7th DOD/NBS Computer Security Conference

September 24-26, 1984

Structure of a Rapid Prototype Secure Military Message System

Mark R. Cornwell
Robert J. K. Jacob

Computer Science & Systems Branch
Naval Research Laboratory
Washington, D.C. 20375

ABSTRACT

Past attempts at building multilevel-secure systems have resulted in human-interfaces that were difficult to understand and use. We posit that part of this difficulty results from a poor fit between conventional security models and the intuitive notion of security users apply to their application. The Secure Military Message Systems project attacks these problems by defining a security model intuitively closer to the application and testing this model by constructing rapid prototype systems and trying them out. Techniques used to construct these rapid prototypes include the definition of abstract data types, an intermediate command language, and an executable formal specification of the human-interface. Features of the MMS security model are presented using examples from a rapid prototype system.

1. The Problem

The Secure Military Message Systems project is building rapid prototypes in order to learn about techniques for building secure computer systems. In the past, secure systems have been built from general-purpose security models. While this yields an internal model that is elegant and easy to understand, the user interface of the resulting systems is often confusing, because it enforces security restrictions that appear counter-intuitive from the perspective of a user. Our solution to this problem is to define a security model that attempts to capture the user's intuitive notion of security in a military message system [1]. Then, we examine the effects of that model on the system behavior visible at the user interface by building and studying a series of rapid prototype systems that implement the model. A sample session with one such rapid prototype is given in the appendix. We have designed message systems offering a representative range of functions for composing, reading, distributing, and processing military messages. While the present security model was motivated by message systems, it has been found to be adaptable to other similar types of document systems.

Our definition of security is embodied in a security model for military message systems. This model is described in detail by Landwehr [1]. The MMS security model consists of a set of definitions, assumptions and assertions. It differs from some conventional models of security such as that of Bell & Lapadula [2] in that it directly models multi-level objects, such as a SECRET message containing CONFIDENTIAL paragraphs; and recognizes message system operations such as RELEASE, rather than the generic READ, WRITE, EXECUTE.

2. Design Goals of the Rapid Prototype Systems

The rapid prototype systems are intended to exhibit the user-visible behavior of secure message systems conforming to the MMS model. To make our prototyping effort feasible we have chosen to concentrate on the security model, functional requirements, and user interface. We have tried to implement these aspects of the design faithfully, while other concerns that could be important for a production system are not addressed. For example, time and space efficiency are not a concern as long as they are acceptable for demonstration purposes. The rapid prototypes support only a few users and a low volume of message traffic. Genuine security was not addressed, but the normal behavior of the rapid prototype is just like that of the corresponding secure system. Some other concerns are important for the rapid prototypes but not necessary for a production system. For example, the rapid prototypes themselves should be designed and built quickly. They should be easy to modify to reflect changes both in functionality and in the underlying security model. At this stage, obtaining a genuinely secure implementation is not as important as obtaining an apparently secure one quickly.

The software decomposition of the M2 rapid prototype (earlier prototypes were called M0 and M1 [3]) does address problems of building a system that satisfies the MMS security model. The internal design has incorporated lessons learned over several generations of rapid prototype systems. In M0 and M1, security checks were widely scattered throughout the code. The decomposition we present here identifies and localizes many of the mechanisms that enforce the MMS security model. This locality increases the promise of a verifiable implementation of the design.

3. Structure of the Rapid Prototypes

The M2 system is partitioned into two components, a user interface component and a semantic action component as shown in figure 1. The user interface handles the details of transforming sequences of keystrokes, mouse clicks, and other user input into requests in an Intermediate Command Language (ICL) and presenting the results of such requests as output to the user. The ICL requests themselves are processed by the semantic action component.

The user interface component is specified as a set of state diagrams after the manner described by Jacob [4]. In the state diagram model, an automaton reads from a stream of tokens and makes a transition to another state based on the token read. Actions may be associated with state transitions; whenever such a transition is made, its associated actions are performed. In the present system, the actions consist of ICL commands, which are transmitted to and executed by the semantic action component. The user interface component is implemented by an interpreter that executes the state diagram specifications [5]. It traverses the diagrams and performs the actions associated with each transition.

This division permits new systems with different user interfaces to be constructed from the existing system conveniently. If the design of the user interface is changed, only the user interface component must be modified so that it will translate from the new command language into the same ICL commands; the semantic component of the system need not be changed. The division also provides a useful decomposition of programming tasks. Given a stable description of the ICL, the user interface and semantic components of the system can be developed in isolation from one another, since the only communication between them is via the defined ICL commands. In fact, the M2 prototype was coded in just this fashion, by the authors working in parallel. Our experience showed

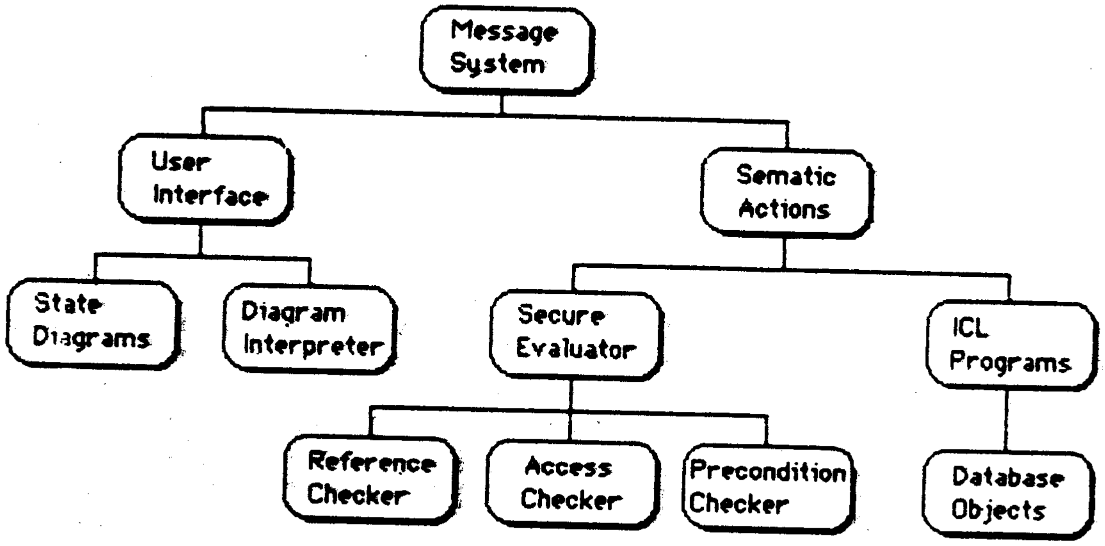


fig. 1. System Decomposition

that, with this decomposition, it was possible to make major changes to either component of the system without affecting the other component or even telling the person writing it.

The user interface itself is designed to be easy for naive users to learn (in contrast to being easy for experienced users to operate). A command typically consists of a verb (such as Display), an object (such as "the file named inbox"), and, possibly, some extra parameters (in this example, a display filter). The verb is selected from a menu; objects and other parameters are selected from menus where possible or else typed in a window. For each item, a default value is available with a single keystroke. After the user interface component has accepted an entire ICL command with all its arguments and found it to be syntactically correct, it issues the complete command to the semantic component. Finally, there are some user-level operations that do not correspond to ICL commands, such as the commands to abort a command, scroll the windows, select special-purpose menus, and exit. These are placed on function keys and may be entered by the user at any point in the dialogue.

The semantic action component of the system is itself partitioned into a secure evaluator and a set of ICL programs. The ICL programs perform the actions requested by the user. The secure evaluator ensures that ICL programs never get a chance to perform actions that would violate security. It performs all the security checks necessary before an ICL program is invoked.

The actions taken by ICL commands are the user's means for inspecting and manipulating the sensitive information in the system. Each command available to the user corresponds to one of these ICL programs. The ICL programs manipulate objects in the message system database, such as directories, message files, citations, and messages, which embody the sensitive information in the system. ICL programs are written in terms of operations on these lower level objects. Unlike the ICL commands, the user does not have access to these lower level operations directly. Users can only invoke them indirectly by invoking ICL commands.

The secure evaluator is made up of programs to perform three kinds of security checks. The reference checker enforces security constraints while determining what objects in the system a requested ICL operation would act upon. The access checker compares the access permissions of these objects with the privileges of the user supplying the request. A precondition checker uses its knowledge of the semantics of the ICL programs to determine if a request should be denied or allowed to proceed.

In the interest of building the system rapidly, we decided to use an existing text editor, Emacs [6], for composing and editing messages. Emacs is an extensible text editor with its own language for defining new commands and modifying its user interface. We took advantage of this extensibility to tailor it to approximate a secure editor for messages and to integrate it into our rapid prototype. It protects some fields of a message from modification and it provides some prompts and syntax checking on messages.

4. Abstract Types, Inheritance and Locality

Most of the implementation is oriented around abstract data types. Our notion of abstract data types is fairly conventional but includes operator inheritance and overloading concepts similar to those of Smalltalk [7], and Flavors [8]. An abstract data type characterizes a class of data by associating a type name with a set of values and a set of named access operators. The access operators may alter values, return information about values, or both. A type may be a subtype of another type. If *B* is a subtype of *A*, then by default *B* inherits the access operators of *A*. *B*'s definition can then add new access operators not associated with *A*. By overloading operator names (defining new operators with the same names as inherited operators) *B* can hide inherited operators.

This notion of types, and operator inheritance in particular, helped in implementing security. A type ENTITY was associated with the security specific information (e.g. classification, access set, CCR mark). Other types such as MESSAGE were defined as subtypes of type ENTITY. The security checking programs were written for ENTITY, and thus didn't depend on specifics of type MESSAGE, only type ENTITY. We found it possible to add new secure types to the system by defining new subtypes of ENTITY without adding new security checking programs.

Our design uses abstract data types to define a wide variety of data classes. Some are visible to users: ENTITY, ACCESS_SET, MESSAGE, MESSAGE_FILE. Others are used internally: STATE_DIAGRAM, TRANSITION, REQUEST.

5. Security Model and Mechanisms for Security Checking

This section describes some specific techniques used to implement the MMS security model in the semantic component of the M2 prototype. Our presentation of the model will be in terms of the techniques we have chosen to implement it, though other implementations could satisfy the model without using these particular techniques. Concepts from the model will be briefly described as necessary. A complete definition and explanation of the model is given by Landwehr [1].

The MMS security model provides a set of definitions and assertions characterizing a secure system. *Entities* are units of information in the system that are associated with protection information. Every entity has a *classification*, an *access set*, a *type* and a *value*. *User ID's* represent the human users of the system. Every user ID has an associated *clearance* and a set of *roles*. Users indicate the entities they access by providing *references* to them. A *direct reference* (e.g., MSG1190) is an atomic identifier that denotes exactly one entity independent of the values of any other entities. Entities may *contain* other entities. An *indirect reference* indicates an entity by referring to an entity that contains it. (e.g., "the fifth message in Cornwell's inbox file"). Say, an entity *e1* contains an entity *e2*. If *e1* is marked *container clearance required (CCR)*, then a user can't use *e1* in an indirect reference to *e2* unless the user is cleared for *e1*.

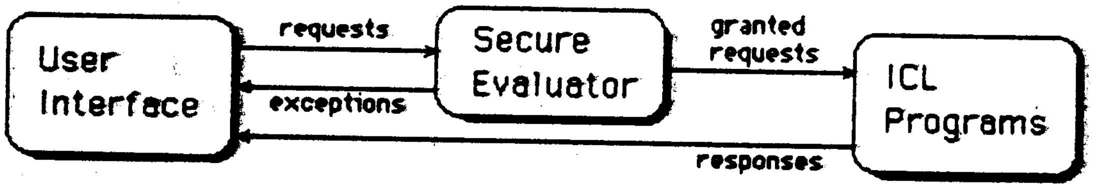


fig 2. Security Mediation

The division of the semantic action component into a secure evaluator and ICL programs separates security checking from the normal case semantics of user commands. The secure evaluator mediates the access of the user interface to the ICL programs as shown in figure 2. The user interface constructs ICL requests and sends them to the secure evaluator, which either grants the request, by invoking an ICL program, or sends an exception back to the user interface. After a request is granted, the requested ICL program can run to completion without the need to perform any other run-time security checks. This obviates the need to consider problems that occur when a operation encounters a security exception after it has begun its execution, such as restoring state or translating low level security exceptions into meaningful responses to the user.

A request that enters the secure evaluator must pass three kinds of security checks before it is granted. Each kind is handled by a different component of the secure evaluator, these components being a reference checker, an access checker, and a precondition checker. A request <*op r1 ... rN*> consists of an operator followed by a sequence of references and possibly other parameters. The requests goes through the following checks:

1) *Reference Checker*. Any violation of constraints imposed by CCR marks is detected at this step. Each of the indirect references is dereferenced. This yields a structure of the form <*op e1 ... eN*> where the *ei* is the the entity denoted by *ri*.

2) *Access Checker*. For each entity *ei* in *e1...eN* we check to see that the access set of the entity permits the user to apply the operator with that entity as its *i*th parameter.

3) *Precondition Checker*. Finally, we perform a check to determine whether performing the requested actions the current state will maintain the security assertions. If so, we apply *op* to *e1...eN*.

Failure to pass any one of the above checks will cause the request to be denied and a security exception to be generated. A brief explanation of why the request was denied is passed back to the user interface which conveys it to the user.

With this outline of run-time security checking in place, we examine each component of the secure evaluator elaborating on the checking each performs.

The reference checker evaluates the references in an ICL request based on the classifications, values, and types of the entities appearing in each indirect reference. Every type of entity that can contain other entities is associated with a selector function S that given an entity and an index returns the entity contained by the given entity at that index. For example, if *e* denotes a directory of message files and *i1* is the name of a file in that directory, then *S(e,i1)* denotes that file. Each indirect reference <*e,i1,...,ik*> is a sequence whose first element is an atomic identifier for an entity and whose remaining elements are indices. There is a procedure for evaluating a reference to determine the entity it denotes. This dereferencing procedure maps references to direct references. It acts as an identity on direct references. For indirect references it replaces the first two elements of its argument with the entity denoted by applying a selector function to e and i1, and applies itself recursively to the result.

The dereferencing procedure also performs some security checks and will generate an exception if attempted violations occur. An exception will occur if the reference is indirect, the entity heading that reference is marked CCR and the user's clearance does not dominate the classification of that entity. Notice that it is possible to dereference an indirect reference that depends on entities for which the user is not cleared without generating an exception.

The access checker compares the access sets of the requested entities with the operator name and the privileges of the user making the request. *Operators* are the commands users may invoke (directly) to change or inspect the entities in the database. In the M2 rapid prototype, these operators are defined to be the ICL programs. The access set of an entity determines what operators may be applied to it and by what users. An access set for a given entity *e* is a set of triples of the form (userID or role, operator, k). A user can use entity *e* as the *k*th parameter of an operator if a triple with the user's userID (or one of the user's roles), that operator and *k* is in the access set of *e*.

The precondition checker compares the state of the system with the requested action to determine whether the action can take place without violating any security assertions not already addressed by the reference checker and access checker. One such assertion is the hierarchy assertion, which states that the classification of every entity must dominate the classification of every entity it contains. For example, a message file contains a set of citations and each citation contains a message. The message file must be classified higher than any of its citations and each citation in turn must be classified higher than its message. Another assertion states that only a user acting in the role of system security officer can change the clearance associated with a user ID.

In order to guarantee that such assertions remain invariant, every operator *op* is associated with a precondition *pre(op)* characterizing the conditions under which applying *op* will leave the system in a state satisfying the security assertions. Before applying any operation, this precondition is checked and an exception is generated if the precondition does not hold. Assuming the preconditions are correct, an operation never causes an action that will invalidate the security assertions.

Preconditions are attractive because mathematical techniques (weakest preconditions, predicate transformers [9] ,Hoare logic [10]) exist to derive them from a specification of the semantics of the operator and a specification of the security invariants. Dijkstra [9], Gries [11], and others have argued convincingly that deriving programs from specifications yields benefits that verifying programs after they are written does not. In deriving precondition checks for our rapid prototype, we applied the concepts of invariant assertions and weakest preconditions informally to specify and program the system. Using this approach, we could then attempt to provide formal proofs that preconditions are sufficient to insure that the operations maintain the security invariants.

The use of an external text editor poses some problems to our security design. Within the message system, messages are represented in linked structures laden with security information. A message sent to the editor is translated into a text form visually familiar to users but with much of the security information (e.g. access sets) stripped off. The system should prevent users from editing certain fields, corrupting the security labels on entities, or entering ill-formed messages into the system. To this end, we partition the message being edited into an editable and noneditable part, displaying the latter in a window where the user cannot modify it. In Emacs, the user edits a textual representation of a message in the absence of any security checking. When Emacs exits, the message system parses the message, checking it for well-formedness and conformity with the security constraints before we allowed it to be stored in the message system data types. For example, a message might fail the well-formedness check if a user mistypes a field name or leaves off a security label on a paragraph. The message would not conform to security constraints if, say, a CONFIDENTIAL text field held a SECRET paragraph.

6. Future Directions

The rapid prototype described here is one of a series of systems being built to investigate secure message systems. The next steps in this work include:

a) Obtain feedback from real message system users using the M2 prototype.

b) Develop a user interface incorporating a bitmapped display, graphics and mouse. This will be done by modifying the user interface component (and moving it to a different host), while leaving the semantic component unchanged.

c) Investigate formal techniques for deriving precondition checks from the specifications.

d) Build a genuinely secure full-scale prototype based on the current rapid prototype.

7. Summary

We have observed that conventional security models, while intuitively appealing to designers, can appear confusing and inappropriate when viewed through the user interface of a finished system. To overcome these problems, a specific security model has been defined to conform to an intuitive notion of how the user interface to a secure message system should behave. A series of rapid prototype systems has been built to allow us to observe directly the interactions between this security model and a user interface. The M2 rapid prototype demonstrates a particular approach to the implementation of a system that incorporates this security model. The techniques used in building it are applicable to other application based security models.

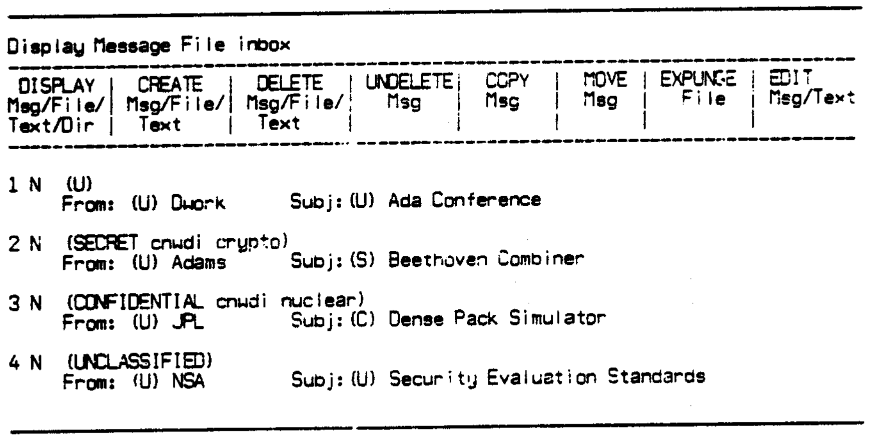
8. Acknowledgments

Carl Landwehr, Connie Heitmeyer, and John McLean developed and formalized the MMS security model. Jean Tschohl, Ken Pon, and Brian Tretick contributed to building and testing the rapid prototypes. This work was supported by the Naval Electronic Systems Command under the direction of H.O. Lubbes.

The following scenario, based on the M2 prototype, illustrates some fundamental ideas of the Secure Military Message System Design. In this example a user, Jones, logs onto the message system and reads some incoming mail. The session illustrates some of the data objects the users manipulate and how message processing is integrated with the security policy.

To gain access to the system, Jones must first log in. Jones does this by providing a userID, the classification that the screen is to assume, in this case (T cnwdi nato crypto), and and a password. A menu appears on the screen from which Jones selects active roles for the session. The system checks to see that a user with userID Jones and the given password is authorized to use the system, and that Jones is authorized for each of the roles selected. With this precondition satisfied, the login operation proceeds.

The screen has a current classification at the level specified at login. Citations from Jones's message file "inbox" are displayed on the screen (*fig. 1a.*)

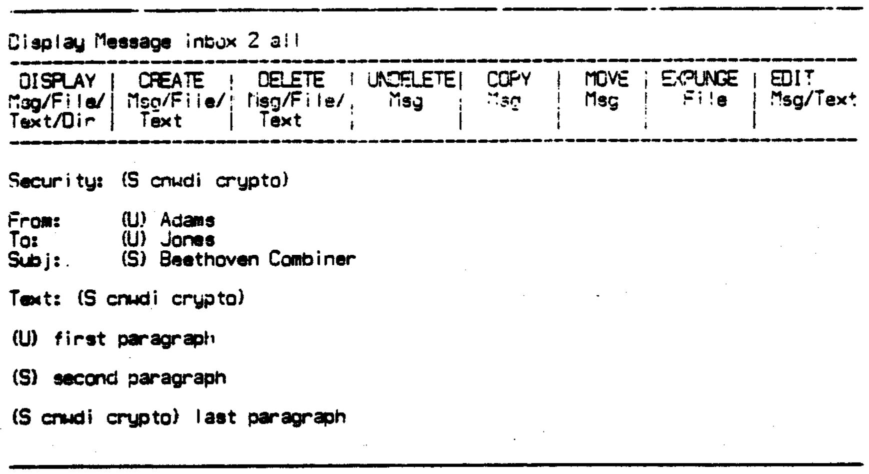


```
Display Message File inbox
----------------------------------------------------------------------------------
 DISPLAY  |  CREATE   |  DELETE   | UNDELETE|  COPY  |  MOVE  | EXPUNGE | EDIT
Msg/File/ | Msg/File/ | Msg/File/ |   Msg   |  Msg   |  Msg   |  File   | Msg/Text
Text/Dir  |   Text    |   Text    |         |        |        |         |
----------------------------------------------------------------------------------

1 N  (U)
     From: (U) Dwork        Subj: (U) Ada Conference

2 N  (SECRET cnwdi crypto)
     From: (U) Adams        Subj: (S) Beethoven Combiner

3 N  (CONFIDENTIAL cnwdi nuclear)
     From: (U) JPL          Subj: (C) Dense Pack Simulator

4 N  (UNCLASSIFIED)
     From: (U) NSA          Subj: (U) Security Evaluation Standards
```

fig. a1.

At this point the screen (an entity) contains a sequence of citations (also entities). Each citation contains the From, Subject, and Security fields of the message to which it refers. Only citations below the the classification of the screen are displayed.

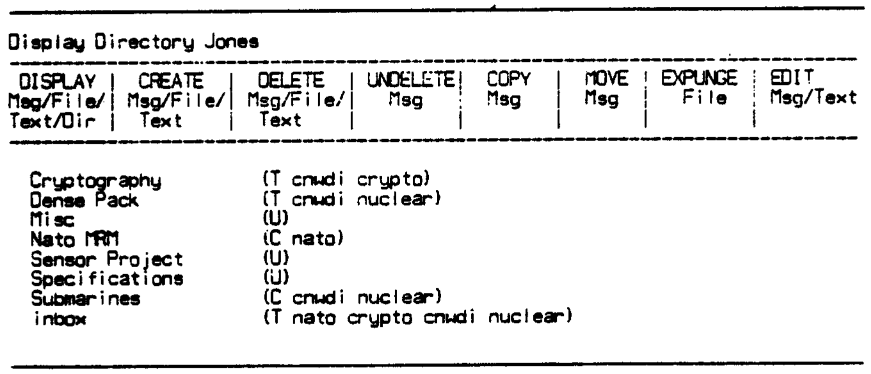
Displaying the second message, Jones can see all of its fields (*fig. a2.*) Notice that the message classification dominates the classification of each of its fields. Similarly, the text field classification dominates the classifications of each of its paragraphs.



```
Display Message inbox 2 all

Security: (S cnwdi crypto)

From:     (U) Adams
To:       (U) Jones
Subj:     (S) Beethoven Combiner

Text: (S cnwdi crypto)

(U) first paragraph

(S) second paragraph

(S cnwdi crypto) last paragraph
```

fig. a2.

Jones's directory contains all of the message files belonging to Jones. The message file names are displayed along with the file classifications (*fig. a3.*)



```
Display Directory Jones

  Cryptography        (T cnwdi crypto)
  Dense Pack          (T cnwdi nuclear)
  Misc                (U)
  Nato MRM            (C nato)
  Sensor Project      (U)
  Specifications      (U)
  Submarines          (C cnwdi nuclear)
  inbox               (T nato crypto cnwdi nuclear)
```

fig. a3.

Accidental violations of the security model are prevented. Jones is informed of denied requests in terms of familiar message system concepts. For example, if Jones attempted to save the second inbox entry (displayed earlier) into the file "Nato MRM", the system would deny the request. A brief explanation would appear in the error window (just below the menu) saying that the message was classified to high to be inserted in the given message file.

Jones may move the message in the file Cryptography, since that file's classification dominates that of the message. Jones does so and logs out leaving the terminal ready for another login.

References

1. C.E. Landwehr, C.L. Heitmeyer, and J. McLean, *A Security Model for Military Message Systems*, Naval Research Laboratory, Washington, D.C. (May 1984).
2. D.E. Bell and L.J. Lapadula, "Secure Computer Systems: Mathematical Foundations and Model," MTR-2779, Mitre Corp., Bedford, Mass. (July 1975).
3. C. Heitmeyer, C. Landwehr, and M. Cornwell, "The Use of Quick Prototypes in the Secure Military Message Systems Project," *Software Engineering Notes* **7**(5) pp. 85-87 (December 1982).
4. R.J.K. Jacob, "Using Formal Specifications in the Design of a Human-Computer Interface," *Comm. ACM* **26** pp. 259-264 (1983).
5. R.J.K. Jacob, "An Executable Specification Technique for Describing Human-Computer Interaction," in *Advances in Human-Computer Interaction*, ed. H.R. Hartson, Ablex Publishing Co., Norwood, N.J. (1984). in press.
6. J. Gosling, *Unix Emacs*, Unipress Software, Inc., Highland Park, N.J. (January 1983).
7. A. Goldberg and D. Robson, *Smalltalk-80 The Language and Its Implementation*, Addeson Wesley (1983).
8. D. Weinreb and D. Moon, *LISP Machine Manual*, Massachusetts Institute of Technology, Cambridge, Mass. (March 1981).
9. E.W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Edgewood Cliffs, N.J. (1976).
10. C.A.R. Hoare, "An Axiomatic Basis for Computer Programming," *Comm. ACM* **12** pp. 576-580 (1969).
11. D. Gries, *The Science of Programming*, Springer-Verlag, New York (1981).