Chapter 3

Message Filtering and Mandatory Security Enforcement

In this chapter we introduce the message filter object-oriented security model. This model is used to enforce basic mandatory access control among objects. We begin with a discussion of the message filtering algorithm and message filtering functions. This is followed by a discussion of architectural and other implementation issues. In particular, we illustrate what it takes to map an abstract specification of the filtering functions to an executable one.

3.1 The Message Filter Model

Objects and messages constitute the main entities in the message filter model. As far as the security model is concerned, an entire object is classified at a single level. Modeling flexibility is not lost due to this as a user may model multilevel entities. The multilevel entities form a conceptual schema that is broken down into an implementation schema of single-level objects [JK90]. Messages are assumed, and required to be, the only means by which objects communicate and exchange information. Thus the core idea is that information flow be controlled by mediating the flow of messages. Consequently, even basic object activity such as access to internal attributes and object creation, are to be implemented by having an object send messages to itself (we consider such messages to be primitive messages).
The message filter is the analog of the reference monitor in traditional access-mediation models. The message filter takes appropriate action upon intercepting a message and examining the classifications of the sender and receiver of the message. It may let the message pass unaltered or interpose a NIL reply in place of the actual reply; or set the status of method invocations as restricted or unrestricted (explained later).

### 3.1.1 The Message Filtering Algorithm

The message filter algorithm is given in figure 3.1. (In this and other algorithms, the % symbol is used to delimit comments.) Cases (1) through (4) deal with abstract messages, which are processed by methods. Cases (5) through (7) deal with primitive messages, which are directly processed by the security kernel. In case (1), the sender and receiver are at the same security level, and the message $g_1$ and its reply are allowed to pass. In case (2) the levels are incomparable and thus the filter blocks the message from getting to the receiver object, and further injects a NIL reply. Case (3) involves a receiver at a higher level than the sender. The message is allowed to pass but the filter discards the actual reply, and substitutes a NIL instead. (As we have argued, the timing of this NIL reply is a critical consideration.) In case (4), the receiver object is at a lower level than the sender and the filter allows both the message and the reply to pass unaltered.

The cases (1) through (4) that we have seen so far deal with abstract messages. However abstract messages will eventually lead to the invocation of primitive messages. These include read, write and create (cases (5) through (7)).\(^1\) Now read operations always succeed, while writes succeed only if the status of the method in-

\(^1\)The delete operation has not been directly incorporated into the model. It can be viewed as a particularly drastic form of write and is subject to the same restrictions.
% let $g_1 = (h_1, (p_1, \ldots, p_k), r)$ be the message sent from $o_1$ to $o_2$ where
% $h_1$ is the message name, $p_1, \ldots, p_k$ are message parameters, $r$ is the return value
% i.e., $g_1$ is a non-primitive message
if $o_1 \neq o_2 \lor h_1 \not\in \{\text{read}, \text{write}, \text{create}\}$ then case
% (1) $L(o_1) = L(o_2)$:
% let $g_1$ pass, let reply pass
% invoke $t_2$ with $rlevel(t_2) = rlevel(t_1)$;
% $r$ = reply from $t_2$; return $r$ to $t_1$;
(2) $L(o_1) < L(o_2)$:
% block $g_1$, inject NIL reply
% $r$ = NIL; return $r$ to $t_1$;
(3) $L(o_1) < L(o_2)$:
% let $g_1$ pass, inject NIL reply, ignore actual reply
% $r$ = NIL; return $r$ to $t_1$;
% invoke $t_2$ with $rlevel(t_2) = \text{lub}[L(o_2), rlevel(t_1)]$;
% where lub denotes least upper bound
% discard reply from $t_2$;
(4) $L(o_1) > L(o_2)$:
% let $g_1$ pass, let reply pass
% invoke $t_2$ with $rlevel(t_2) = rlevel(t_1)$;
% $r$ = reply from $t_2$; return $r$ to $t_1$;
end case;
if $o_1 = o_2 \land h_1 \in \{\text{read}, \text{write}, \text{create}\}$ then case
% i.e., $g_1$ is a primitive message
% let $v_i$ be the value that is to be bound to attribute $a_i$
(5) $g_1 = (\text{read}, (a_j), r)$:
% allow unconditionally
% $r$ = value of $a_j$; return $r$ to $t_1$;
(6) $g_1 = (\text{write}, (a_j, v_j), r)$:
% allow if status of $t_1$ is unrestricted
if $rlevel(t_1) = L(o_1)$
then [$a_j \leftarrow v_j$; $r \leftarrow \text{SUCCESS}$]
else $r \leftarrow \text{FAILURE}$;
return $r$ to $t_1$;
(7) $g_1 = (\text{create}, (v_1, \ldots, v_k, S_j), r)$:
% allow if $t_1$ is unrestricted relative to $S_j$
if $rlevel(t_1) \leq S_j$
then [CREATE $i$ with values $v_1, \ldots, v_k$ and $L(i) \leftarrow S_j$;
$r \leftarrow i$]
else $r \leftarrow \text{FAILURE}$;
return $r$ to $t_1$;
end case;

Figure 3.1: Message filtering algorithm
voking the operation is unrestricted. Thus if a message is sent to a receiver object at a lower level (as in case (4)), the resulting method invocation will always be restricted and the corresponding primitive write operation will not succeed. This will ensure that a write-down violation will not occur. Finally, the create operation allows the creation of a new object at or above the level of the method invoking the create.

3.1.2 Restricted Methods and Invocation Trees

We now revisit the notion of restricted method invocations alluded to earlier. To start with, observe that in cases (1), (3), and (4) of the filtering algorithm, the method in the receiver object is invoked at a security level given by the variable rlevel. In other words, the method body is executed by a subject (process) running at level rlevel. The intuitive significance of rlevel is that it keeps track of the least upper bound (lub) of all objects encountered in a chain of method invocations, going back to the root (first object and method) of the chain. The value of rlevel needs to be computed for each receiver method invocation. In cases (1) and (4) the rlevel of the receiver method is the same as the rlevel of the sender method. In case (3), rlevel is the least upper bound of the rlevel of the sender method, and the classification of the receiver object. These ideas are illustrated in Figure 3.2 for a method chain starting at an unclassified object.

Let us see how rlevel implements the notion of restricted method invocations so as to prevent write-down violations. Observe that if t_i is a method invocation in
Figure 3.3: A tree with restricted subtrees

object $o_i$, then $rlevel(t_i) \geq L(o_i)$. We say that a method invocation $t_i$ has a restricted status if $rlevel(t_i) > L(o_i)$. When $t_i$ is restricted, it can no longer update the state of the object $o_i$, it belongs to (i.e., its home object). Why? This is because if a method invocation is assigned a level given by $rlevel$, then information classified at $rlevel$ is now available to (flowing into) the method from one or more objects classified at $rlevel$ and encountered earlier in the chain. For example, such information flow could occur through message parameters. To illustrate, consider the secret (S) and confidential (C) objects in the chain in 3.2. The secret object sends a message to the confidential one, resulting in a restricted method invocation in the latter. Secret information could be passed in message parameters to the lower level (confidential) receiver. If information derived from such parameters is used by the receiver method
to update its home object, a write-down violation would occur. Hence the method invocation in the lower level (home) object is restricted.

We can visualize chains of method invocations as belonging to a tree such as in figure 3.3. Restricted method invocations in these chains now show up as restricted paths and subtrees. In figure 3.3, \( t_k \) represents a method in object \( o_k \) that sent a message, and \( t_n \) represents the method invoked in the receiver object \( o_n \). The method \( t_n \) is given a restricted status as \( L(o_n) < L(o_k) \). The children and descendants of \( t_n \) will continue to have a restricted status till such point as \( t_s \). At \( t_s \), the restricted status is removed since \( L(o_s) \geq L(o_k) \) and a write by \( t_s \) to the state of \( o_s \) no longer constitutes a write-down violation.

### 3.2 Implementing Message Filtering

Having given an introduction to the message filter model, we now turn our attention to implementation. In particular, we discuss what it takes to map an abstract specification of the filtering functions (as given in the filtering algorithm) to an executable one [STJ91, TS94a]. We begin by elaborating some architectural considerations.

#### 3.2.1 System Layering and the Security Perimeter

In architectural terms, how should systems be structured and layered to incorporate and enforce mandatory security and message filtering? A good and logical starting point for our investigations is the architecture of existing object-based systems. This is because we want our solutions to be cost-effective and thus fit within existing implementation frameworks.

An analysis of many prototypes and proposals for object-oriented database systems such as GEMSTONE, IRIS, ORION (to name a few) would reveal a common
architectural structure; a demarcation into a lower storage layer and an object layer on top of the storage layer. For example, in GEMSTONE the lower layer is referred to as STONE, while the object layer consists of GEM processes. The lower storage layer essentially interfaces to the operating system and file system primitives, and is responsible for the management (i.e., the read, write, and creation) of typeless chunks of bytes representing objects. Every object (chunk of bytes) is associated and represented by a unique object-identifier. This layer typically does not understand the abstraction of objects or the object-oriented data model, rather sees itself as one that provides basic services to the higher layers such as the object layer.

In contrast to the storage layer, the object layer is not typeless, but rather supports the abstraction of objects as encapsulated and typed units of information. This
layer is thus responsible for implementing the object-oriented data model. Object-oriented concepts such as classes, class-hierarchies, inheritance, as well as message passing lie within the purview of this layer.

Given the above layering structure, how and where will the TCB fit in? Also, what subsets or functions of these layers should lie within the security perimeter? It is clear that some subset of the operating system and the storage layer need to be within the TCB. But what about the object layer? Could we not realize a secure system by having just part of the storage layer in the TCB? If confidentiality were our only objective, the answer to the latter question would be “yes”. Mandatory confidentiality can be enforced by the subset of the operating system and storage layer within the TCB. However, integrity is a vital requirement that has to be maintained alongside confidentiality. The maintenance of integrity for objects cannot be done at the lower layer since it does not recognize the abstractions of the object model. Hence, a subset of the object layer needs to be within the TCB for integrity purposes.

Recall that a good design principle for security kernels is to keep its size to a minimum. We thus require that much of the functionality to implement the object-oriented data model be outside the TCB. Thus even support for the notion of objects as units of encapsulation, is provided by the object layer subset outside the TCB. The subset within the TCB implements high assurance and integrity preserving message passing and filtering. This is accomplished through modules called message managers. A message manager process is created dynamically whenever a message is sent upwards in the security lattice and concurrent execution of the sender and receiver methods is required. Once created, it implements the message filter-
ing algorithm for the chain of methods emanating from such a concurrent receiver method. A message manager is thus a relatively short-lived process, and one that eventually terminates along with the last method in the associated chain. In a trusted subject architecture the level managers that are forked by a single user session are coordinated by a trusted multilevel process called a session manager. A user session encompasses all the computations (activities) initiated by a user between consecutive logins. In an architecture without trusted subjects (such as a kernelized one), the message managers are coordinated by untrusted single-level processes called level managers.

In summary, the message manager and level (or session) manager modules in the object layer need to be within the security perimeter (TCB) so as to ensure serial correctness (mentioned earlier in chapter 2). If the concurrent message managers cannot guarantee this, the integrity of the data in the database could be severely compromised.

3.2.2 An Executable Specification

The message filtering algorithm presented earlier can be thought of as an abstract non-executable specification of the filtering functions. An executable specification, as implemented by a message manager, is given in figure 3.5. The security perimeter of the object layer exports the following operations: send, quit, read, write, and create. The read, write, and create operations handle primitive messages. The system primitives send and quit are used by methods to send messages and replies. A stack is used to save the contexts associated with nested message sends. The interface between a message manager and a level or session manager consists of two calls: (1) fork issued by a message manager to request creation of a new message manager at a higher level and (2) terminate issued by a message manager to its
local level or session manager to terminate itself.

Whenever a message is sent by a method $t_1$ in an object $o_1$ to a second object $o_2$ at the same or lower level (cases (1) and (4)), the message manager saves the message parameters on a new stack frame, suspends execution of $t_1$, and begins execution of the method $t_2$ in object $o_2$. When $t_2$ terminates, the stack is popped and the return value from $t_2$ is recorded on the stack. The suspended sender method $t_1$ is then resumed, and it retrieves the return value from $t_2$ from the top frame of the stack.

When messages are sent to incomparable or higher levels (cases (2) and (3)), a NIL value is recorded on the stack and $t_1$ is resumed immediately. In case (3) when a message is sent upwards in the security lattice, a message manager issues a fork call resulting in concurrent computations (as $t_1$ is resumed independently of the termination of $t_2$). The parameters of this call include the level of forking message manager, the level of the forked message manager, a unique fork stamp identifying the start order in the equivalent sequential execution for the forked message manager, and a vector (astamps) of timestamps to process read down requests. Whenever a reply is returned and a message manager finds its stack to be empty, it means that there are no suspended methods waiting to be resumed. The message manager then issues a terminate call to its local level manager, to terminate itself. The parameters of the terminate call include the level and fork stamp of the terminated message manager, as well as a timestamp identifying the last written version.

In moving from an abstract to an executable specification, we have so far described how the filter allows and blocks messages, and how return values are set to NIL. Now it remains to show how the notions of rlevel and restricted method invocations are implemented. The basic idea is very straightforward. Every message manager (process) is assigned a security level that is equivalent to the rlevel assigned in the filtering algorithm, and all methods executed by a message manager run at this
level. The effect of restricted method invocations is now achieved by the enforcement of the standard $\star$-property in the Bell-LaPadula type security models [BL76]. In other words, whenever a method's status is restricted, its level (and the level of its message manager) will be higher than the object accessed, and the $\star$-property will prevent any write-down attempts.
procedure send(g1, o1, o2)
% let g1 = (h1, (p1, ..., pk), r) be the non-primitive message sent from o1 to o2
% where h1 is the message name, p1, ..., pk are message parameters, and r
% p is the parameter set p1, ..., pk and lmsgmgr is the level of the message manager t1
1. \( L(o_1) = L(o_2) \): push-stack(p);
   \( t_2 \leftarrow \) select method for \( o_2 \) based on \( h_1 \); execute \( t_2 \);
2. \( L(o_1) \sim L(o_2) \): write-stack(NIL); resume;
% Let astamps be a vector that is passed to a forked message manager
3. \( L(o_1) < L(o_2) \): append-astamps-vector(astamps, wstamp);
   fork(lmsgmgr, lub[lmsgmgr, L(o_2)], forkstamp, astamps);
   wstamp ← wstamp + 1;
   write-stack(NIL); resume;
4. \( L(o_1) > L(o_2) \): push-stack(p);
   \( t_2 \leftarrow \) select method for \( o_2 \) based on \( h_1 \); execute \( t_2 \);
end case;
if \( o_1 = o_2 \land h_1 \in \{ \text{read, write, create} \} \) then case % i.e., \( g_1 \) is a primitive message
5. \( h_1 = \text{read} \): if \( L(o_1) = \text{lmsgmgr} \) then \( v \leftarrow \) wstamp
   else \( v \leftarrow \) local-stamp \((L(o_1))\);
   read \( o_1 \) with version ← max\{version: version ≤ v\};
6. \( h_1 = \text{write} \): write \( o_1 \) with version ← wstamp;
% Let \( o \) be the object-identifier of the new object created at level \( S_j \)
7. \( h_1 = \text{create} \): create \( o \) with \( L(o) \leftarrow S_j \) and version ← wstamp;
   write-stack(o);
end case;
end procedure send;
procedure quit(r)
   pop-stack;
   if empty-stack then terminate(lmsgmgr, wstamp, forkstamp)
   else [write-stack(r); resume;]
end procedure quit;

Figure 3.5: Message manager algorithms for SEND and QUIT