A Resource Allocation Model for Denial of Service

Jonathan K. Millen
The MITRE Corporation
Bedford, MA

Abstract

A denial-of-service protection base is characterized as a resource monitor closely related to a TCB, supporting a waiting-time policy for benign processes. Resource monitor algorithms and policies can be stated in the context of a state- transition model.

1 Introduction

1.1 Background

The three traditional concerns of computer security are confidentiality, integrity, and denial of service. Confidentiality and integrity have been addressed with fair success by designing operating systems that enforce various access-control models. It seems reasonable to expect that an operating system could also help to prevent denial of service, and that an important step in achieving that goal is to propose a denial-of-service model. The model proposed in this paper is the successor to a model presented in [Mill'92]. The present model is simpler.

A computer security model usually consists of a system model, often an abstract state-transition machine, plus a security policy stated in the context of that sort of machine. The policy is driven partly by considerations of realism in the system model, and partly by higher-level requirements. The model to be given below is meant to support a family of related policies rather than one specific policy, but even so it represents a narrow slice of the spectrum of conceivable denial-of-service policies.

1.1.1 Availability Requirements

There have been a number of attempts to formulate denial-of-service protection requirements. The term "availability" is sometimes used for requirements aimed at denial-of-service protection.

Universally applicable requirements might not be possible. At the 1985 DOD Workshop on Network Security [DoDNW], the working group on denial of service concluded that: "No generic denial of service conditions could be identified which were independent of mission objectives."

The workshop did, however, suggest "detection, recovery, and resistance" as the major categories of requirements.

The mere presence of access control, for confidentiality and integrity protection, is a kind of denial of service. Any availability policy must be qualified by the constraints of the access control policy. Dobson [Dobs'91] views the tradeoff between confidentiality and access to information in terms of a negotiation, possessing some rather subtle aspects, and leading to a contract for a combined policy.

The European "Harmonized Criteria" [ITSEC] went beyond TCSEC [TCSEC] requirements by including a functionality class F7 with a terse "continuity-of-service" requirement:

Irrespective of its load at any time, the system shall be able to guarantee a maximum response time for certain specified actions. In addition, for certain specified actions, it shall be guaranteed that the system will not be subject to deadlock.

A workshop was held in Canada in 1990 to crystallize the issues and guide development of effective requirements [CTCPEC]. One of the recommendations of this workshop was to focus on denial of service, which was defined as the loss of availability due to accidental or malicious user actions, as opposed to random failures that impact functionality, which are the concern of reliability engineering. Another recommendation was to develop a model. Two rudimentary models were suggested; a general availability model that added a "Services" layer around a TCB to control access time, and a state-transition model that showed detection and recovery transitions between initial, failed, and partially recovered states.

A more detailed conceptual model was given by Bacic and Kuchta [BaKu'91].

Their paper recognized that the central problem of denial-of-service protection was resource allocation, and that a "resource allocation monitor" had to have the reference monitor characteristics of being tamper-proof, always invoked, and subject to analysis and testing. That paper also reviewed the literature for prior models, among them the Yu-Gligor model.

1.1.2 The Yu-Gligor Model

Gligor had characterized the denial-of-service problem in 1983 as how to provide a shared service with a specified maximum waiting time, despite competition between groups of users [Glig'83]. The paper contains a number of detailed, instructive examples of denial-of-service situations. Subsequently, joint work by Yu and Gligor [YuGl'90] resulted in a deeply developed approach which introduced the following ideas:

- A 'finite waiting time' policy expressed as a liveness condition
- The notion of user agreement
- Fairness and simultaneity policies
- Service specifications
- A general resource allocator model.

The finite-waiting-time policy says that whenever service has been requested, it will eventually be provided. In the context of a service specification, which is a general framework for organizing the description of a service interface, a finite-waiting-time policy is implemented through a combination of user agreements, fairness, and simultaneity policies. The resource allocator model is an example of a service specification. It has Acquire and Release operations, and the required properties and policies are stated with temporal logic formulas.

1.1.3 User Agreements

User agreements are constraints on the behavior of service users; they must be obeyed in order to prevent denial of service. If one views a service as an abstract machine with inputs and outputs, user agreements are input

constraints. They exclude certain inputs in certain states, or disallow input sequences, that are otherwise legal. An example mentioned by Yu and Gligor is an "ordered resource acquisition" constraint for preventing deadlock.

In a denial-of-service protection context, user agreements must be enforced or supplemented by trusted code, since some users may be malicious. Yu and Gligor suggest compile-time checks on user code or a layer of code to filter requests.

1.2 Approach

The model presented in this paper resembles the Yu-Gligor resource allocator. The principal difference is that it represents the passage of time explicitly. By doing so, a maximum-waiting-time policy can be expressed as easily as a finite-waiting-time policy, and it can also support other policies of a probabilistic nature. Policies and user agreements are expressed more explicitly, without temporal logic.

1.2.1 Attacks

There is an important difference between a system that enforces a denial of service policy and one that does not: the system guarantees to maintain certain specified service in the face of deliberate attack.

We will have to circumscribe the kinds of attack that are addressed, because there is no hope of addressing them all within a single conceptual and technical framework. Maintaining network connectivity despite physical destruction of switching nodes, for example, is a very different problem from maintaining service despite attempts to inject false control messages. In a computer security context, it makes sense to focus on attacks that can be carried out by untrusted programs.

1.2.2 Denial-of-Service Protection Base

The scope of protection is also limited by the means employed to enforce it. In a computer security context, denial-of- service protection is accomplished through trusted hardware and software. Our approach here is to try a variation of the reference monitor concept. Let us define a Denial-of-

service Protection Base (DPB) as a hardware/software mechanism with the following three properties:

1. It is tamperproof.
2. It cannot be prevented from operating.
3. It guarantees (authorized) access to resources under its control.

If a DPB is possible at all, it can exist in a computer system that does not have a Trusted Computing Base (TCB; see [TCSEC]) for secrecy and integrity protection. On the other hand, if there is a TCB for a particular system, a DPB could be combined with it, to take advantage of the TCB's mechanism for protecting itself and other data from unauthorized modification. A DPB is also subject to access controls imposed by a TCB. The DPB cannot and need not provide unauthorized access.

If there is both a TCB and a DPB on the same system, it is natural to ask what the structural relationship is between them. Are they co-extensive, is one a subset of the other, can one be implemented as a layer on top of the other, etc. One conclusion follows from the fact that the TCB maintains ultimate control over access to most, if not all, essential system resources. A DPB cannot guarantee access to those resources without the cooperation of the TCB. Any guarantee of service by the DPB is made not only on its own behalf but also on behalf of the TCB. In that sense, the TCB must be regarded as part of the DPB. The TCB might have to be reexamined and even redesigned in order to ensure that it supports service guarantees.

The scope of DPB protection is alluded to in property (3), "resources under its control." It is obvious that the only services whose availability can be guaranteed by a DPB are those provided by the DPB itself. *Hence the DPB must offer those services whose loss would be viewed as a denial-of-service problem.*

1.2.3 Resource Allocation

The relationship between "services" and "resources" is, for our purposes, that a service furnishes access to a resource. The scheduler in an operating system, for example, is a service furnishing access to the CPU resource. The access control mechanism is a service furnishing access to a data resource such as a file or segment.

It is agreed by all authors who have addressed denial of service that long delays in service constitute denial, and absolute denial can be viewed as an infinite delay. Hence, one necessary aspect of denial-of-service protection is the ability to limit waiting times for access to resources. This implies, in particular, that a process cannot be allowed to maintain exclusive access to a resource forever, if another process has requested it. *The DPB must therefore be able to revoke access to a resource.*

One can distinguish between shared and private resources. It is reasonable to permit processes to hold some quota of certain resources forever, if there is enough to go around. More generally, one can set up a maximum holding time policy that is an arbitrary function of the current resource allocations. The maximum holding time for the last few remaining units of a resource might, for example, be much less than for the first few. Similarly, there are time- slicing algorithms that lengthen time slices when the system load is low and shorten them when the load is high.

1.2.4 Resource Destruction

When studying a resource allocation model for denial of service, there is an implicit assumption that resources can be denied to one process only due to allocation of that resource to another process. But there are examples of denial of service whereby a malicious process makes a resource unavailable by removing it from the pool, without acquiring the resource itself. Gligor [Glig'83] gives an example due to Saltzer in which certain directory pages could be rendered inaccessible. Thus, we must distinguish between two kinds of denial of service:

1. denial through resource allocation
2. denial through resource destruction.

Resource destruction depends on the existence of some design or implementation mistake in the DPB, since there is no legitimate excuse to allow user programs to destroy resources that are presumably under DPB control.

This paper focusses on resource allocation and does not attempt to model or suggest countermeasures for resource destruction problems.

1.2.5 Progress, Simultaneity and Deadlock

Deadlock is often brought up in discussions about denial of service. Deadlock is defined to be a condition in which two or more processes have acquired certain resources, but each one needs access to at least one additional resource (without giving up access to the resource it has) in order to make progress. Deadlock occurs when each of the additional resources needed has already been acquired by some other process.

Deadlock, as such, is somewhat off-center as an aim for denial-of-service protection, because none of the processes involved is necessarily malicious. If a process is malicious, it need not be deadlocked itself; it can block other processes simply by acquiring resources they need. Consideration of deadlock, however, reminds us of the fact that processes may fail to make progress because they need simultaneous access to two or more resources. *The DPB policy should ensure that a process will eventually gain simultaneous access to all resources it needs simultaneously.* If this design objective is satisfied, deadlock will, perforce, be prevented.

For many purposes, a process needs to maintain exclusive access – a lock – on some resource over a number of successive time slices, until some logically unitary task is completed. *A DPB should respect a request to maintain exclusive access to a resource, as long as it is for a reasonable length of time.* A permanent lock would, of course, defeat shared access and deny service.

1.2.6 Waiting Time Policies

The early work of Gligor suggested a Maximum Waiting Time (MWT) policy, in which a requested service is provided within some fixed time bound. The Yu-Gligor paper introduced the weaker Finite Waiting Time (FWT) policy, in which the service will eventually be provided, but there is no fixed upper bound to the amount of time the process might have to wait.

Another category of policies should be considered: Probabilistic Waiting Time (PWT) policies. There are many kinds of probabilistic policies: e.g., one that specifies an mean waiting time for service, or one that says only that service will eventually be provided with probability one. Both of these policies are, in some sense, weaker than the FWT policy, since it is possible that some individual service request will never be satisfied. On the other hand, the FWT policy does not guarantee any bound on average service

time, and a probability-one service is it almost as good as FWT service.

Performance requirements often specify a mean waiting time for service, or other statistical constraints on waiting time. The difference between denial-of-service policies and performance requirements is just that performance requirements may assume some probabilistic load model, while denial-of-service protection must consider worst-case stress due to malicious processes.

A real-world example of a policy with probability-one service without a FWT guarantee is CSMA/CD (carrier-sense multiple-access with collision detection) on a local-area network. An attempt by two processors to place a message on the bus at the same time results in a random "back-off" wait and a retry by both. It is possible, though only with probability zero, that every retry results in another collision, forever. Yet the usual performance is satisfactory for reasonable loads.

The random element in a probabilistic policy would be part of the DPB itself, and not due to user behavior. For example, a DPB might randomly select the next process to run or the next resource to revoke.

With a probabilistic policy, the best that can be said is that there is some designated probability such that a request will be satisfied within a certain time. This is still much better than a system in which malicious processes could repeatedly force an arbitrarily long delay in request satisfaction. Because the random element is not under the control of malicious processes, each process can expect consistent, if not good, service, on the average, despite malicious interference.

A malicious process can defeat any waiting time policy for itself, by attempting to hold a resource forever (i.e., insisting on an infinite service time). If some other process requests that resource, it will eventually become necessary for the DPB to revoke the resource from the malicious process. Since that process has not completed its requested service, the malicious process has been denied service. However, it is reasonable not to extend service guarantees to malicious processes.

Finally, we should remark that the most viable policies might not be simple, but perhaps a combination of different types. Furthermore, different classes of processes might have different priorities, resulting in different time bounds or other policy parameters.

2 Resource Monitor Model

2.1 Introduction

The following model is based on a task-resource model given in Coffman and Denning [CoDe'73], used to study deadlock. The new aspect of our model, needed to address denial of service, is the introduction of time. We will also define denial-of- service protection in the context of this model.

2.1.1 Basic Sets and Parameters

A *resource monitor (RM)* is built on a set of *processes* P and a set of *resource types* R. The number of resource types is finite. Each resource type $r \in R$ has a *capacity* $\mathbf{c}(r)$, representing the number of (interchangeable) units of that resource in the system.

Note that the resources reflected in R are shared resources. Each process might also have some private resources that it is allowed to hold as long as it likes. We assume that those resources are not subject to denial.

A process is in one of two states: running or asleep. It is running when it has been allocated a resource of the particular type r_{CPU}. The RM may permit more than one process to be running at a time; this would be the case with a multiprocessor or network. Since each process (by definition) can occupy only one CPU, the capacity of the CPU resource is always one unit, even though there may be several CPUs in the system. The limitation of one CPU per process does not prevent processes from cooperating in some higher-level organization such as a process family or distributed transaction.

2.2 State Structure

The current state of the monitor is represented by a 4-tuple $(A, T, {}^{\mathrm{S}}Q, {}^{\mathrm{T}}Q)$, where A is the *allocation matrix*, T is the *time vector*, ${}^{\mathrm{S}}Q$ is a *space requirement matrix*, and ${}^{\mathrm{T}}Q$ is a *time requirement matrix.* Their characteristics are specified below.

We will write

$$(A, T, {}^{\mathrm{S}}Q, {}^{\mathrm{T}}Q) \rightarrow (A', T', {}^{\mathrm{S}}Q', {}^{\mathrm{T}}Q')$$

to denote a possible state transition. In formal axioms below, primed state components will always refer to a state that may (directly) follow the state referred to by unprimed components.

2.2.1 Allocation Matrix

The allocation matrix, A, is a function on $P \times R$ into $\mathbf{N}$, where $\mathbf{N}$ is the set of non-negative integers. This function can be represented as a matrix if one chooses some fixed ordering of the processes and resources, as was done by Coffman and Denning. (Although we have not assumed that the number of processes is finite, in actual systems the number that are active at a given time will be manageable.)

The value of the allocation matrix $A(p, r)$ is the number of units of resource type r that are currently allocated to process p. The allocation vector A_p for a process p is one row of the matrix, defined by

$$A_p(r) = A(p, r).$$

The CPU resource is allocated with an amount of one each time a process begins running, i.e., is *activated*, and is deallocated to zero each time it goes to sleep, i.e., is *deactivated*. In fact, the amount of CPU allocation is what determines whether the process is running or asleep. Let us define

$$\text{running}(p) \text{ if } A_p(r_{CPU}) = 1$$

in the context of a particular state, and

$$\text{asleep}(p) \text{ if } A_p(r_{CPU}) = 0.$$

The next-state notation also applies to $\text{running}'(p)$ and $\text{asleep}'(p)$, referring to the state of p after a transition.

There is a feasibility constraint on space allocation. If we regard $\mathbf{c}$ as a vector, the constraints on each resource are expressible all at once as a vector inequality:

$$\sum_{p \in P} A_p \leq \mathbf{c} \qquad (R1)$$

The total amount of currently allocated units of any resource must not exceed the system capacity for it.

2.2.2 Time Vector

The time vector T is a function on P into $\mathbf{N}$. Its value $T(p)$ represents the system real time at the last time process p was activated or deactivated. It will be needed to help specify waiting-time policies.

Time is measured in the number of "ticks" of some sufficiently fine-grained time unit relative to an arbitrary origin (e.g., we could choose time zero to be when the system was initialized).

Time passes while running processes are using their time slices, and also while the resource monitor is performing its reallocation functions. These activities occur while the system is in some state, and state transitions are viewed as instantaneous events occurring at the end of the activity.

The time vector is an abstraction of the real-time clock of the CPU on which the process is running. Clocks on different CPUs might not be synchronized, so we cannot, in general, assume that new clock readings are greater than prior values obtained by a different process. Successive readings by different processes on the same CPU will increase, but the model does not keep a history of which CPU was used by which process. We must be cautious about interpreting process virtual time and its relation to real time in large asynchronous systems.

We will require that an activation or deactivation event time is (at least one tick) greater than the previous event time for the same process. Thus,

$$\text{if } A_p(r_{CPU}) \neq A'_p(r_{CPU}) \text{ then } T'(p) > T(p). \qquad (R2)$$

2.2.3 Requirement Matrices

The space and time requirement matrices indicate the resources needed by a process to complete its current task.

Both matrices serve as a means of communicating requests from processes to the system. They are regarded as inputs to the state machine, and are never modified by the resource monitor (this is different from the model in [Mill'92]). Consequently, there are no constraints (other than value-type) on them imposed by the resource monitor. User processes may voluntarily observe constraints such as user agreements, however. These will be discussed later.

2.2.4 The Space Requirement Matrix

The space requirement matrix ${}^{\mathrm{S}}Q$ is a function on $P \times R$ into $\mathbf{N}$. The value ${}^{\mathrm{S}}Q(p, r)$ is a non-negative integer representing resource units of resource r required by process p for its current task.

Each row of the matrix, for a process p, is a *space requirement vector* ${}^{\mathrm{S}}Q_p$. It is the function on R into N defined by

$$ {}^{\mathrm{S}}Q_p(r) = {}^{\mathrm{S}}Q(p, r). $$

Incidentally, we do not insist that a process have requirements for more than one resource at a time, or, when several resources are needed, that a process request them all together. In many systems, a process may only be able to request one resource at a time. However, in other systems, what appears as a single request at the system call interface is in reality a request to claim several resources, so the flexibility for multiple simultaneous requests should be available in the model. Also, even when only one resource is requested at a time, and previously requested resources have not been released, the space requirement matrix will show the accumulated resources.

There is no separate matrix (as there is in [CoDe'73]) for release requests; instead, the fact that a space requirement has become less than the currently allocated amount is an indication that the process no longer needs those resources.

2.2.5 The Time Requirement Matrix

The time requirement matrix ${}^{\mathrm{T}}Q$ is a function on $P \times R$ into $\mathbf{N}$. The value ${}^{\mathrm{T}}Q(p, r)$ is a non-negative integer showing how long, or how much longer, the process p wants exclusive access to each resource r. Time is measured in a number of "ticks", and it represents virtual time, i.e., it should be credited only while the process is running.

For each process p, the corresponding row of the matrix is a *time requirement vector*, ${}^{\mathrm{T}}Q_p$, defined by

$$ {}^{\mathrm{T}}Q_p(r) = {}^{\mathrm{T}}Q(p, r). $$

It is not usual in operating systems for processes to have to announce how long they want a resource. Nor is it common for the operating system

to keep track of how long a process has had a resource, with the exception of the CPU, although this has been done for purposes of performance monitoring. More explicit attention needs to be paid to this aspect of resource management in a system that provides denial-of-service protection. If it is considered burdensome for a process to make explicit resource time requests, one could provide them implicitly instead, by assuming that each resource is requested for some default amount of virtual time that is fixed as a parameter of the system or of the particular resource.

2.2.6 Sharing and Locking

The model assumes that individual units of a resource cannot be shared. Yet files can be shared for read access. One way to represent this in the model is to artificially view a file as a resource type with a very large capacity, larger than the number of processes that might concurrently want read access. Each process requesting read access asks for only one unit of the resource.

Ordinarily, write accesses are exclusive; they are not shared with any other process, regardless of which mode of access the other process wants. This can be handled by treating a request for write access as a requirement for the entire capacity of the resource.

Another way to handle this, slightly more elegant but still artificial, is to consider a file as a resource with only one unit. A request for read access is interpreted as a requirement, not by the process requesting the access, but by an imaginary global system daemon, the same one regardless of which process made the request. Subsequent read accesses by other processes are no problem because the daemon already has the resource. A write access request would require transferring the resource from the daemon to the requesting process, possible only when all read accesses are released.

2.2.7 Progress and Simultaneity

Since the space requirements requested by a process are simultaneous requirements, there is no point in executing a process until all its space requirements have been satisfied, i.e., when its space requirement vector is met (or exceeded) by its allocation vector. Otherwise it cannot make progress on its current task. This will be reflected as a constraint on activation transitions.

With this assumption, it is fair to view a process as making progress whenever it runs. This does not mean that the process will finish its task, merely that it has not been permanently blocked. The fact that a process has made progress should show up as some change in its time or space requirement vector. This change represents the result of a process redetermining its needs during its execution.

As a process executes, it may make arbitrary changes in its requirement vectors. Time requirements do not always go down, since processing time may be data dependent and therefore not fully predictable when a resource is first requested. But if a process can run for a time slice and have the same space and time requirements at the end of it, it could do the same thing repeatedly and hog its resources forever. It is important, then, to have a user agreement to show some progress after each time slice. Of course, if the only progress made is to increase a time requirement, the process may try to hold a resource forever; clearly some additional user agreement would also be required.

When time requirements are implicit and maintained by the resource monitor rather than the user processes, there could be a convention that if there is no change in space requirements, the time requirements will be assumed to be reduced automatically by the size of the time slice when the process is deactivated. This is an example of how the system can help to support one kind of user agreement.

Theoretically, there might be a concern that a process could run an infinite number of times, with converging time slices, so that it progressed only a finite amount of virtual time, which might be less than it needed to finish a task. That cannot happen, however, when time is allocated in discrete ticks, since the time slices cannot approach zero.

2.3 Constraints on Transitions

Recall that a transition is a *deactivate transition* for p when process p has just surrendered the CPU; i.e., running(p) and asleep$'$(p). Also, a transition that allocates the CPU, i.e., a transition such that asleep(p) and running$'$(p), is an *activate* transition for p. Transitions which modify A_p, but are neither deactivate nor activate transitions for p, are called *reallocation* transitions for p.

Note that when we refer to a transition as a "deactivate transition" or

a "reallocation transition" the characterization is relative to some process p. The same transition may be a deactivate transition for one process, an activate transition for another, and a reallocation transition for a third, while still other processes are unaffected.

We also refer to a state as an *activation state* for a process if it is the outcome of an activate transition for that process. As remarked earlier,

The allocation vector must meet or exceed the space requirement vector in the activation state.

$$\text{if asleep}(p) \text{ and running}'(p) \text{ then } A'_p \geq {}^{\mathrm{S}}Q'_p. \qquad (R3)$$

It seems reasonable to assume that a resource monitor does not allocate or deallocate a process's resources while it is running. Hence, we require that:

No reallocation affects a process that remains running:

$$\text{if running}(p) \text{ and running}'(p) \text{ then } A'_p = A_p \qquad (R4)$$

Furthermore, the requirement matrices are under the control of running processes, and cannot be changed by the resource monitor. Changes caused by a running process in its requirements are not visible to the resource monitor until the process deactivates. Hence,

Requirement changes are seen only on deactivation.

$$\text{if asleep}(p) \text{ or running}'(p) \text{ then} {}^{\mathrm{S}}Q_p = {}^{\mathrm{S}}Q'_p \text{ and } {}^{\mathrm{T}}Q_p = {}^{\mathrm{T}}Q'_p. \qquad (R5)$$

Process deactivation, incidentally, does not necessarily result in swapping out the process. Whenever the system is running its own code, as opposed to that of the application, the time taken may be regarded as system overhead rather than progress, so the process is considered to be asleep during that time. This also implies that a process may voluntarily deactivate itself simply by requesting any operating system service, such as an I/O operation.

2.3.1 Illustration

The figure illustrates a succession of possible transitions for one process and resource. It shows CPU allocation (time and space requests for the CPU

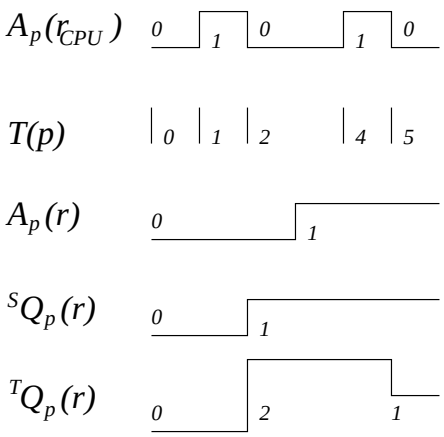

Figure 1: Sample Transitions, One Process and Resource

are assumed but not shown), the time, a resource request, allocation of a resource r, and decrementing of the time requirement for r on the next deactivate transition. (Time slices are only one tick long in this example, for simplicity, but they would normally be longer.)

2.4 Summary

A resource monitor has been defined as an abstract machine whose current state is of the form $(A, T, {}^{\mathrm{S}}Q, {}^{\mathrm{T}}Q)$, subject to (R1), and whose state transitions are governed by (R2) - (R5). The next section discusses what else is necessary to create a denial-of-service protection base.

3 Denial-of-service Protection

The definition of resource monitor in Section 2 just identifies a family of abstract machines. It does not say which members of the family provide denial-of-service protection. In order to specify a denial-of-service protection base (DPB), it is necessary to constrain the states and transitions further, by imposing conditions expressing the denial-of-service policy.

Our present objective is, first, to introduce the additional concepts needed to define denial-of-service protection precisely in the context of our model,

and then to give an example of the process of showing that a resource monitor algorithm provides denial-of-service protection.

3.1 DPB Definition

3.1.1 The Resource Allocation Algorithm

In the general definition of a resource monitor, many different state changes are possible from a given state. The time at which a state change may occur, and changes in the allocation set, are under the control of the resource monitor. If one supposes that the resource monitor acts according to some time-driven algorithm, one can regard the system as a deterministic machine whose inputs are the requirements vectors specified by the processes on deactivate transitions. Another possibility is that the resource monitor has a probabilistic algorithm, with a stochastic element in its decisions. The result is a probabilistic machine.

From a denial-of-service protection point of view, some resource allocation algorithms are better than others. In order to do its job, a resource monitor must also expect some cooperation from user processes. For example, it might insist that user requirements be feasible.

3.1.2 Feasible Requirements

A space requirement vector ${}^{\mathrm{S}}Q_p$ is *feasible* if its space requirements do not exceed the system capacity, i.e.,

$$ {}^{\mathrm{S}}Q_p \text{ is feasible if } {}^{\mathrm{S}}Q_p \leq \mathbf{c} \qquad (F1) $$

Because requirement vectors are settable by processes that may be malicious, they are not necessarily feasible. Non- malicious processes may be constrained to feasible requirement vectors by user agreements.

3.1.3 User Agreements and Benign Processes

Generating infeasible requirements is an example of how a process may make it impossible for the resource monitor to guarantee progress for it, since the

process will never be activated. Although the feasibility constraint just mentioned would apply to any system, there may also be additional constraints that are specific to a particular resource monitor algorithm. Examples will be given below.

In general, it is legitimate for a resource monitor algorithm to be accompanied by additional constraints on requirement vector changes during deactivate transitions. These constraints are called user agreements. If they are respected by a process, the process is called benign.

3.1.4 Waiting-Time Policies

The various types of waiting-time policies, MWT, FWT, or PWT, can be formalized using the model. These policies are expressed in terms of two states,$S = (A, T, {}^{\mathrm{S}}Q, {}^{\mathrm{T}}Q)$ and a later state $S'' = (A'', T'', {}^{\mathrm{S}}Q'', {}^{\mathrm{T}}Q'')$ that are not necessarily consecutive. The MWT and FWT policies are shown below. Probabilistic policies are not shown because they need additional apparatus to exhibit probability distributions, and are beyond the scope of this paper.

Maximum Waiting Time Policy:

$$\exists B : \forall p, S, \exists S'' : \mathrm{running}''(p) \text{ and } 0 < T''(p) - T(p) \leq B$$

Finite Waiting Time Policy:

$$\forall p, S, \exists S'' : \ \mathrm{running}''(p) \text{ and } T''(p) \geq T(p)$$

These policies state that a process will be activated within a fixed or finite time. Strictly speaking, they do not guarantee progress, but only provide opportunities for progress. We can expect that a benign process will always be able to make some progress when given an opportunity, however, since time slices have a minimum size. Whether the minimum time slice is enough in a practical sense is an implementation question.

3.1.5 Time-Boundedness

Although any requirements change implies progress for the process that makes the change, user agreements are also needed to ensure that a process eventually releases resources needed by other processes. One type of agreement that will help serve that purpose is time-boundedness.

A resource is *time-bounded* for a process if the time requirement for that resource is bounded by a quantity that is (1) set at a fixed amount when the resource is first requested and (2) thereafter decreases by each slice of running time. Formally, we attach a matrix $M(p,r)$ to the state representing the time left for p to hold r. Also, we specify a maximum holding time $h(r)$ for each resource, such that:

$$^{\mathrm{T}}Q_p(r) \leq M(p,r) \qquad (B1)$$

$$\text{if } A_p(r) = 0 \text{ then } M(p,r) = h(r) \qquad (B2)$$

$$\begin{aligned} &\text{if running}(p) \text{ and asleep}'(p) \textit{ and } A_p(r) > 0 \text{ then} \\ &M'(p,r) = Max\{M(p,r) - (T'(p) - T(p)), 0\}. \end{aligned} \qquad (B3)$$

That is, (B1) the current time requirement is bounded by the time left, (B2) the time left is equal to the maximum holding time if the resource is not allocated, and (B3) the time left decreases by the amount of each time slice.

Note that a process may ask for less than the maximum initially, in which case the time requirements on some resources may increase. However, the bound on the time requirement always decreases by the time slice on each deactivation, until the time requirement becomes zero. On a subsequent deactivation, the time requirement may go up again, whether the resource was revoked or not.

Time-boundedness is not usually appropriate for the CPU, since a process will typically request another time slice every time it is deactivated (until it finally, voluntarily, terminates), and in time-sharing systems it is not unusual for some processes to exist indefinitely. (Although batch jobs may very well have a CPU time limit.)

There is still the question of how long a process may run before deactivation. If a process is never deactivated, it can hold any resource it has forever, since reallocations occur only on or after deactivation. But, if we assume that time slices are bounded above, it follows that any allocated resource can eventually be reallocated.

Another problem is that before a process releases one resource, it can request another, so that a competing process that wishes both resources will be blocked. This kind of situation can lead to deadlock, but it can be dealt with by various tactics such as ordered acquisition agreements, revocation, etc.

3.1.6 The Denial-of-Service Protection Base

A DPB is characterized by:

- a resource monitor,
- a waiting time policy (e.g., MWT, FWT, or PWT), and
- user agreements.

It must satisfy the following conditions:

(Progress) Each benign process will make progress in accordance with the waiting time policy.

(Patience) No non-CPU resource is revoked from a benign process until its time requirement is zero.

Formally,

$$\text{if } r \neq r_{CPU} \text{ and } A_p(r) \neq 0 \text{ and } A'_p(r) = 0 \text{ then } {}^{\mathrm{T}}Q_p(r) = 0.$$

Note that a DPB may insist on some outrageous user agreement – for example, that the requirement vectors must always be all-zero. Such a DPB is easy to implement but will not garner a large market share.

3.2 A DPB Example

We will give a simple abstract example of a DPB that provides denial-of-service protection with a MWT policy. The purpose of the example is to illustrate the process for proving that a DPB's policy is satisfied, and

not to recommend the particular resource allocation algorithm used. The resource monitor is unconstrained except that we will assume it has only one CPU, and it employs the allocation algorithm given below. For this DPB, a process is benign if it satisfies the following user agreements on deactivate transitions:

1. Its space requirement vectors are feasible.
2. For each process, there is a fixed time-bounded resource type, which we will call its sentinel, whose time requirement is always the largest.

The resource monitor (RM) allocation algorithm is as follows.

- In the initial state, no resources are allocated to any process.
- In each state, the RM has a currently favored process, which will remain favored until the time requirement for its sentinel resource becomes zero or the RM has determined that it is malicious. The RM will then select another favored process. All processes will become favored in a round-robin fashion.
- Just before the RM selects a favored process, it will always be in a state where no resources are allocated. Hence, the next benign favored process can be given all its space requirements. (A process with infeasible requirements is malicious and will be passed over.) The process is activated and runs for a maximum time slice called the *quantum* and denoted q, or until a voluntary deactivation occurs.
- The RM maintains a register H that is initialized to the maximum holding time H_0 of the sentinel resource when a process becomes favored. At the conclusion of each time slice, H is decremented by one tick. If, on any deactivation, any time requirement exceeds H, the process is identified as malicious, and its resources are revoked.
- At the conclusion of the time slice, the RM returns the favored process to running status and gives it another time slice. The same process is run repeatedly until its time requirement vector becomes zero, at which time all its resources are revoked, or until the RM determines that it is malicious.

Clearly, no resource is revoked from a benign process until its time requirement is zero. We will show that there is a maximum time for each cycle; that will prove denial-of- service protection.

We claim that on each cycle, the entire time requirement vector of the currently favored process will reach zero within a bounded time, because of time-boundedness on the sentinel, and a further assumption that RM activity is bounded.

By user agreement (2), the register H serves as an upper bound for all elements of the time requirement vector. By (B2) and (R2), each time slice reduces the maximum time requirement by at least 1. Hence, the time requirement vector is reduced to zero within a total running time of H_0 (or the process is seen to be malicious).

Now, let K be the time needed by the RM from the beginning of the cycle to the time it first activates the favored process. This is the time during which the RM allocates all the requested resources and updates the space requirement vector. Since there is a fixed finite limit to the system resources, this activity takes a bounded amount of time.

For each time slice, the running time is between 1 and q; the worst-case total arises from assuming 1, since this maximizes the system overhead. This represents the case in which the process is changing its requirements and voluntarily deactivating as often as possible. Between successive time slices, the RM takes a bounded time, say D. When the time requirement vector reaches all-zero, the RM can revoke all resources from the favored process. The time for this is bounded by, say, L.

The total time taken to reduce the favored process's time requirements to zero is therefore at most

$$K + H_0(D + 1) + L.$$

After this, the RM will proceed to the next favored process. If the total number of processes is N, the whole cycle will take at most:

$$N(K + H_0(D + 1) + L).$$

This will do as a very conservative waiting time bound.

3.3 Application to Networks

Denial of service is an important problem in networks. Some real examples in this category are given in [Glig'86]. While our approach to denial-of-service modelling is designed to handle concurrency and multiprocessing systems, more effort is needed to extend it in ways suitable for networking objectives. For example, the waiting time policies given here measure the time between events of a single process. The principal service provided by a network, however, is a communications service between different processes running (typically) on different CPUs. The waiting time we need to measure for a communications service might be, for example, the time from a "send" request by one process to the time at which a "receive" request by the destination process will yield the message that was sent.

We could still use this model if we interpret it differently. We need to take a more abstract view of what a "process" is. Here is a sketch of how this might be accomplished. We could define a "transport process" as a series of transient tasks joined by links. A link is a DPB function that performs interprocess communication, either locally or across a network link. As a DPB function, a link transmits the contents of a buffer from one process to another over a medium that is no less reliable than the hardware involved. (Keep in mind that we are concerned here only with software attacks.)

Successive tasks of the same transport process pass messages, one to the next. A local task process could leave a message in a buffer and deactivate itself with a request for a link. Its new space requirement will be for a buffer accessible to the destination process (perhaps in a different CPU). The newly created destination process is viewed as the next incarnation of the same transport process. Meanwhile, the sending task has terminated. All tasks occupy the same "process" row in the model.

Some experience in applying this proposal would be necessary to establish its practicability. This approach was presented only to make the point that the more obvious local interpretation of the model is not necessarily the only one, and that there may be ways of dealing with network objectives that take advantage of this model.

4 Conclusion

By focussing on malicious attacks by untrusted software on a service that allocates shared resources, we have arrived at a model that takes advantage of, and further defines, several concepts that have been suggested in prior work on denial of service, such as user agreements. A denial-of-service protection base (DPB) has been characterized as a resource monitor closely related to a TCB, supporting a waiting-time policy for benign processes. Resource monitor algorithms and policies can be stated in the context of a state-transition model. The possibility of probabilistic waiting-time policies have been suggested in addition to the finite- and maximum-waiting-time policies. The model supports concurrency and multi-processing.

The separation between requirement matrices, for requests from user processes to the resource monitor, and the time vector and allocation matrices, as a record of how the resource monitor has made allocations, is needed to handle potentially uncooperative behavior by malicious processes. The particular structure of the time vector, showing events per process rather than a single global state time, is convenient for stating waiting-time policies.

The simple example of a DPB was given only as an illustration, permitting a feasiblity and consistency check on the definitions. The example of a DPB illustrates the form of argument that can be made to show denial-of-service protection. However, the algorithm given is essentially a batch-processing algorithm. It would be unsatisfactory in a real-time or interactive environment, because it does not adapt to give any process more frequent time slices.

In practice, the job of proving DPB properties is much more difficult. There are a number of complications that arise. In our example, we implicitly assumed that the reference monitor cannot be interrupted except by the end of a time slice or a voluntary deactivation request. In a real system, that assumption may be difficult to show or simply not true. The example was also a single-processor system; it does not investigate the impact of parallel processes, such as the possibility of multiple stacked interrupts.

Future work should include examples of probabilistic waiting-time policies and investigation of more realistic reference monitor algorithms with some general results about them, both in single-processor and multiprocessor or network architectures.

5 References

References

[DoDNW] (no author) Proceedings of the Department of Defense Computer Security Center Invitational Workshop on Network Security New Orleans, LA, March 19-22 1985

[ITSEC] (no author) *Information Technology Security Evaluation Criteria* (DRAFT) der Bundesminister der Innern, Bonn, May 1990

[TCSEC] (no author) *Department of Defense Trusted Computer System Evaluation Criteria* DOD 5200.28-STD, December 1985

[CTCPEC] (no author) *Proceedings of The 1990 CTCPEC Availability Workshop* February 6-7 1990 Communications Security Establishment, Government of Canada

[BaKu'91] E. M. Bacic and M. Kuchta Considerations in the Preparation of a Set of Availability Criteria *Third Annual Canadian Computer Security Symposium* Ottawa, Canada, 15-17 May 1991 pp. 283-292

[Glig'83] V. Gligor A Note on the Denial-of-Service Problem *Proc. 1983 Symposium on Security and Privacy* IEEE Computer Society pp. 139-149 1983

[Glig'86] V. Gligor On Denial of Service in Computer Networks *Proc. International Conf. on Data Engineering* Los Angeles, CA, IEEE pp. 608-617 1986

[YuGl'90] C-F. Yu, V. D. Gligor A Specification and Verification Method for Preventing Denial of Service *IEEE Trans. on Software Engineering*, Vol. 16, No. 6, June 1990 pp. 581-592

[CoDe'73] E. G. Coffman, Jr., P. J. Denning *Operating Systems Theory* Prentice-Hall 1973

[Dobs'91] J. Dobson Information and Denial of Service *Database Security V, IFIP Transactions A-6* IFIP, North- Holland 1991 pp. 21-46

[Mill'92] J. K. Millen A Resource Allocation Model for Denial of Service *Proc. 1992 IEEE Computer Society Symposium on Research in Security and Privacy* IEEE Computer Society, May 1992 pp. 137-147