitimate and covert channels.

Masking: A program to be confined must allow its caller to determine all its inputs into legitimate and covert channels. We say that the channels are *masked* by the caller.

At first sight it seems absurd to allow the customer to determine the bill, but since the service has the right to reject the call, this scheme is an exact model of the purchase order system used for industrial procurement. Normally the vendor of the service will publish specifications from which the customer can compute the bill, and this computation might even be done automatically from an algorithmic specification by a trusted intermediary.

In the case of the covert channels one further point must be made.

Enforcement: The supervisor must ensure that a confined program's input to covert channels conforms to the caller's specifications.

This may require slowing the program down, generating spurious disk references, or whatever, but it is conceptually straightforward.

The cost of enforcement may be high. A cheaper alternative (if the customer is willing to accept some amount of leakage) is to bound the capacity of the covert channels.

Summary

From consideration of a number of examples, we have proposed a classification of the ways in which a service program can transmit information to its owner about the customer who called it. This leakage can happen through a call on a program with memory, through misuse of storage facilities provided by the supervisor, or through channels intended for other uses onto which the information is encoded. Some simple principles, which it might be feasible to implement, can be used to block these paths.

Acknowledgments. Examples 5 and 6 are due to A.G. Fraser of Bell Laboratories.

Received July 1972; revised January 1973

References

1. Lampson, B.W. Dynamic protection structures. Proc. AFIPS 1969 FJCC, Vol. 35, AFIPS Press, Montvale, N.J., pp. 27–38.
2. Schroeder, M.D., and Saltzer, J.H. A Hardware Architecture for implementing protection rings. *Comm. ACM 15*, 3 (Mar. 1972), 157–170.

Operating Systems

C. Weissman
Editor

A Class of Dynamic Memory Allocation Algorithms

Daniel S. Hirschberg
Princeton University

A new dynamic memory allocation algorithm, the Fibonacci system, is introduced. This algorithm is similar to, but seems to have certain advantages over, the "buddy" system. A generalization is mentioned which includes both of these systems as special cases.

Key Words and Phrases: dynamic storage allocation, buddy system, simulation, Fibonacci, fragmentation

CR Categories: 3.89, 4.32, 4.39

Introduction

For many applications, there is a need for dynamically reserving (and releasing) variable-size blocks of contiguous memory cells. Several algorithms have been formulated and compared [3, 4]. The buddy system, introduced by Knowlton [1, 2], is preferred to other algorithms such as first-fit and best-fit on the basis of simulations conducted [3, 4].

One scheme for memory allocation transforms storage area requests (which can ask for any integral number of memory cells up to a maximum of *maxreq*) into block requests, where the number of permissible

This work was supported in part by NSF Grants GJ-965 and GJ-30126 and a National Science Foundation Graduate Fellowship. Author's address: Department of Electrical Engineering, Princeton University, Princeton, NJ 08540.