Computer Security[1]

Butler W. Lampson
Digital Equipment Corporation

1 Requirements for Security

Organizations and people that use computers can describe their needs for information security under four major headings:

secrecy[2]: controlling who gets to read information;

integrity: controlling how information changes or resources are used;

accountability: knowing who has had access to information or resources;

availability: providing prompt access to information and resources.

Each user of computers must decide what security means to him. For example, a defense agency is likely to care more about secrecy, a commercial firm more about integrity of assets. A description of the user's needs for security is called a security policy. A system that meets those needs is called a secure system.

Since there are many different sets of needs, there can't be any absolute notion of a secure system. An example from a related field may clarify this point. We call an action legal if it meets the requirements of the law. Since the law is different in different jurisdictions, there can't be any absolute notion of a legal action; what is legal under the laws of Britain may be illegal in the US.

Having established a security policy, a user might wonder whether it is actually being carried out by the complex collection of people, hardware, and software that make up the information processing system. The question is: can the system be trusted to meet the needs for security that are expressed by the policy? If so, we say that the system is *trusted*[3]. A trusted system must be trusted *for* something; in this context it is trusted to meet the user's needs for security. In some other context it might be trusted to control a shuttle launch or to retrieve all the 1988 court opinions dealing with civil rights. People concerned about security have tried to take over the word "trusted" to describe their concerns; they have had some success because security is the area in which the most effort has been made to specify and build trustworthy systems.

Technology is not enough for a trusted system. A security program must include other managerial controls, means of recovering from breaches of security, and above all awareness and acceptance by people. Security cannot be achieved in an environment where people are not committed to it as a goal. And even a technically sound system with informed and watchful management and users cannot dispel all the risk. What remains must be managed by surveillance, auditing, and fallback procedures that can help in detecting or recovering from failures.

The rest of this section discusses security policies in more detail, explains how they are arrived at, and describes the various elements that go to make up a trusted system.

---

[1] This description of requirements and technology for computer security was written for the National Research Council report on computer security [NRC 1991]. Much of this material appears there in a revised form.
[2] Often called 'confidentiality', a seven syllable jawbreaker.
[3] Perhaps we ought to say that it is trustworthy, but we don't.

1.1 Policies and controls

The major headings of secrecy, integrity, availability, and accountability are a good way to classify security policies. Most policies include elements from all four categories, but the emphasis varies widely. Policies for computer systems generally reflect long-standing policies for security of systems that don't involve computers. The defense community is most concerned with secrecy, the commercial data processing community with integrity and accountability, the telephone companies with availability. Obviously integrity is also important for national security: an intruder should not be able to change the sailing orders for a carrier, and certainly not to cause the firing of a missile or the arming of a nuclear weapon. And secrecy is important in commercial applications: financial and personnel information must not be disclosed to outsiders. Nonetheless, the difference in emphasis remains.

A different classification of policies has to do with who can modify the rules. With a *mandatory* policy most of the rules are fixed by the system or can be changed only by a few administrators. With a *discretionary* policy the owner of a resource in the system can make the rules for that resource. What kind of policy is appropriate depends on the nature of the system and the threats against it; these matters are discussed in the next section.

People have also developed a set of tools that are useful for enforcing policies. These tools are called 'management controls' by businessmen and accountants, 'security services' by technologists, and they go by different names although they have much the same content.

Control	*Service*	*Meaning*
Individual accountability	Authentication	Determining who is responsible for a request or statement, whether it is "the loan rate is 10.3%" or "read file 'Pricing'" or "launch the rocket".
Separation of duty	Authorization (a broader term)	Determining who is trusted for a purpose: establishing a loan rate, reading a file, or launching a rocket. Specifically, trusting only two different people when they act together.
Auditing	Auditing	Recording who did what to whom, and later examining these records.
Recovery	—[4]	Finding out what damage was done and who did it, restoring damaged resources, and punishing the offenders.

The rest of this section discusses the policies in more detail and explains what the controls do and why they are useful.

1.1.1.1 Secrecy

Secrecy seeks to keep information from being disclosed to unauthorized recipients. The secrets might be important for reasons of national security (nuclear weapons data), law enforcement (the identities of undercover drug agents), competitive advantage (manufacturing costs or bidding plans), or personal privacy (credit histories).

[4] Technologists usually don't consider recovery to be a security service.

The most highly developed policies for secrecy reflect the concerns of the national security community, because this community has been willing to pay to get policies defined and implemented[5]. This policy is derived from the manual system for handling information that is critical to national security. In this system information is classified at levels of sensitivity and isolated compartments, and people are cleared for access to particular levels and/or compartments. Within each level and compartment, a person with an appropriate clearance must also have a "need to know" in order to get access. The policy is mandatory: elaborate procedures must be followed to downgrade the classification of information.

It is not hard to apply this policy in other settings. A commercial firm, for instance, might have access levels such as restricted, company confidential, unclassified, and categories such as personnel, medical, toothpaste division, etc. Significant changes are usually needed because the rules for downgrading are usually quite relaxed in commercial systems.

Another kind of secrecy policy, more commonly applied in civilian settings, is a discretionary one: every piece of information has an owner who can decide which other people and programs are allowed to see it. When new information is created, the creator chooses the owner. With this policy there is no way to tell where a given piece of information may flow without knowing how each user and program that can access the information will behave. It is still possible to have secrecy, but much more difficult to enforce it.

There is lots more to be said about privacy.

1.1.1.2 Integrity

Integrity seeks to maintain resources in a valid and intended state. This might be important to keep resources from being changed improperly (adding money to a bank account) or to maintain consistency between two parts of a system (double-entry bookkeeping). Integrity is not a synonym for accuracy, which depends on the proper selection, entry and updating of information.

The most highly developed policies for integrity reflect the concerns of the accounting and auditing community for preventing fraud. A classic example is a purchasing system. It has three parts: ordering, receiving, and payment. Someone must sign off on each step, the same person cannot sign off on two steps, and the records can only be changed by fixed procedures, e.g., an account is debited and a check written only for the amount of an approved and received order.

1.1.1.3 Accountability

In any real system there are many reasons why actual operation will not always reflect the intentions of the owners: people make mistakes, the system has errors, the system is vulnerable to certain attacks, the broad policy was not translated correctly into detailed specifications, the owners change their minds, etc. When things go wrong, it is necessary to know what has happened: who has had access to information and resources and what actions have been taken. This information is the basis for assessing damage, recovering lost information, evaluating vulnerabilities, and taking compensating actions outside the system such as civil suits or criminal prosecution.

[5] They are embodied in Department of Defense Directive 5200.28, also known as the Orange Book [ref DoD 1985], a document which specifies policy for safeguarding classified information in computer systems.

1.1.1.4 Availability

Availability[6] seeks to ensure that the system works promptly. This may be essential for operating a large enterprise (the routing system for long-distance calls, an airline reservation system) or for preserving lives (air traffic control, automated medical systems). Delivering prompt service is a requirement that transcends security, and computer system availability is an entire field of its own. Availability in spite of malicious acts and environmental mishaps, however, is often considered an aspect of security.

An availability policy is usually stated like this:

> On the average, a terminal shall be down for less than ten minutes per month.

A particular terminal (e.g., an automatic teller machine, a reservation agent's keyboard and screen, etc.) is *up* if it responds correctly within one second to a standard request for service; otherwise it is *down*. This policy means that the up time at each terminal, averaged over all the terminals, must be at least 99.9975%.

Such a policy covers all the failures that can prevent service from being delivered: a broken terminal, a disconnected telephone line, loss of power at the central computer, software errors, operator mistakes, system overload, etc. Of course, to be implementable it must be qualified by some statements about the environment, e.g. that power doesn't fail too often.

A security policy for availability usually has a different form, something like this:

> No inputs to the system by any user who is not an authorized administrator shall cause any other user's terminal to be down.

Note that this policy doesn't say anything about system failures, except to the extent that they can be caused by user actions. Also, it says nothing about other ways in which an enemy could deny service, e.g. by cutting a telephone line.

1.1.1.5 Individual accountability (authentication)

To answer the question "Who is responsible for this statement?" it is necessary to know what sort of entities can be responsible for statements. These entities are (human) users or (computer) systems, collectively called *principals*. A user is a person, but a system requires some explanation. A computer system is comprised of hardware (e.g., a computer) and perhaps software (e.g., an operating system). Systems implement other systems, so, for example, a computer implements an operating system which implements a database management system which implements a user query process. As part of authenticating a system, it may be necessary to verify that the system that implements it is trusted to do so correctly.

The basic service provided by authentication is information that a statement was made by some principal. Sometimes, however, there's a need to ensure that the principal will not later be able to claim that the statement was forged and he never made it. In the world of paper documents, this is the purpose of notarizing a signature; the notary provides independent and highly credible evidence, which will be convincing even after many years, that the signature is genuine and not forged. This aggressive form of authentication is called *non-repudiation*.

---

[6] Often called 'preventing denial of service'.

1.1.1.6 Authorization and separation of duty

Authorization determines who is trusted for a given purpose. More precisely, it determines whether a particular principal, who has been authenticated as the source of a request to do something, is trusted for that operation. Authorization may also include controls on the time at which something can be done (only during working hours) or the computer terminal from which it can be requested (only the one on the manager's desk).

It is a well established practice, called separation of duty, to insist that important operations cannot be performed by a single person, but require the agreement of (at least) two different people. This rule make it less likely that controls will be subverted because it means that subversion requires collusion.

1.1.1.7 Auditing

Given the reality that every computer system can be compromised from within, and that many systems can also be compromised if surreptitious access can be gained, accountability is a vital last resort. Accountability policies were discussed earlier --e.g., all significant events should be recorded and the recording mechanisms should be nonsubvertible. Auditing services support these policies. Usually they are closely tied to authentication and authorization, so that every authentication is recorded as well as every attempted access, whether authorized or not.

The audit trail is not only useful for establishing accountability. In addition, it may be possible to analyze the audit trail for suspicion patterns of access and so detect improper behavior by both legitimate users and masqueraders. The main problem however, is how to process and interpret the audit data. Both statistical and expert-system approaches are being tried [Lunt 1988].

1.1.1.8 Recovery

Need words here.

1.2 Choosing a policy

Ideally a comprehensive spectrum of security measures would ensure that the secrecy, integrity, accountability, and availability of computer-based systems are appropriately maintained. In practice it is not possible to make iron-clad guarantees. For most users and organizations, very high levels of defense against every possible threat are not cost-effective (or even feasible, because they interfere with normal operations). Each user must evaluate his needs and establish a suitable policy.

Organizations and people confront a wide range of risks every day: natural disaster, business fluctuations, insolvency, embezzlement, tort liability, and malicious mischief. Part of the job of management is to manage and contain those risks, balancing inherent business risk and risks taken explicitly in hopes of earning rewards against risks that can undermine the enterprise. At this abstract level there is little new about computer security. Yet the scale, speed, reach, mutability and invisibility of computer activity bring a need for a qualitatively different level of planning and attention to assure safe operations. Computers magnify both the difficulties and the opportunities for management controls and auditing.

How to do this:

Evaluate expected losses

Find vulnerabilities (in users as well as systems).
Vulnerability + attack = threat (bad event)
Threat + cost of bad event = risk
Risk * probability of threat = expected loss

Find countermeasures.
What to they cost?

If countermeasure is available and costs less than expected loss, then it makes sense to implement it.

Otherwise, make sure that adequate recovery is possible.

All this can't be done very precisely: quantitative risk assessment of a myriad of qualitatively different low-probability, high-impact risks has not been successful.

Ideally, controls will be instituted as the result of this kind of careful analysis. In practice, the most important consideration is what controls are available. For instance, customers appear to demand password-based authentication because it is available, not because analysis has shown that this relatively weak mechanism provides enough protection. This effect works in both directions: a service is not demanded if it isn't available, but once it becomes available somewhere, it soon becomes wanted everywhere.

Some consensus exists about rock-bottom security mechanisms. A recent informal survey shows that nearly everyone wants the ability to identify users and limit times and places of access, particularly over networks, and to watch for intrusion by logging attempts at invalid actions. Ad hoc virus checkers, well known in the personal computer market, are also demanded. On the other hand, there is no demand for system managers to be able to obtain positive confirmation that the software running on their systems today is the same as what was running yesterday. Such a simple analog of hardware diagnostics should be a rock-bottom requirement; probably it is not seen as such because vendors don't offer it.

Threats are different for different kinds of systems. To illustrate typical concerns, we consider the threats for some generic categories of computer application: strategic, commercial, operational, and general computing.

A strategic application is one that keeps high-value, perhaps unique information, the loss of which entails high risk. The main need is secrecy: outsiders must be kept from learning what's stored in a system, and insiders must be kept from leaking it. National defense applications are the classic examples: loss of secret could result in loss of lives or even a country. Risks are not so great in business, where aggrieved firms may resort to the courts.

A commercial system is primarily concerned with the maintenance of voluminous similar records, as in banking, personnel, inventory or purchasing systems. The main need is integrity of assets and accountability.. Outsiders must be kept from stealing, and insiders must be kept from misapplying money, goods, or services. Commercial systems have little opportunity to automatically correct bad data; it comes to light only when it finally affects some person.

Operational applications are less concerned with records are more with the proper flow of activity, as in air traffic control, telephone switching, chemical process control, or production scheduling. These applications typically run in a closed-loop fashion: telephone calls, radar echoes, or reactor temperatures are monitored continually, and occasional errors can be corrected in the normal course of events. Thus integrity of the working records is not so important, but the

integrity of the overall process is critical; this is governed by programs and configuration data which describe the system environment. Availability is likely to be important also.

In a general computer system the computer serves as a resource which may be put to many uses, running a mix of applications. This arrangement capitalizes on the synergy of close communication among distinct applications and the convenience of using a single medium for diverse purposes. Ready communication is more important than secrecy, and availability is not critical. Integrity against disasters and outside interference is important, since months of work can be lost or become untrustworthy. 'Development' systems, where software is created and debugged for use elsewhere, fall into this category, though their security requirements often follow from those of the ultimate application.

Security policies for general computation tend to be discretionary because of the variety of applications and the need for communication. The other classes of system often need mandatory policies so that management can exercise more control at the price of flexibility.

It is common to built strategic, commercial, or operational applications on general systems, because no single application can justify all the development cost of a complete computer system. Thus there is a tension between the general purposes of computers and the special requirements of typical classes of applications. But as we have seen, broad classes of applications have greatly overlapping needs, so systems can justifiably be expected to cater for a wide spectrum of needs. This is especially true because the various classes of system shade into each other. A telephone switching system (operational) gathers billing records (commercial). Battlefield management (operational) depends on tactical plans (strategic) and also on materiel logistics (commercial). A management information system may contain takeover plans (strategic) and run electronic mail (operational).

2 Technology

This section describes the technology for protecting information and other resources controlled by computer systems, and explains how it can be used to make such systems secure. It is a reasonably complete survey of the entire subject at a high level. It explains the essential technical ideas, gives the major properties of the techniques that are currently known, and tells why they are important. A reader of this section will be equipped to understand the impact of technology on secure computer systems.

Section 3 goes into more detail on a number of topics which are either fundamental or of special current interest. Here the reader can learn how some important things work, gain some insight into why they don't work perfectly, and get an idea of how they may develop in the next few years.

We discuss the technology of computer security under two major headings:

What do we mean by security?

How do we get it, and how do we know when we have it?

The first deals with *specification* of security and the *services* that computer systems provide to support security. The second deals with *implementation* of security, and in particular how we establish confidence that a system will actually provide the security the specifications promise. Each topic gets space according to its importance for the overall goal of providing computer

security, and not according to how much work has already been done on that topic. To make up for this there is a concluding section that points out what technology is already in wide use, what is ready for deployment, and what needs more research. The conclusion also highlights the topics that are especially relevant to distributed and embedded systems.

The careful analysis here may seem tedious, but it is the only known way to ensure the security of something as complicated as a computer system. Security is like ecology: It is not enough to focus on the problem that caused trouble last month, because as soon as one weakness is repaired, an adversary will shift his attack elsewhere.

2.1 Specification vs. implementation

The distinction between *what* the system does and *how* it does it, between specification and implementation, is basic to the design and analysis of computer systems. A specification for a system is the meeting point between the customer and the builder. It says what the system is supposed to do. This is important to the builder, who must ensure that what the system actually does matches what it is supposed to do. It is equally important to the customer, who must be confident that what the system is supposed to do matches what he wants. It is especially critical to know exactly and completely what the system is supposed to do about security, because any mistake can be exploited by a malicious adversary.

Specifications can be written at many levels of detail and with many degrees of formality. Broad and informal specifications of security are called security *policies*[7]; they are discussed in section 1. Here are some examples of policies:

> Secrecy: Information shall be disclosed only to people authorized to receive it.
>
> Integrity: Data shall be modified only according to established procedures and at the direction of properly authorized people.

We can separate the part of a specification that is relevant to security from the rest. Usually the whole specification is much bigger than the security-relevant part. For example, it usually says a good deal about performance, i.e., how much it should cost or how long it should take to do this or that. But if secrecy and integrity are the security policies, performance isn't relevant to security, because the system can provide secrecy and integrity regardless of how well or badly it performs. Since we are only interested in security here, from now on when we say 'specification' we mean only the security-relevant part.

A secure system is one that meets the (security) specifications. Since there are many different specifications, there can't be any absolute notion of a secure system. An example from a related field clarifies this point. We say that an action is legal if it meets the requirements of the law. Since there are many different sets of laws in different jurisdictions, there can't be any absolute notion of a legal action; what is legal under the laws of Britain may be illegal in the US.

A system that is believed to be secure is *trusted*. Of course, a trusted system must be trusted for something; in the context of this report it is trusted to meet the security specifications. In some other context it might be trusted to control a shuttle launch or to retrieve all the 1988 court opinions dealing with civil rights. People concerned about security have tried to take over the word 'trusted' to describe their concerns; they have had some success because security is the area

[7] Policies are often called 'requirements'; sometimes the word 'policy' is reserved for a broad statement and 'requirement' is used for a more detailed one

in which the most effort has been made to specify and build trustworthy systems. We will adopt this usage, while recognizing that in a broader context 'trusted' may have many other meanings.

Policies express a general intent. Of course, they can be more detailed than the very general ones given as examples above; for instance, here is a refinement of the first policy:

> Salary secrecy: Individual salary information shall only be disclosed to the employee, his superiors, and authorized personnel people.

But whether general or specific, policies contain terms which are not precisely defined, so it isn't possible to reliably tell whether a system satisfies them. Furthermore, they specify the behavior of people and of the physical environment as well as the behavior of machines, so it isn't possible for a computer system alone to satisfy them. Technology for security addresses these problems by providing methods for:

> Integrating a computer system into a larger system, comprising people and a physical environment as well as computers, that meets its security policies.
>
> Giving a precise specification, called a security *model*, for the security-relevant behavior of the computer system.
>
> Building a system that meets the specifications out of components that provide and use security *services*.
>
> Establishing *trust*[8] that a system actually does meet the specifications.

This is a tall order, and at the moment we only know how to fill some of it. The first three points are discussed in the next section on specifications, the last two in the following section on implementation. Services are discussed in both sections to explain both the functions being provided and how they are implemented.

2.2 Specification: Policies, models, and services

This section deals with the specification of security. It is based on the taxonomy of known policies given in Section 2; fortunately, there are only a few of them. For each policy there is a corresponding security model, which is a precise specification of how a computer system should behave as part of a larger system that implements the policy. Finally, an implementation of the model needs some components that provide particular security services. Again, only a few of these have been devised, and they seem to be sufficient to implement all the models.

We can illustrate the ideas of policy and model with the simple example of a traffic light; in this example, safety plays the role of security. The light is part of a system which includes roads, cars, and drivers. The safety policy for the complete system is that two cars should not collide. This is refined into a policy that traffic must not move on two conflicting roads at the same time. This policy is translated into a safety model for the traffic light itself (which plays the role of the computer system within the complete system): two green lights may never appear in conflicting traffic patterns simultaneously. This is a pretty simple specification. Observe that the complete specification for a traffic light is much more complex; it includes the ability to set the duration of the various cycles, to synchronize with other traffic lights, to display different combinations of arrows, etc. None of these details, however, is critical to the safety of the system, because cars won't collide if they aren't met. Observe also that for the whole system to meet its safety policy

[8] often called 'assurance'.

the light must be visible to the drivers, and they must obey its rules. If the light remains red in all directions it will meet its specification, but the drivers will lose patience and start to ignore it, so that the entire system may not remain safe.

An ordinary library affords a more complete example, which illustrates several aspects of computer system security in a context that does not involve computers. It is discussed in section 3.1.1.1 below.

2.2.1 Policies

A security policy is an informal specification of the rules by which people are given access to read and change information and to use resources. Policies naturally fall under four major headings:

secrecy: controlling who gets to read information;

integrity: controlling how information changes or resources are used;

accountability: knowing who has had access to information or resources;

availability: providing prompt access to information and resources.

Section 1 describes these policies in detail and discusses how an organization that uses computers can formulate a security policy by drawing elements from all of these headings. Here we summarize this material and supplement it with some technical details.

Security policies for computer systems generally reflect long-standing policies for security of systems that don't involve computers. In the case of national security these are embodied in the classification system; for commercial computing they come from established accounting and management control practices. More detailed policies often reflect new threats. Thus, for example, when it became known that Trojan Horse software (see section 2.3.1.3) can disclose sensitive data without the knowledge of an authorized user, policies for mandatory access control and closing covert channels were added to take this threat into account.

From a technical viewpoint, the most highly developed policies are for secrecy. They reflect the concerns of the national security community and are embodied in Department of Defense Directive 5200.28, also known as the Orange Book [Department of Defense 1985], a document which specifies policy for safeguarding classified information in computer systems.

The DoD computer security policy is based on *security levels*. Given two levels, one may be lower than the other, or they may be incomparable. The basic idea is that information can never leak to a lower level, or even an incomparable level. Once classified, it stays classified no matter what the users and application programs do within the system.

A security level consists of an access level (one of top secret, secret, confidential, or unclassified) and a set of categories (e.g., nuclear, NATO, etc.). The access levels are ordered (top secret highest, unclassified lowest). The categories are not ordered, but sets of categories are ordered by inclusion: one set is lower than another if every category in the first is included in the second. One security level is lower than another if it has an equal or lower access level and an equal or lower set of categories . Thus, (unclassified; NATO) is lower than both (unclassified; nuclear, NATO) and (secret; NATO). Given two levels, it's possible that neither is lower than the other. Thus, (secret; nuclear) and (unclassified; NATO) are incomparable.

Every piece of information has a security level. Information flows only upward: information at one level can only be derived from information at equal or lower levels, never from information which is at a higher level or incomparable. If some information is computed from several inputs, it has a level which is at least as high as any of the inputs. This rule ensures that if some information is stored in the system, anything computed from it will have an equal or higher level. Thus the classification never decreases.

The policy is that a person is cleared to some security level, and can only see information at that level or lower. Since anything he sees can only be derived from other information at its level or lower, the result is that what he sees can depend only on information in the system at his level or lower. This policy is mandatory: except for certain carefully controlled downgrading or declassification procedures, neither users nor programs in the system can break the rules or change the security levels.

As Section 1 explains, both this and other secrecy policies can also be applied in other settings.

The most highly developed policies for integrity reflect the concerns of the accounting and auditing community for preventing fraud. The essential notions are individual accountability, separation of duty, and standard procedures.

Another kind of integrity policy is derived from the information flow policy for secrecy applied in reverse, so that information can only be derived from other information of higher integrity [Biba 1975]. This policy has not been applied in practice.

Integrity policies have not been studied as carefully as secrecy policies, even though some sort of integrity policy governs the operation of every commercial data processing system. Work in this area [Clark and Wilson 1987] lags work on secrecy by about 15 years.

Policies for accountability have usually been formulated as part of secrecy or integrity policies. This subject has not had independent attention.

Very little work has been done on security policies for availability.

2.2.2 Models

In order to engineer a computer system that can be used as part of a larger system that implements a security policy, and to decide unambiguously whether it meets its specification, the informal, broadly stated policy must be translated into a precise *model*. A model differs from a policy in two ways:

> It describes the desired behavior of the computer system, not of the larger system that includes people.
>
> It is precisely stated in a formal language that removes the ambiguities of English and makes it possible, at least in principle, to give a mathematical proof that a system satisfies the model.

There are two models in wide use. One, based on the DoD computer security policy, is the *flow* model; it supports a certain kind of secrecy policy. The other, based on the familiar idea of stationing a guard at an entrance, is the *access control* model; it supports a variety of secrecy, integrity and accountability policies. There aren’t any models that support availability policies.

2.2.2.1 Flow

The flow model is derived from the DoD policy described earlier. In this model [Denning 1976] every piece of data in the system is held in a container called an *object*. Each object has a security level[9]. An object's level gives the security level of the data it contains. Data in one object is allowed to affect another object only if the source object's level is lower than or equal to the destination object's level. All the data within a single object has the same level and hence can be manipulated freely.

The purpose of this model is to ensure that information at a given security level flows only to an equal or higher level. Data is not the same as information; for example, an encrypted message contains data, but it conveys no information unless you know the encryption key or can break the encryption system. Unfortunately, data is the only thing the computer can understand. By preventing an object at one level from being affected in any way by data that is not at an equal or lower level, the flow model ensures that information can flow only to an equal or higher level inside the conputer system. It does this very conservatively and thus forbids many actions which would not in fact cause any information to flow improperly.

A more complicated version of the flow model (which is actually the basis of the rules in the Orange Book) separates objects into active ones called 'subjects' which can initiate operations, and passive ones (just called objects) which simply contain data, such as a file, a piece of paper, or a display screen. Data can only flow between an object and a subject; flow from object to subject is called a read operation, and flow from subject to object is called a write operation. Now the rules are that a subject can only read from an object at an equal or lower level, and can only write to an object at an equal or higher level.

Not all possible flows in the system look like read and write operations. Because the system is sharing resources among objects at different levels, it is possible for information to flow on 'covert channels' [Lampson 1973]. For example, a high level subject might be able to send a little information to a low level one by using up all the disk space if it discovers that a surprise attack is scheduled for next week. When the low level subject finds itself unable to write a file, it has learned about the attack (or at least gotten a hint). To fully realize the flow model it is necessary to find and close all the covert channels.

To fit this model of the computer system into the real world, it is necessary to account for the people. A person is cleared to some level. When he identifies himself to the system as a user present at some terminal, he can set the terminal's level to any equal or lower level. This ensures that the user will never see information at a higher level than his clearance. If the user sets the terminal level lower than his clearance, he is trusted not to take high level information out of his head and introduce it into the system.

Although this is not logically required, the flow model has always been viewed as mandatory: except for certain carefully controlled downgrading or declassification procedures, neither users nor programs in the system can break the flow rule or change levels. No real system can strictly follow this rule, since procedures are always needed for declassifying data, allocating resources, introducing new users, etc. The access control model is used for these purposes.

[9] often called its 'label'.

2.2.2.2 Access control

The access control model is based on the idea of stationing a guard in front of a valuable resource to control who can access it. It organizes the system into

> *objects*: entities which respond to operations by changing their state, providing information about their state, or both;
>
> *subjects*: active objects which can perform operations on objects;
>
> *operations*: the way that subjects interact with objects.

The objects are the resources being protected; an object might be a document, a terminal, or a rocket. There is a set of rules which specify, for each object and each subject, what operations that subject is allowed to perform on that object. A 'reference monitor' acts as the guard to ensure that the rules are followed [Lampson 1985].

There are many ways to organize a system into subjects, operations, and objects. Here are some examples:

Subject	*Operation*	*Object*
Smith	Read File	"1990 pay raises"
White	Send "Hello"	Terminal 23
Process 1274	Rewind	Tape unit 7
Black	Fire three rounds	Bow gun
Jones	Pay invoice 432567	Account Q34

There are also many ways to express the access rules. The two most popular are to attach to each subject a list of the objects it can access (called a 'capability list'), or to attach to each object a list of the subjects that can access it (called an 'access control list'). Each list also identifies the operations that are allowed.

Usually the access rules don't mention each operation separately. Instead they define a smaller number of 'rights'[10] (e.g., read, write, and search) and grant some set of rights to each (subject, object) pair. Each operation in turn requires some set of rights. In this way there can be a number of different operations for reading information from an object, all requiring the read right.

One of the operations on an object is to change which subjects can access the object. There are many ways to exercise this control, depending on what the policy is. With a discretionary policy each object has an owner who can decide without any restrictions who can do what to the object. With a mandatory policy, the owner can make these decisions only inside certain limits. For example, a mandatory flow policy allows only a security officer to change the security level of an object, and the flow model rules limit access. The owner of the object can usually apply further limits at his discretion.

The access control model leaves it open what the subjects are. Most commonly, subjects are users, and any active entity in the system is treated as acting on behalf of some user. In some systems a program can be a subject in its own right. This adds a lot of flexibility, because the program can implement new objects using existing ones to which it has access. Such a program is called a 'protected subsystem'.

[10] often called 'permissions'.

The access control model can be used to realize both secrecy and integrity policies, the former by controlling read operations, and the latter by controlling writes and other operations that change the state, such a firing a gun. To realize an flow policy, assign each object and subject a security level, and allow read and write operations only according to the rules of the flow model. This scheme is due to Bell and LaPadula, who called the rule for reads the simple security property and the rule for writes the *-property [Bell-LaPadula 1976].

The access control model also supports accountability, using the simple notion that every time an operation is invoked, the identity of the subject and the object as well as the operation should be recorded in an audit trail which can later be examined. There are practical difficulties caused by the fact that the audit trail gets too big.

2.2.3 Services

This section describes the basic security services that are used to build systems satisfying the policies discussed earlier. These services directly support the access control model, which in turn can be used to support nearly all the policies we have discussed:

Authentication: determining who is responsible for a given request or statement[11], whether it is "the loan rate is 10.3%" or "read file 'Memo to Mike'" or "launch the rocket".

Authorization: determining who is trusted for a given purpose, whether it is establishing a loan rate, reading a file, or launching a rocket.

Auditing: recording each operation that is invoked along with the identity of the subject and object, and later examining these records.

Given these services, building a reference monitor to implement the access control model is simple. Whenever an operation is invoked, it uses authentication to find out who is requesting the operation and then uses authorization to find out whether the requester is trusted for that operation. If so, it allows the operation to proceed; otherwise, it cancels the operation. In either case, it uses auditing to record the event.

2.2.3.1 Authentication

To answer the question "Who is responsible for this statement?" it is necessary to know what sort of entities can be responsible for statements; we call these entities *principals*. It is also necessary to have a way of naming the principals which is consistent between authentication and authorization, so that the result of authenticating a statement is meaningful for authorization.

A principal is a (human) user or a (computer) system. A user is a person, but a system requires some explanation. A system is comprised of hardware (e.g., a computer) and perhaps software (e.g., an operating system). A system can depend on another system; for example, a user query process depends on a database management system which depends on an operating system which depends on a computer. As part of authenticating a system, it may be necessary to verify that any system that it depends on is trusted to work correctly.

In order to express trust in a principal (e.g., to specify who can launch the rocket) you must be able to give the principal a name. The name must be independent of any information that may change without any change in the principal itself (such as passwords or encryption keys). Also, it

[11] That is, who caused it to be made, in the context of the computer system; legal responsibility is a different matter.

must be meaningful to you, both when you grant access and later when it is time to review the trust being granted to see whether it is still what you want. A naming system must be:

Complete: every principal has a name; it is difficult or impossible to express trust in a nameless principal.

Unambiguous: the same name does not refer to two different principals; otherwise you can't know who is being trusted.

Secure: you can easily tell what other principals you must trust in order to authenticate a statement from a named principal.

In a large system naming must be decentralized. Furthermore, it's not possible to count on a single principal that is trusted by every part of the system.

It is well known how to organize a decentralized naming system in a hierarchy, following the model of a tree-structured file system like the one in Unix or MS-DOS. The CCITT X.500 standard for naming defines such a hierarchy [CCITT 1989b]; it is meant to be suitable for naming every principal in the world. In this scheme an individual can have a name like US/GOV/State/Kissinger. Such a naming system can be complete; there is no shortage of names, and registration can be made as convenient as desired. It is unambiguous provided each directory is unambiguous.

The CCITT X.509 standard defines a framework for authenticating a principal with an X.500 name; the section on authentication techniques below discusses how this is done [CCITT 1989b]. An X.509 authentication may involve more than one agent. For example, agent A may authenticate agent B, who in turn authenticates the principal.

A remaining issue is exactly who should be trusted to authenticate a given name. Typically, principals trust agents close to them in the hierarchy. A principal is less likely to trust agents farther from it in the hierarchy, whether those agents are above, below, or in entirely different branches of the tree. If a system at one point in the tree wants to authenticate a principal elsewhere, and there is no one agent that can certify both, then the system must establish a chain of trust through multiple agents. The simplest such chain involves all the agents in the path from the system, up through the hierarchy to the first ancestor that is common to both the system and the principal, and then down to the principal. Such a chain will always exist if each agent is prepared to authenticate its parent and children. This scheme is simple to explain; it can be modified to deal with renaming and to allow for shorter authentication paths between cooperating pairs of principals.

Since systems as well as users can be principals, systems as well as users must be able to have names.

Often a principal wants to act with less than his full authority, in order to reduce the damage that can be done in case of a mistake. For this purpose it is convenient to define additional principals called 'roles', to provide a way of authorizing a principal to play a role, and to allow the principal to make a statement using any role for which he is authorized. For example, a system administrator might have a 'normal' role and a 'powerful' role. The authentication service then reports that the statement was made by the role rather than by the original principal, after verifying both that the statement comes from the original principal and that he is authorized to play that role.

In general trust is not simply a matter of trusting a single user or system principal. It is necessary to trust the (hardware and software) systems through which that user is communicating. For example, suppose that a user Alice running on a workstation Bob is entering a transaction on a transaction server Charlie which in turn makes a network access to a database machine Dan. Dan's authorization decision may need to take account not just of Alice, but also of the fact the Bob and Charlie are involved and must be trusted. Some of these issues do not arise in a centralized system, where a single authority is responsible for all the authentication and provides the resources for all the applications, but even in a centralized system an operation on a file, for example, is often invoked through an application such as a word processing program which is not part of the base system and perhaps should not be trusted in the same way.

Rather than trusting all the intermediate principals, we may wish to base the decision about whether to grant access on what intermediaries are involved. Thus we want to grant access to a file if the request comes from the user logged in on the mainframe, or through a workstation located on the second floor, but not otherwise.

To express such rules we need a way to describe what combinations of users and intermediaries can have access. It is very convenient to do this by introducing a new, compound principal to represent the user acting through intermediaries. Then we can express trust in the compound principal exactly as in any other. For example, we can have principals "Smith ON Workstation 4" or "Alice ON Bob ON Charlie" as well as "Smith" or "Alice". The names "Workstation 4", "Bob" and "Charlie" identify the intermediate systems just as the names "Smith" and "Alice" identify the users.

How do we authenticate such principals? When Workstation 4 says "Smith wants to read file 'Pay raises'", how do we know

first, that the request is really from that workstation and not somewhere else;

second, that it is really Smith acting through Workstation 4, and not Jones or someone else?

We answer the first question by authenticating the intermediate systems as well as the users. If the resource and the intermediate are on the same machine, the operating system can authenticate the intermediate to the resource. If not, we use the cryptographic methods discussed in the section below on secure channels.

To answer the second question, we need some evidence that Smith has *delegated* to Workstation 4 the authority to act on his behalf. We can't ask for direct evidence that Smith asked to read the file--if we could have that, then he wouldn't be acting through the workstation. We certainly can't take the workstation's word for it; then it could act for Smith no matter who is really there. But we can demand a statement that we believe is from Smith, asserting that Workstation 4 can speak for him (probably for some limited time, and perhaps only for some limited purposes). Given

Smith says: "Workstation 4 can act for me"

Workstation 4 says "Smith says to read the file Pay raises"

we can believe

Smith ON Workstation 4 says "Read the file Pay raises"

How can we authenticate the delegation statement from Smith, "Workstation 4 can act for me", or from Jones, "TransPosting can act for me"? Again, if Jones and the database file are on the

same machine, Jones can tell the operating system, which he does implicitly by running the TransPosting application, and the system can pass his statement on to the file system. Since Smith is not on the same machine, he needs to use the cryptographic methods described below.

The basic service provided by authentication is information that a statement was made by some principal. Sometimes, however, we would like a guarantee that the principal will not later be able to claim that the statement was forged and he never made it. In the world of paper documents, this is the purpose of notarizing a signature; the notary provides independent and highly credible evidence, which will be convincing even after many years, that the signature is genuine and not forged. This aggressive form of authentication is called ‘non-repudiation’. It is accomplished by a digital analog of notarizing, in which a trusted authority records the signature and the time it was made.

2.2.3.2 Authorization

Authorization determines who is trusted for a given purpose, usually for doing some operation on an object. More precisely, it determines whether a particular principal, who has been authenticated as the source of a request to do an operation on an object, is trusted for that operation on that object.

In general, authorization is done by associating with the object an *access control list* or ACL which tells which principals are authorized for which operations. The authorization service takes a principal, an ACL, and an operation or a set of rights, and returns yes or no. This way of providing the service leaves the object free to store the ACL in any convenient place and to make its own decisions about how different parts of the object are protected. A data base object, for instance, may wish to use different ACLs for different fields, so that salary information is protected by one ACL and address information by another, less restrictive one.

Often several principals have the same rights to access a number of objects. It is both expensive and unreliable to repeat the entire set of principals for each object. Instead, it is convenient to define a group of principals, give it a name, and give the group access to each of the objects. For instance, a company might define the group “executive committee”. The group thus acts as a principal for the purpose of authorization, but the authorization service is responsible for verifying that the principal actually making the request is a member of the group.

Our discussion of authorization has been mainly from the viewpoint of an object, which must decide whether a principal is authorized to invoke a certain operation. In general, however, the subject doing the operation may also need to verify that the system implementing the object is authorized to do so. For instance, when logging in over a telephone line, a user may want to be sure that he has actually reached the intended system and not some other, hostile system which may try to spoof him. This is usually called ‘mutual authentication’, although it actually involves authorization as well: statements from the object must be authenticated as coming from the system that implements the object, and the subject must have access rules to decide whether that system is authorized to do so.

2.2.3.3 Auditing

Given the reality that every computer system can be compromised from within, and that many systems can also be compromised if surreptitious access can be gained, accountability is a vital last resort. Accountability policies were discussed earlier --e.g., all significant events should be recorded and the recording mechanisms should be nonsubvertible. Auditing services support

these policies. Usually they are closely tied to authentication and authorization, so that every authentication is recorded as well as every attempted access, whether authorized or not.

The audit trail is not only useful for establishing accountability. In addition, it may be possible to analyze the audit trail for suspicion patterns of access and so detect improper behavior by both legitimate users and masqueraders. The main problem however, is how to process and interpret the audit data. Both statistical and expert-system approaches are being tried [Lunt 1988].

2.3 Implementation: The trusted computing base

In this section we explore how to build a system that meets the kind of security specifications discussed earlier, and how to establish confidence that it does meet them.

Systems are built out of components; we also say that the system depends on its components. This means that the components have to work (i.e., meet their specifications) for the system to work (i.e., meet its specification). If this were not true, then the system would work no matter what the components do, and we wouldn't say that it depends on them or is built out of them.

Each component is itself a system with specifications and implementation, so it's systems all the way down. For example, a distributed system depends on a network, workstations, servers, mainframes, printers, etc. A workstation depends on a display, keyboard, disk, processor, network interface, operating system, spreadsheet application, etc. A processor depends on integrated circuit chips, wires, circuit boards, connectors, etc. A spreadsheet depends on display routines, an arithmetic library, a macro language processor, etc., and so it goes down to the basic operations of the programming language, which in turn depend on the basic operations of the machine, which in turn depend on changes in the state of the chips, wires, etc. A chip depends on adders, memory cells, etc., and so it goes down to the electrons and photons, whose behavior is described by quantum electrodynamics.

A component is trusted if it has to work for the system to meet its specification. The set of trusted hardware and software components is called the *trusted computing base* or TCB. If a component is in the TCB, so is every component that it depends on, because if they don't work, it's not guaranteed to work either. Don't forget that we are only concerned with security, so the trusted components need to be trusted only for security, even though much of what we have to say applies more generally.

From this perspective, there are three aspects of implementing a secure system:

- Keeping the trusted computing base as small as possible, by having as few trusted components as possible and making the implementation of each one as simple as possible.
- Assurance that each trusted component is trustworthy, i.e., does its job, by
  - applying systematic, and preferable formal, methods to validate each component and to find the components it depends on (each one of which must also be trusted),
  - keeping the specifications as simple as possible, and
  - regularly checking that the TCB is what you think it is (hardware and software not corrupted, authorization rules as intended, authentication methods not compromised).
- Techniques for implementing the individual components, such as cryptography and access control lists.

A system depends on more than its hardware and software. The physical environment and the people who use, operate, and manage it are also components of the system. Some of them must also be trusted. For example, if the power fails the system may stop providing service; thus the power source must be trusted for availability. Another example: every system has security officers who set security levels, authorize users, etc.; they must be trusted to do this properly. Yet another: the system may disclose information only to authorized users, but they must be trusted not to publish the information in the newspaper. Thus the security of the entire system must be evaluated, using the basic principles of analyzing dependencies, minimizing the number and complexity of trusted components, and carefully analyzing each one. The rest of this section deals only with the TCB, but it is only part of the story.

The basic method for keeping the TCB small is to make it a *kernel* that contains only functions that are needed for security, and separate out all the other functions into untrusted components. For example, an elevator has a very simple braking mechanism whose only job is to stop the elevator if it starts to move faster than a fixed maximum speed, no matter what else goes wrong. The rest of the elevator control mechanism may be very complex, involving scheduling of several elevators, responding to requests from various floors, etc., but none of this must be trusted for safety, because the braking mechanism doesn't depend on anything else. The braking mechanism is the safety kernel.

The purchasing system mentioned in the earlier discussion of integrity policies is another example. A large and complicated word processor may be used to prepare orders, but the TCB can be limited to a simple program that displays the completed order and asks the user to confirm it. An even more complicated database system may be used to find the order that corresponds to an arriving shipment, but the TCB can be limited to a simple program that displays the received order and a proposed payment authorization and asks the user to confirm them. If the order and authorization can be digitally signed (using methods described later), even the components that store them need not be in the TCB.

The basic method for finding dependencies is careful analysis of how each step in building and executing a system is carried out. Ideally we then get assurance for each system by a formal mathematical proof that the system satisfies its specification provided all its components do. In practice such proofs are only sometimes feasible, because it is hard to formalize the specifications and to carry out the proofs. The state of the art is described in 3.1.4. In practice, we get assurance by a combination of relying on components that have worked for lots of people, trusting implementors not to be malicious, carefully writing specifications for components, and carefully examining implementations for dependencies and errors.

These methods are bound to have flaws. Because there are so many bases to cover, and every one is critical to true security, there are bound to be mistakes. Hence two other important parts of assurance are redundant checks like the security perimeters discussed below, and methods for recovering from failures such as audit trails and backup databases.

We now proceed to discuss the main components of a trusted computing base, under the major headings of computing and communications. This division reflects the fact that a modern distributed system is made up of computers that can be analyzed individually, but must communicate with each other quite differently than they do internally.

2.3.1 Computing

This part of the TCB includes the application programs, the operating system that they depend on, and the hardware that both depend on.

2.3.1.1 Hardware

Since software consists of instructions which must be executed by hardware, the hardware must be part of the TCB. Security requires, however, very little from the hardware beyond the storage and computing functions needed by all software, nothing more than a "user state" in which a program can access only the ordinary computing instructions and a restricted part of the memory, as well as a "supervisor state" in which a program can access every part of the hardware. There is no need for fancier hardware features; at best they may improve performance, and usually they don't even do that because they add complications that slow down basic hardware operations that are executed much more frequently.

The only essential thing, then, is to have simple hardware that is trustworthy. For most purposes the ordinary care that competent engineers take to make the hardware work is good enough. It's possible to get higher assurance by using formal methods to design and verify the hardware; this has been done in several projects, of which the Viper verified microprocessor chip is typical [Cullyer 1989]. There is a mechanically checked proof that the Viper chip's gate-level design implements its specification. Viper pays the usual price for high assurance: it is several times slower than ordinary microprocessors built at the same time. It is described in more detail in section 3.1.2.2.1.

Another approach to hardware support for high assurance is to provide a separate, simple processor with specialized software to implement the basic access control services. If this hardware controls the computer's memory access mechanism and forces all input/output data to be encrypted, that is enough to keep the rest of the hardware and software out of the TCB. This approach has been pursued in the Lock project, which is described in more detail in section 3.1.2.2.2.

Unlike the other components of a computing system, hardware is physical and has physical interactions with the environment. For instance, someone can open a cabinet containing a computer and replace one of the circuit boards. Obviously if this is done with malicious intent, all bets are off about the security of the computer. It follows that physical security of the hardware must be assured. There are less obvious physical threats. In particular, computer hardware involves changing electric and magnetic fields, and therefore generates electromagnetic radiation[12] as a byproduct of normal operation. Because it can be a way for information to be disclosed, secrecy may require this radiation to be controlled. Similarly, radiation from the environment can affect the hardware.

2.3.1.2 Operating system

The job of an operating system is to share the hardware among several application programs and to provide generic security services so that the applications don't need to be part of the TCB. This is good because it keeps the TCB small, since there is only one operating system but many applications. Within the operating system itself the same idea can be used to divide it into a 'kernel' which is part of the TCB and other components which are not [Gasser 1988]. It is well known how to do this.

The generic services are authorization services for the basic objects that the system implements: virtual processors as subjects, and files and communication devices such as terminals as objects. The operating system can enforce various security models for these objects, which may be enough to satisfy the security policy. In particular it can enforce an flow model, which is enough

[12] often called 'emanations'.

for the DoD secrecy policy, as long as it is enough to keep track of security levels at the coarse granularity of whole files.

To enforce an integrity policy like the purchasing system policy described earlier, there must be some trusted applications to handle functions like approving orders. The operating system must be able to treat these applications as principals, so that they can access objects that the untrusted applications running on behalf of the same user cannot access. Such applications are called 'protected subsystems'.

2.3.1.3 Applications

Ideally applications should not be part of the TCB, since there are many of them, they are often large and complicated, and they tend to come from a variety of sources that are difficult to police. Unfortunately, attempts to build electronic mail or database applications on top of an operating system that enforces flow have run into difficulties. It is necessary to use a different operating system object for information at each security level, and often these objects are too big and expensive. And for an integrity policy it is always necessary to trust some application code. It seems that the best we can do is to apply the kernel method again, putting the code that must be trusted into separate components which are protected subsystems. The operating system must support this [Boebert 1985]. See section 3.1.3 for more on this subject.

In many systems any application program running on behalf of a user has the full access to all objects that the user has. This is considered acceptable on the assumption that the program, even if not trusted to always do the right thing, is unlikely to do an intolerable amount of damage. What if the program is not just wrong, but malicious? Such a program, which appears to do something useful but has hidden within it the ability to cause serious damage, is called a 'Trojan horse' [Thompson 1984]. When the Trojan horse runs, it can do a lot of damage: delete files, corrupt data, send a message with the user's secrets to another machine, disrupt the operation of the host, waste machine resources, etc. There are many ways to package a Trojan horse: the operating system, an executable program, a shell command file, a macro in a spreadsheet or word processing program are only a few of the possibilities. It is even more dangerous if the Trojan horse can also make copies of its evil genius. Such a program is called a virus. Because it can spread fast in a computer network or by copying disks, it can be a serious threat. There have been several examples of viruses that have infected thousands of machines [Computers & Security 1988, Dewdney 1989]. Section 3.2.4 gives more details and describes countermeasures.

2.3.2 Communications

Methods for dealing with communications and distributed systems security are less well developed than those for stand-alone centralized systems; distributed systems are both newer and more complex. Even though there is less of a consensus about methods for distributed systems, we describe one way of thinking about them, based on suitable functions inside a TCB built up of trusted code on the various distributed components. We believe that distributed systems are now well enough understood that these approaches should also become recognized as effective and appropriate in achieving security.

The TCB for communications has two important aspects: *secure channels* for communication among the various parts of a system, and *security perimeters* for restricting communication between one part of a system and the rest.

2.3.2.1 Secure Channels

We can describe a distributed system as a set of computers which communicate with each other quite differently than they operate internally. More generally, the access control model describes the working of a system in terms of requests for operations from a subject to an object and corresponding responses. Either way, it makes sense to study the communication separately from the computers, subjects, or objects.

Secure communication has one or both of the properties:

(1) Integrity: You can know who originally created a message you receive.

(2) Secrecy: You can know who can read a message you send.

The concept of a secure channel is an abstraction of these properties. A channel is a path by which two or more principals communicate. A secure channel may be a physically protected path (e.g., a physical wire, a disk drive and associated disk, or memory protected by hardware and an operating system), or a logical path secured by encryption. A channel need not be real time: a message sent on a channel may be read much later, for instance if it is stored on a disk. A secure channel provides integrity, confidentiality, or both. The process of finding out who can send or receive on a secure channel is called authenticating the channel; once a channel has been authenticated, statements and requests arriving on it are also authenticated.

Typically the secure channels between subjects and objects inside a computer are physically protected: the wires in the computer are assumed to be secure, and the operating system protects the paths by which programs communicate with each other. This is one aspect of a broader point: every component of a physical channel is part of the TCB and must meet a security specification. If the wire connects two computers it may be difficult to secure physically, especially if the computers are in different buildings.

To keep such wires out of the TCB we resort to encryption, which makes it possible to have a channel whose security doesn't depend on the security of any wires or intermediate systems through which the bits of the message are passed. Encryption works by computing from the data of the original message, called the 'cleartext', some different data, called the 'ciphertext', which is actually transmitted. A corresponding decryption operation at the receiver takes the ciphertext and computes the original plaintext. A good encryption scheme has the property that there are some simple rules for encryption and decryption, and that computing the plaintext from the ciphertext, or vice versa, without knowing the rules is too hard to be practical. This should be true even if you already know a lot of other plaintext and its corresponding ciphertext.

Encryption thus provides a channel with secrecy and integrity. All the parties that know the encryption rules are possible senders, and those that know the decryption rules are possible receivers. Since we want lots of secure channels, we need lots of sets of rules, one for each channel. To get them, we divide the rules into two parts, the algorithm and the key. The algorithm is fixed, and everyone knows it. The key can be expressed as a reasonably short sequence of characters, a few hundred as most. It is different for each secure channel, and is known only to the possible senders or receivers. It must be fairly easy to come up with new keys that can't be easily guessed.

There are two kinds of encryption algorithms:

> Secret key encryption, in which the same key is used to send and receive (i.e., to encrypt and decrypt). The key must be known only to the possible senders and receivers, who are the

same. The Data Encryption Standard (DES) is the most widely used form of secret key encryption [National Bureau of Standards, 1977].

Public key encryption, in which different keys are used to send and receive. The key for sending must be known only to the possible senders, and the key for receiving only to the possible receivers. For a broadcast the key for receiving may be public while the key for sending is secret; in this case there is no secrecy (since everyone knows the receiving key), but there is still integrity (since you have to know the secret sending key in order to send). The Rivest-Shamir-Adelman (RSA) algorithm is the most widely used form of public key encryption [Rivest 1978].

Known algorithms for public key encryption are slow (a few thousand bits per second at most), while it is possible to buy hardware that implements DES at 15 megabits per second, and an implementation at one gigabit per second is feasible with current technology. A practical design therefore uses a secret key scheme for handling bulk data, and uses public key encryption only for setting up secret keys and a few other special purposes. Section 3.1.5 has more on encryption.

A digital signature is a secure channel for sending a message to many receivers who may see the message long after it is sent, and are not necessarily known to the sender. Digital signatures have many important applications in making the TCB smaller. For instance, in the purchasing system described earlier, if an approved order is signed digitally it can be stored outside the TCB, and the payment component can still trust it. See section 3.1.5.3 for a more careful definition and some discussion of how to implement digital signatures.

2.3.2.2 Authenticating channels

Given a secure channel, it is still necessary to find out who is at the other end, i.e., to authenticate it. We begin with authenticating a channel from one computer system to another. The simplest way to do this is to ask for a password. Then if there is a way to match up the password with a principal, authentication is complete. The trouble with a password is that the receiver can impersonate the sender to anyone else who trusts the same password. As with secret key encryption, this means that you need a separate password to authenticate to every system that you trust differently. Furthermore, anyone who can read the channel can also impersonate the sender. If the channel is an ethernet or token ring, there may be lots of people who can read it.

Both of these problems can be overcome by challenge-response authentication schemes. These schemes make it possible to prove that you know a secret without disclosing what it is to an eavesdropper. The simplest scheme to explain is based on public key encryption. The challenger finds out the public key of the principal being authenticated, chooses a random number, and sends it to him. The principal encrypts the number with his private key and sends back the result. The challenger decrypts the result with the principal's public key; if he gets back the original number, the principal must have done the encrypting.

How does the challenger learn the principal's public key? The CCITT X.509 standard defines a framework for authenticating a secure channel to a principal with an X.500 name; this is done by authenticating the principal's public key using certificates that are digitally signed. The standard does not define how other channels to the principal can be authenticated, but technology for doing this is well understood. An X.509 authentication may involve more than one agent. For example, agent A may authenticate agent B, who in turn authenticates the principal.

Challenge-response schemes solve the problem of authenticating one computer system to another. Authenticating a user is more difficult, since users are not good at doing public-key

encryption or remembering long secrets. Traditionally, you are authenticated by what you know (a password), what you are (biometrics), or what you have (a ‘smart card’ or ‘token’). The first is the traditional method. Its drawbacks have already been explained, and are discussed in more detail in section 3.2.2.1.

Biometrics involves measuring some physical characteristic of a person: handwriting, fingerprint, retinal patterns, etc, and transmitting this to the system that is authenticating the person . The problems are forgery and compromise. It may be easy to substitute a mold of someone else’s finger, especially if the person is not being watched. And if the digital encoding of the fingerprint pattern becomes known, anyone who can bypass the physical reader and just inject the bits can impersonate the person.

A smart card or token is a way to reduce the problem of authenticating a user to the problem of authenticating a computer, by providing the user with a tiny computer he can carry around that will act as his agent to authenticate him [NIST 1988]. A smart card fits into a special reader and communicates electrically with the system; a token has a keypad and display, and the user keys in the challenge, reads the response, and types it back to the system. For more details on both, see section 3.2.2.2. A smart card or token is usually combined with a password to keep it from being easily used if it is lost or stolen; automatic teller machines require a card and a PIN for the same reason.

2.3.2.3 Perimeters

A distributed system can become very large; systems with 50,000 computers exist today, and they are growing fast. In a large system no single agent will be trusted by everyone; security must take account of this fact. To control the amount of damage that a security breach can do and to limit the scope of attacks, a large system may be divided into parts, each surrounded by a security perimeter.

Security is only as strong as its weakest link. The methods described above can in principle provide a very high level of security even in a very large system that is accessible to many malicious principals. But implementing these methods throughout the system is sure to be difficult and time-consuming. Ensuring that they are used correctly is likely to be even more difficult. The principle of “divide and conquer” suggests that it may be wiser to divide a large system into smaller parts and to restrict severely the ways in which these parts can interact with each other.

The idea is to establish a security perimeter around part of the system, and to disallow fully general communication across the perimeter. Instead, there are *gates* in the perimeter which are carefully managed and audited, and which allow only certain limited kinds of traffic (e.g., electronic mail, but not file transfers or general network datagrams). A gate may also restrict the pairs of source and destination systems that can communicate through it.

It is important to understand that a security perimeter is not foolproof. If it passes electronic mail, then users can encode arbitrary programs or data in the mail and get them across the perimeter. But this is less likely to happen by mistake, and it is more difficult to do things inside the perimeter using only electronic mail than using terminal connections or arbitrary network datagrams. Furthermore, if, for example, a mail-only perimeter is an important part of system security, users and managers will come to understand that it is dangerous and harmful to implement automated services that accept electronic mail requests.

As with any security measure, a price is paid in convenience and flexibility for a security perimeter: it's harder to do things across the perimeter. Users and managers must decide on the proper balance between security and convenience.

See section 3.2.5 for more details.

2.3.3 Methodology

An essential part of establishing trust in a computing system is ensuring that it was built using proper methods. This important subject is discussed in detail in Chapter 4 of *Computers at Risk*.

2.4 Conclusion

The technical means for achieving greater system security and trust are a function of the policies and models that people care about. Most interest and money has been spent on secrecy, so the best services and implementations support secrecy. What is currently on the market is thus only some of what is needed.

3 Some technical details

This section goes into considerably more detail on selected topics in computer security technology. The topics have been chosen either because they are well understood and fundamental, or because they are solutions to current urgent problems.

3.1 Fundamentals

3.1.1 Examples of security policies

3.1.1.1 Library example

Another "trusted system" that illustrates a number of principles is that of a library. In a very simple library where there is no librarian, anyone (a subject) can walk in and take any book (a object) desired. In this case, there is no policy being enforced and there is no mechanism for enforcing the policy. In a slightly more sophisticated case where the librarian checks who should have access to the library, but doesn't particularly care who takes out which book, the policy that is being enforced is anyone allowed in the room is allowed to access anything in the room. The policy requires only identification of the subject. In a third case, a simple extension of the previous one, no one is allowed to take more than five books out at a time. A sophisticated version of this would have the librarian check how many books you already have out before you can take more out. The policy requires a check of the person's identity and current status.

Moving toward an even more complex policy, only certain people are allowed to access certain books. The librarian performs a check by name of who is allowed to access which books. This policy frequently involves the development of long lists of names and may evolve toward, in some cases, a negative list, that is, a list of people who should not be able to have access to specific information. In large organizations users frequently have access to specific information based on the project they are working on or the sensitivity of data for which they are authorized. In each of these cases, there is an access control policy and an enforcement mechanism. The policy defines the access that an individual will have to information contained in the library. The librarian serves as the policy enforcing mechanism.

3.1.1.2 Orange Book security models

The best known and most widely used formal models of computer security functionality, e.g., Bell-LaPadula and its variants [Bell & LaPadula 1976], emphasize confidentiality (protection from unauthorized disclosure of information) as their primary security service. In particular, these models attempt to capture the 'mandatory' (what ISO 7498-2 refers to as "administratively directed, label based") aspects of security policy. This is especially important in providing protection against Trojan Horse software, a significant concern in the classified data processing community. This policy is typically enforced by operating system mechanisms at the relatively coarse granularity of processes and files. This state of affairs has resulted from a number of factors, several of which are noted below:

1) The basic security models were perceived to accurately represent DoD security concerns in which protecting classified information from disclosure, especially in the face of Trojan Horse attacks, was a primary concern. Since it was under the auspices of DoD funding that the work in formal security policy models was carried out, it is not surprising that the emphasis was on models which represented DoD concerns with regard to confidentiality.

2) The embodiment of the model in the operating system has been deemed essential in order to achieve a high level of assurance and to make available a secure platform on which untrusted (or less trusted) applications could be executed without fear of compromising overall system security. It was recognized early that the development of trusted software, i.e., software that trusted to not violate the security policy imposed on the computer system, was a very difficult and expensive task. This is especially true if the security policy calls for a high level of assurance in a potentially "hostile" environment, e.g., execution of software from untrusted sources.

The strategy evolved of developing trusted operating systems which could segregate information and processes (representing users) to allow controlled sharing of computer system resources. If trusted application software were written, it would require a trusted operating system as a platform on top of which it would execute. (If the operating system were not trusted it, or other untrusted software, could circumvent the trusted operation of the application in question.) Thus development of trusted operating systems is a natural precursor to the development of trusted applications.

At the time this strategy was developed, in the latter part of the 60's and in the 70's, computer systems were almost exclusively time-shared computers (mainframes or minis) and the resources to be shared (memory, disk storage, and processors) were expensive. With the advent of trusted operating systems, these expensive computing resources could be shared among users who would develop and execute applications without requiring trust in each application to enforce the system security policy. This has proven to be an appropriate model for systems in which the primary security concern is disclosure of information and in which the information is labelled in a fashion which reflects its sensitivity.

3) The granularity at which the security policy is enforced is, in large part, due to characteristics of typical operating system interfaces and concerns for efficient implementation of the security enforcement mechanisms. Thus, for example, since files and processes are the objects managed by most operating systems, these were the objects which were protected by the security policy embodied in the operating system. In support of Bell-LaPadula, data sensitivity labels are associated with files and authorizations for data access are associated with processes operating on behalf of users. The operating system enforces the security policy by controlling access to data based on file labels and process (user) authorizations. This type of security policy implementation is the hallmark of high assurance systems as defined by the TCSEC.

3.1.2 TCB structuring of systems

3.1.2.1 Purchasing system

A third illustration of a trusted system involves a typical business situation. In most organizations there is an individual (or group) authorized to order material, another individual (or group) authorized to receive material that has been ordered and an individual (or group) that is authorized to pay for material that has been received. To avoid theft or embezzlement, businesses establish policies that restrict 1) ordering material to the individual with proper authority; 2) receiving material to those authorized and then only for material that has been properly ordered; and 3) paying for material received to the individual who is authorized to pay, and then only for material that has been properly ordered and received.

In a nonautomated environment, controls such as review and audit procedures are established to prevent misuse (often only to limited effectiveness). The policy to be enforced can be rather simple to state. The enforcement mechanisms in a nonautomated situation may turn out to be quite complex.

In an automated environment, the material handling case might be performed in the following manner by a trusted system. The person who orders material may use her favorite untrusted word processing system to prepare an order. The person receiving material may use his favorite untrusted database management system to indicate that material has been received, and the person who writes the check may use her favorite untrusted spreadsheet to be able to generate the information for the check. How can a trusted system enforce the overall system policy and allow individuals to use untrusted tools to do their work? In this case, the trusted portion of the system (referred to as the trusted computing base) establishes separate, noncomparable project categories for the three functions. They are isolated from one another except by a specific trusted process that are described shortly. The order preparer invokes her word processor and prepares a purchase order for 500 pencils. When the document is completed, she indicates her desire to send a copy to the pencil salesman and a copy to the receiving department. All the work up to this point has been done in the ordering person’s isolated work space using her untrusted word processor without interference. At this point, the trusted computer base invokes a small trusted process that clears the user’s screen, displays the order, and requests that the user authenticate that this is indeed the order that she wishes to send. When she approves, the trusted process proceeds to print the order for the pencil salesman and labels an electronic copy so that the receiving department can now have access to it.

Up to this point, all of the information that the ordering person has employed has been labeled by the system to be accessible only to the ordering department. Once the user has indicated that the order is complete and proper, the order is made available to the receiving department and an audit record is made that the ordering person at this time and on this date authorized this order. From here on the ordering person can no longer modify the order (except by issuing another); she can read it but no longer has write access to it.

Time goes by; the pencils show up on the loading dock. The receiving person checks to see if an authorized order for the pencils exists; if so, he accepts them and indicates in his database management system that he has received the pencils. Now he prepares an authorization for the accounts payable department to pay for the pencils. All of this is done in an untrusted database management system, all of the information which the receiving department has is labeled by the trusted computing base so that only the receiving department can access the information.

Once the receipt is completed and the receiving department wishes to authorize payment, another trusted process is invoked which clears the user's screen, displays the order for payment on the screen, and requests the receiver to acknowledge that this is indeed a proper authorization for payment. Once acknowledged as correct, the trusted process copies the authorization for payment to a file labeled by the trusted computing base as accessible to the accounts payable department. Until this point the accounts payable department has known nothing about the order or the fact that it might be received shortly. Now, in the queue for the accounts payable department, a new authorization for payment appears and the check can be prepared using an untrusted spreadsheet program. Prior to the check being issued, a third trusted process will ask the accounts payable person to acknowledge that this is a correct check payment and pass the check order to the check writing mechanism.

This is a simple illustration of a case that appears in most organizations in which untrusted programs of considerable sophistication may be used in conjunction with the trusted computing base to ensure that an important business function is performed in a manner which enforces the organization's overall accounting policies. The ordering department must acknowledge the preparation of a proper order before the receiving department can receive the material. The receiving department must acknowledge the receipt of material for which it has a proper order before the accounts payable department can authorize a check.

3.1.2.2 Examples of hardware TCB components

3.1.2.2.1 VIPER

The VIPER microprocessor was designed specifically for high integrity control applications at the Royal Signals and Radar Establishment (RSRE), which is part of the UK Ministry of Defense (MoD). VIPER attempts to achieve high integrity with a simple architecture and instruction set, designed to meet the requirements of formal verification and to provide support for high integrity software.

VIPER 1 was designed as a primitive building block which could be used to construct complete systems capable of running high integrity applications. Its most important requirement is that it stop immediately if any hardware error is detected, including illegal instruction codes, numeric underflow and overflow. By stopping when an error is detected, VIPER assures that no incorrect external actions are taken following a failure. Such 'fail-stop' operation [Schlichting 1983] simplifies the design of higher-level algorithms used to maintain the reliability and integrity of the entire system.

VIPER 1 is a memory-based processor, making use of a uniform instruction set (i.e., all instructions are the same width). The processor has only three programmable 32-bit registers. The instruction set limits the amount of addressable memory to be 1 Megaword, with all access on word boundaries. There is no support for interrupts, stack processing or micro-pipelining.

The VIPER 1 architecture provides only basic program support. In fact, multiplication and division aren't supported directly by the hardware. This approach was taken primarily to simplify the design of VIPER, thereby allowing it to be verified. If more programming convenience is desired, it must be handled by a high-level compiler, assuming the resulting loss in performance is tolerable.

The VIPER 1A processor allows two chips to be used in tandem in an active/monitor relationship. That is, one of the chips can be used to monitor the operation of the other. This is achieved by comparing the memory and I/O addresses generated by both chips as they are sent

off-chip. If either chip detects a difference in this data, then both chips are stopped. In this model, a set of two chips are used to form a single fail-stop processor making use of a single memory module and I/O line.

It is generally accepted that VIPER's performance falls short of conventional processors and always will. Because it is being developed for high-integrity applications, the VIPER processor must always depend on well-established, mature implementation techniques and technologies. Many of the design decisions made for VIPER were done with static analysis in mind. Consequently, the instruction set was kept simple, without interrupt processing, to allow static analysis to be done effectively.

3.1.2.2.2 LOCK

The LOgical Coprocessing Kernel (LOCK) Project, which intends to develop a secure microcomputer prototype by 1990 that provides A1-level security for general-purpose processing. The LOCK design makes use of a hardware-based reference monitor, known as SIDEARM, that can be used to retrofit existing computers or to be included in the design of new computers as an option. The goal is to provide the highest level of security currently defined by the NCSC standards, while providing 90 percent of the performance achievable by an unmodified, insecure computer. SIDEARM is designed to achieve this goal by monitoring the memory references made by applications running on the processor to which it is attached. Assuming that SIDEARM is always working properly and can not be circumvented, it provides high assurance that applications can only access those data items for which they have been given permission. The LOCK Project centers around guaranteeing that these assumptions are valid.

The SIDEARM module is the basis of the LOCK architecture and is itself an embedded computer system, making use of its own processor, memory, communications and storage subsystems, including a laser disk for auditing. It is logically placed between the host processor and memory, examining all memory requests and responses to prevent unauthorized accesses. Since it is a separate hardware component, applications can not modify any of the security information used to control SIDEARM directly.

Access control is accomplished by assigning security labels to all subjects (i.e., applications or users) and objects (i.e., data files and programs), and enforcing a mandatory security policy independent of the host system. The SIDEARM module is also responsible for type enforcement controls, providing configurable, mandatory integrity. That is, 'types' can be assigned to data objects and used to restrict the processing allow by subjects with given security levels. Mandatory access control (MAC), discretionary access control (DAC) and type enforcement are 'additive' in that a subject must pass all three criteria before being allowed to access an object.

The LOCK project makes use of multiple TEPACHE-based TYPE-I encryption devices to safeguard SIDEARM media (e.g., hard-disk), data stored on host system media, data transmitted over the host system network interface and the unique identifiers assigned to each object. (The object identifiers are encrypted to prevent a 'covert channel' that would otherwise allow a subject to determine how many objects were generated by another subject.) As such, LOCK combines aspects of both COMSEC (Communications Security) and COMPUSEC (Computer Security) in an inter-dependent manner. The security provided by both approaches are critical to LOCK's proper operation.

The LOCK architecture requires a few trusted software components, including a SIDEARM device driver and extensions to the operating system kernel. The operating system extensions handle machine-dependent support, such as printer and terminal security labeling, and

application specific security policies, such as that required by a database management system. These functions implemented outside the TCB provide the flexibility needed to allow SIDEARM to support a wide range of applications, without becoming too large or becoming architecture-dependent.

One of LOCK's advantages is that a major portion of the operating system, outside of the kernel extensions and the SIDEARM device driver, can be considered 'hostile'. That is, even if the operating system is corrupted, LOCK will not allow an unauthorized application to access data objects. However, parts of the operating system must still be modified or removed to make use of the functionality provided by SIDEARM. The LOCK project intends to port UNIX System V directly to the LOCK architecture and to certify the entire system at the A1-level.

3.1.3 Application Layer Security

Although the operating systems which have resulted from the Bell- LaPadula paradigm have been used to provide the sort of process and file segregation noted above, the development of trusted applications using these operating systems has often proven to be quite difficult and/or very limiting. This has been the case even for applications in which the emphasis is primarily one of confidentiality, as noted in examples below.

In the course of developing a secure messaging system based on DoD requirements, the Naval Research Laboratory (NRL) has found the operating system support provided by TCSEC-evaluated system to be lacking. They have spent considerable effort over more than 5 years developing a security policy model and, more recently, an operating system interface tailored to this application. Electronic messaging is a widespread application, not just in the military environment.

Trusted (disclosure-secure) database management systems represent another major application area. This is viewed as a sufficiently important and specialized application as to warrant the development of evaluation criteria of its own, i.e., the Trusted Database Interpretation of the TCSEC. These criteria have been under development for several years and represent an attempt to apply the TCSEC policy and evaluation model to database systems. Trusted database systems developed directly on top of trusted operating systems (as defined by the TCSEC criteria) have exhibited some significant limitations, e.g., in terms of functionality and/or performance [Denning 1988]. Development of full functionality trusted database management systems would seem to require a significant effort in additional to the use of a TCSEC-secure operating system base.

These examples suggest that, even when the primary security policy is one of confidentiality, trusted operating systems do not necessarily provide a base which makes development of trusted applications straightforward. Even though a trusted operating system can be seen as providing a necessary foundation for a secure application, it may not provide a sufficient foundation.

Over time it has become increasingly apparent that many military and commercial security applications call for a richer set of security policies, e.g., various flavors of integrity and more complex authorization facilities [Clark and Wilson1987], not just confidentiality. This poses a number of problems for developers of applications which would be deemed "trusted" relative to these policies:

1) These other forms of trustedness are generally viewed as being harder to achieve. Security from unauthorized disclosure of information is simple compared to some of the other security requirements which users envision. Specification and verification of general program

“correctness” properties are much more complex than statements dealing only with data flow properties of programs.

2) There are no generally accepted security policy models for the wider range of security services alluded to above and thus criteria against which such systems can be evaluated.

3) Existing trusted operating systems generally do not provide facilities tailored to support these security policies, thus requiring an application developer to provide these services directly.

These problems are closely related. For example, trusted operating systems may not provide a good foundation for development of applications with more sophisticated security service requirements in in part because of the orientation of the TCSEC. Inclusion of software in an operating system to support an application developer who wishes to provide these additional services would delay the evaluation of the operating system. The delay would arise because of the greater complexity that accrues from the inclusion of the additional trusted software and because there are no policy models on which to base the evaluation of this added trust functionality. Thus the TCSEC evaluation process discourages inclusion of other security facilities in trusted operating systems as these facilities do not yield higher ratings and may hinder the evaluation relative to existing policy models. If the criteria for trusted operating systems were expanded to include additional security facilities in support of application security policies, vendors might be encouraged to include such facilities in their products.

Also note that, in many instances, one might expect that the level of assurance provided by security facilities embodied in applications will tend to be lower than that embodied in the operating system. The assumption is that developers of applications will, in general, not be able to devote as much time and energy to security concerns as do the developers of secure operating systems. (Database systems constitute an exception to this general observation as they are sufficiently general and provide a foundation for so many other applications as to warrant high assurance development procedures and their own evaluation process and criteria.) This is not so much a reflection on the security engineering capabilities of application developers versus operating system developers, but rather an observation about the relative scope of specific applications versus specific operating systems. It seems appropriate to devote more effort to producing a secure operating system for a very large user community than to devote a comparable level of effort to security facilities associated with a specific application for a smaller community.

In discussing security policies for applications it is also important to observe that in many contexts the application software must, in some sense, be viewed as “trusted” because it is the correct operation of the application which constitutes trusted operation from the user’s perspective. For example, trusted operation of an accounts payable system may require creating matching entries in two ledgers (debits and credits) and authorization by two independent individuals before a check is issued. These requirements are not easily mapped into the conventional security facilities, e.g., access controls, provided by TCSEC-evaluated operating systems.

This is not to say that the process and file-level segregation security facilities provided by current secure operating systems are irrelevant in the quest for application security. Although these facilities have often proven to be less than ideal building blocks for secure applications, they do provide useful, high assurance firewalls among groups of users, different classes of applications, etc. If nothing else, basic operating system security facilities are essential to ensure the inviolability of security facilities that might be embedded within applications or extensions to the operating system. Thus the process and data isolation capabilities found in current trusted

systems should be augmented with, not replaced by, security facilities designed to meet the needs of developers of trusted applications.

One Solution Approach

In order to better support the development and operation of trusted applications, vendors probably need to provide a set of security facilities than goes beyond those currently offered by most trusted operating systems. Development of trusted applications has probably been hindered by the coarseness and limited functionality of the protection facilities offered by current trusted operating systems. None of the trusted operating system products now on the Evaluated Products List (EPL) appear to incorporate security mechanisms sufficient to support the development of trusted applications as described above. This is not a criticism of these systems, nor of their developers. It is probably reflects the difficulty of engineering operating systems with such added functionality, and the penalty imposed by the current evaluation criteria for systems which might attempt to incorporate such facilities in their products, as discussed earlier.

To counter the latter problem the R&D and vendor communities must be encouraged to develop operating systems which embody explicit security facilities in support of applications. Such facilities might take the form of a set of security primitives which are deemed generally useful in the construction of secure applications, if such a set of application security primitives can be identified. It also may be appropriate to provide facilities to support the construction of what sometimes have been called ‘protected subsystems’, so that application developers can build systems which are protected from user software and from other applications, without subverting the basic security services offered by the operating system.

The LOCK system [Boebert 1985], a trusted operating system program funded by the NCSC, incorporates an elaborate set of security facilities which may be an appropriate model for the sort of operating system constructs that are required for secure application development. This system, which is designed as an adjunct to various operating system/hardware bases, includes facilities to support protected subsystems. These facilities are designed to enable applications be to constructed in a layered fashion (from a security standpoint). Application-specific security software could be protected from other applications and from user-developed software and would not be able to compromise the basic security guarantees offered by the operating system (e.g., process and data segregation according).

It is not yet known if the constructs provided by LOCK will yield application-specific security software which is “manageable”, e.g., which can be readily understood by application developers, nor is the performance impact fo these facilities well understood. LOCK is designed as a “beyond A1” system and thus includes features to support very high assurance for all of its security functionality, e.g., encryption of storage media. A trusted operating system which provided the security functionality of LOCK without some of these assurance features might be adequate for many of trusted application contexts. Finally, LOCK should not be viewed as the only approach to this problem, but rather as one technical approach which is the product of considerable research and development activity.

3.1.4 Formal validation

Working from the bottom up, we have a lot of confidence in quantum electrodynamics; though we certainly can’t give any proof that electrons behave according to that specification, there is a lot of experimental support, and many people have studied the theory over more than fifty years. We are pretty sure about the device physics of transistors and the behavior of signals on wires, especially when conservatively designed, even though in practice we can’t calculate these these

things from their quantum basis. It is one more step to assemble transistors and wires and abstract their behavior as digital zeros and ones and AND and OR gates. This step is validated by simulations of the electrical behavior of the circuits, based on our models of transistors and wires. None of these steps is completely formal; all rely on inspection by many people over a long period of time.

From this point there are three steps which can be formalized completely:

logic design goes from digital signals and gates to a computer with memory and instructions;

an assembler goes from a computer interpreting bit patterns as instructions to an assembly language description of a program;

a compiler goes from the assembly language program to a higher level language.

Each of these steps has been formalized and mechanically checked at least once, and putting all three together within the same formal system is within reach. We must also ask how the mechanical checker is validated; this depends on making it very simple and publishing it for widespread inspection. The system that checks a formal proof can be much simpler than the theorem-proving system that generates one; the latter need not be trusted.

It's important to realize that no one has built a system using a computer, assembler and compiler that have all been formally validated, though this should be possible in a few years. Furthermore, such a system will be significantly slower and more expensive than one built in the usual way, because the simple components that can be validated don't have room for the clever tricks that make their competitors fast. Normally these parts of the TCB are trusted based on fairly casual inspection of their implementation and a belief that their implementors are trustworthy. It has been demonstrated that casual inspection is not enough to find fatal weaknesses [Thompson 1984].

3.1.5 Cryptography

3.1.5.1 Fundamental Concepts of Encryption

Cryptography and cryptanalysis have existed for at least two thousand years, perhaps beginning with a substitution algorithm used by Julius Caesar [Tanebaum 1981]. Using his method, every letter in the original message, known as the plaintext, is replaced by the letter which occurs three places later in the alphabet. That is, A is replaced by D, B is replaced by E, and so on. For example, the plaintext "VENI VIDI VICI" would yield "YHQL YLGL YLFL". The resulting message, known as the ciphertext, is then couriered to the awaiting centurion, who 'decrypts' it by replacing each letter with the letter which occurs three places'before it in the alphabet. The encryption and decryption algorithms are essentially controlled by the number three, which is known as the encryption and decryption 'key'. If Caesar suspected that an unauthorized person had discovered how to decrypt the ciphertext, he could simply change the key value to another number and inform the field generals of that new value using some other communication method.

Although Caesar's cipher is a relatively simple example of cryptography, it clearly depends on a number of essential components: the encryption and decryption algorithms, a key which is known by all authorized parties, and the ability to change the key. Figure X shows the encryption process and how the various components interact.

Encryption Decryption Key Key | | v v Plaintext -> Encryption -> Ciphertext -> Decryption -> Plaintext (Message) Algorithm Algorithm (Message)

Figure X. The Encryption Process

If any of these components are compromised, the security of the information being protected decreases. If a weak encryption algorithm is chosen, an opponent might be able to guess the plaintext once a copy of the ciphertext is obtained. In many cases, the cryptanalyst need only know the type of encryption algorithm being used in order to break it. For example, knowing that Caesar used only a cyclic substitution of the alphabet, we could simply try every key value from 1 to 25, looking for the value which resulted in a message containing Latin words. Similarly, there are many encryption algorithms which appear to be very complicated, but are rendered ineffective by an improper choice of a key value. In a more practical sense, if the receiver forgets the key value or uses the wrong one then the resulting message will probably be unintelligible, requiring additional effort to re-transmit the message and/or the key. Finally, it is possible that the enemy will break the code even if the strongest possible combination of algorithms and key values is used. Therefore, keys and possibly even the algorithms need to be changed over a period of time to limit the loss of security when the enemy has broken the current system. The process of changing keys and distributing them to all parties concerned is known as ‘key management’ and is the most difficult aspect of security management after an encryption method has been chosen.

In theory, any logical function can be used as an encryption algorithm. The function may act on single bits of information, single letters in some alphabet, single words in some language or groups of words. The Caesar cipher is an example of an encryption algorithm which operates on single letters within a message. A number of ‘codes’ have been used throughout history in which a two column list of words are used to define the encryption and decryption algorithms. In this case, plaintext words are located in one of the columns and replaced by the corresponding word from the other column to yield the ciphertext. The reverse process is performed to regenerate the plaintext from the ciphertext. If more than two columns are distributed, a key can be used to designate both the plaintext and ciphertext columns to be used. For example, given 10 columns, the key (3,7) might designate that the third column represents plaintext words and the seventh column represents ciphertext words. Although code books (e.g., multi-column word lists) are convenient for manual enciphering and deciphering, their very existence can lead to compromise. That is, once a code book falls into enemy hands, ciphertext is relatively simple to decipher. Furthermore, code books are difficult to produce and to distribute, requiring accurate accounts of who has which books and which parties can communicate using those books. Consequently, mechanical and electronic devices have been developed to automate the encryption and decryption process, using primarily mathematical functions on single bits of information or single letters in a given alphabet.

3.1.5.2 Private vs. Public Crypto-Systems

The security of a given crypto-system depends on the amount of information known by the cryptanalyst about the algorithms and keys in use. In theory, if the encryption algorithm and keys are independent of the decryption algorithm and keys, then full knowledge of the encryption algorithm and key wouldn’t help the cryptanalyst break the code. However, in many practical crypto-systems, the same algorithm and key are used for both encryption and decryption. The security of these ‘symmetric cipher’ systems depends on keeping at least the key secret from others, making them known as ‘private key crypto-systems’.

An example of a symmetric, private-key crypto-system is the Data Encryption Standard (DES) [NBS 1978]. In this case, the encryption/decryption algorithm is widely known and has been widely studied, relying on the privacy of the encryption/decryption key for its security. Other private-key systems have been implemented and deployed by the NSA for the protection of classified government information. In contrast to the DES, the encryption/decryption algorithms within those crypto-systems have been kept private, to the extent that the computer chips on which they are implemented are coated in such a way as to prevent them from being examined.

Users are often intolerant of private encryption and decryption algorithms because they don't know how the algorithms work or if a 'trap-door' exists which would allow the algorithm designer to read the user's secret information. In an attempt to eliminate this lack of trust a number of crypto-systems have been developed around encryption and decryption algorithms based on fundamentally-difficult problems, or 'one-way functions', which have been studied extensively by the research community. In this way, users can be confident that no trap-door exists that would render their methods insecure.

For practical reasons, it is desirable to use different encryption and decryption keys in the crypto-system. Such 'asymmetric' systems allow the encryption key to be made available to anyone, while remaining confident that only people who hold the decryption key can decipher the information. These systems, which depend solely on the privacy of the decryption key, are known as 'public-key cryptosystems'. An example of an asymmetric, public-key cipher is the patented RSA system.

The primary benefit of a public-key system is that anyone can send a secure message to another person simply by knowing the encryption algorithm and key used by the recipient. Once the message is enciphered, the sender can be confident that only the recipient can decipher it. This mode of operation is facilitated by the use of a directory or depository of public keys which is available to the general public. Much like the telephone book, the public encryption key for each person could be listed, along with any specific information about the algorithm being used.

3.1.5.3 Digital Signatures

Society accepts handwritten signatures as legal proof that a person has agreed to the terms of a contract as stated on a sheet of paper, or that a person has authorized a transfer of funds as indicated on a check. But the use of written signatures involves the physical transmission of a paper document; this is not practical if electronic communication is to become more widely used in business. Rather, a 'digital signature' is needed to allow the recipient of a message or document to irrefutably verify the originator of that message or document.

A written signature can be produced by one person (although forgeries certainly occur), but it can be recognized by many people as belonging uniquely to its author. A digital signature, then, to be accepted as a replacement for a written signature would have to be easily authenticated by anyone, but producible only by its author.

3.1.5.3.1 Detailed Description

A digital signature system consists of three procedures

- the generator, which produces two numbers called the mark and the secret;
- the signer, which accepts a secret and an arbitrary sequence of bytes called the input, and produces a number called the signature;

- the checker, which accepts a mark, an input, and a signature and says whether the signature matches the input for that mark or not.

The procedures have the following properties

(1) If the generator produces a mark and a secret, and the signer when given the secret and an input produces a signature, then the checker will say that the signature matches the input for that mark.

(2) Suppose you have a mark produced by the generator, but don't have the secret. Then even if you have a large number of inputs and matching signatures for that mark, you still cannot produce one more input and matching signature for that mark. In particular, even if the signature matches one of your inputs, you can't produce another input that it matches. A digital signature system is useful because if you have a mark produced by the generator, and you have an input and matching signature, then you can be sure that the signature was computed by a system that knew the corresponding secret. The reason is that, because of property (2), a system that didn't know the secret couldn't have computed the signature.

For instance, you can trust a mark to certify an uninfected program if

- you believe that it came from the generator, and
- you also believe that any system that knows the corresponding secret is one that can be trusted not to sign a program image if it is infected.

Known methods for digital signatures are based on computing a secure check sum of the input to be signed, and then encrypting the check sum with the secret. If the encryption uses public key encryption, the mark is the public key that matches the secret, and the checker simply decrypts the signature.

For more details, see Davies and Price, Security for Computer Networks: An Introduction to Data Security in Teleprocessing and Electronic Funds Transfers (New York, J. Wiley, 1084), ch 9.

3.1.5.3.2 Secure Check Sums

A secure check sum or one-way hash function accepts any amount of input data (in this case a file containing a program) and computes a small result (typically 8 or 16 bytes) called the check sum. Its important property is that it takes a lot of work to find a different input with the same check sum. Here "a lot of work" means "more computing than an adversary can afford".

A secure check sum is useful because it identifies the input: any change to the input, even a very clever one made by a malicious person, is sure to change the check sum. Suppose someone you trust tells you that the program with check sum 7899345668823051 does not have a virus (perhaps he does this by signing the check sum with a digital signature). If you compute the check sum of file WORDPROC.EXE and get 7899345668823051, you should believe that you can run WORDPROC.EXE without worrying about a virus.

For more details, see Davies and Price, ch 9.

3.1.5.4 Public-Key Cryptosystems and Digital Signatures

Public key cryptosystems offer a means of implementing digital signatures. In a public key system the sender enciphers a message using the receiver's public key creating ciphertext1. To sign the message he enciphers ciphertext1 with his private key creating ciphertext2. Ciphertext2

is then sent to the receiver. The receiver applies the sender's public key to decrypt ciphertext2, yielding ciphertext1. Finally, the receiver applies his private key to convert ciphertext1 to plaintext. The authentication of the sender is evidenced by the fact that the receiver successfully applied the sender's public key and was able create plaintext. Since encryption and decryption are opposites, using the sender's public key to decipher the sender's private key proves that only the sender could have sent it.

To resolve disputes concerning the authenticity of a document, the receiver can save the ciphertext, the public key, and the plaintext as proof of the sender's signature. If the sender later denies that the message was sent, the receiver can present the signed message to a court of law where the judge then uses the sender's public key to check that the ciphertext corresponds to a meaningful plaintext message with the sender's name, the proper time sent, etc. Only the sender could have generated the message, and therefore the receiver's claim would be held up in court.

3.1.5.5 Key Management

In order to use a digital signature to certify a program (or anything else, such as an electronic message), it is necessary to know the mark that should be trusted. Key management is the process of reliably distributing the mark to everyone who needs to know it. When only one mark needs to be trusted, this is quite simple: someone you trust tells you what the mark is. He can't do this using the computer system, because you would have no way of knowing that the information actually came from him. Some other communication channel is needed: a face-to-face meeting, a telephone conversation, a letter written on official stationery, or anything else that gives adequate assurance. When several agents are certifying programs, each using its own mark, things are more complex. The solution is for one agent that you trust to certify the marks of the other agents, using the same digital signature scheme used to certify anything else. CCITT standard X.509 describes procedures and data formats for accomplishing this multi-level certification.

3.1.5.6 Algorithms

3.1.5.6.1 One-Time Pads

There is a collection of relatively simple encryption algorithms, known as one-time pad algorithms, whose security is mathematically provable. Such algorithms combine a single plaintext value (e.g., bit, letter or word) with a random key value to generate a single ciphertext value. The strength of one-time pad algorithms lies in the fact that separate random key values are used for each of the plaintext values being enciphered and the stream of key values used for one message is never used for another, as the name implies. Assuming there is no relationship between the stream of key values used during the process, the cryptanalyst has to try every possible key value for every ciphertext value, which can be made very difficult simply by using different representations for the plaintext and key values.

The primary disadvantage of one-time pad systems is that it requires an amount of key information equal to the size of the plaintext being enciphered. Since the key information must be known by both parties and is never re-used, the amount of information exchanged between parties is twice that contained in the message itself. Furthermore, the key information must be transmitted using mechanisms different from those for the message, thereby doubling the resources required. Finally, in practice, it is relatively difficult to generate large streams of "random" values efficiently. Any non-random patterns which appear in the key stream provides the cryptanalyst with valuable information which can be used to break the system.

One-time pads can be implemented efficiently on computers using any of the primitive logical functions supported by the processor. For example, the Exclusive-Or (XOR) operator is a convenient encryption/decryption function. When 2 bits are combined using the XOR operator, the result is 1 if one and only one of the input bits is 1, otherwise the result is 0, as defined by the table in Figure X.

1 XOR 0 = 1 0 XOR 1 = 1 0 XOR 0 = 0 1 XOR 1 = 0

Figure X. The XOR Function

The XOR function is convenient because it is fast and you can decrypt the encrypted information simply by XORing the ciphertext with the same data (key) that you used to encrypt the plaintext, as shown in Figure Y.

ENCRYPTION Plaintext 0101 0100 0100 0101 Key 0100 0001 0100 0001 ---------- Ciphertext 0001 0101 0000 0100

DECRYPTION Ciphertext 0001 0101 0000 0100 Key 0100 0001 0100 0001 ---------- Plaintext 0101 0100 0100 0101

Figure Y. Encryption/Decryption using XOR Function

3.1.5.6.2 Data Encryption Standard (DES)

In 1972, the NBS identified a need for a standard cryptosystem for unclassified applications and issued a call for proposals. Although it was poorly received at first, IBM proposed, in 1975, a private-key cryptosystem which operated on 64-bit blocks of information and used a single 128-bit key for both encryption and decryption. After accepting the initial proposal, NBS sought both industry and NSA evaluations. Industry evaluation was desired because NBS wanted to provide them with a secure encryption that they would want to use and NSA's advice was requested because of their historically strong background in cryptography and cryptanalysis. NSA responded with a generally favorable evaluation, but recommended that the key length be changed from 128-bits to 56 bits and that some of its fundamental components, known as S-boxes, be re-designed. Based primarily on that recommendation, the Data Encryption Standard (DES) became a federal information processing standard in 1977 and an ANSI standard in 1980, using a 56-bit key.

The DES represents that first time that the U.S. government has developed a cryptographic algorithm in public. Historically, such algorithms have been developed by the NSA as highly classified projects. However, despite the openness of its design, many researchers believed that NSA's influence on the S-box design and the length of the key introduced a trap-door which allowed the NSA to read any message encrypted using the DES. In fact, one researcher described the design of a special-purpose parallel processing computer that was capable of breaking a DES system using 56-bit keys and that, according to the researcher, could be built by the NSA using conventional technology. Nonetheless, in over ten years of academic and industrial scrutiny, no flaw in the DES has been made public. Unfortunately, as with all cryptosystems, there is no way of knowing if the NSA or any other organization has succeeded in breaking the DES.

The controversy surrounding the DES was reborn when the NSA announced that it would not recertify the algorithm for use in unclassified government applications after 1987. (Note, DES has never been used to protect classified, government information, which is protected using methods controlled by the NSA.) An exception to this ruling was made for electronic funds transfer applications, most notably FedWire, which had invested substantially in the use of DES.

NSA cited the widespread use of the DES as a disadvantage, stating that if it were used too much it would become the prime target of criminals and foreign adversaries. In its place, NSA has offered a range of private-key algorithms based on classified algorithms that make use of keys which are generated and managed by NSA.

The DES algorithm has four approved modes of operation, the electronic cookbook, cipher block chaining, cipher feedback and output feedback modes. Each of these modes has certain characteristics that make them more appropriate for specific purposes. For example, the cipher block chaining and cipher feedback modes are intended for message authentication purposes, while the electronic cookbook mode is used primarily for encryption and decryption of bulk data.

3.1.5.6.3 RSA

RSA is a public key crypto-system, invented and patented by Ronald Rivest, Adi Shamir and Leonard Adelman, which is based on large prime numbers. In their method, the decryption key is generated by selecting a pair of prime numbers, P and Q, (i.e., numbers which are not divisible by any other) and another number, E, which must pass a special mathematical test based on the values of the pair of primes. The encryption key consists of the product of P and Q, which we call N, and the number E, which can be made publicly available. The decryption key consists of N and another number, called D, which results from a mathematical calculation using N and E. The decryption key must be kept secret.

A given message is encrypted by converting the text to numbers (using conventional conversion mechanisms) and replacing each number with a number computed using N and E. Specifically, each number is multiplied by itself E times, with the result being divided by N, yielding a quotient, which is discarded, and a remainder. The remainder is used to replace the original number as part of the ciphertext. The decryption process is similar, multiplying the ciphertext number by itself D times (vice E times) and dividing it by N, with the remainder representing the desired plaintext number (which is converted back to a letter).

RSA's security depends on the fact that, while finding large prime numbers is computationally easy, factoring large integers into their component primes is not. That is, in order to break the RSA algorithm, a cryptanalyst must be able to factor large numbers into their prime components and this is computationally intensive. However, in recent years, parallel processing techniques have significantly increased the size of numbers (measured as the number of decimal digits in its representation) that can be factored in a relatively short period of time (i.e., less than 24 hours). Seventy digit numbers are well within reach of modern computers and processing techniques, with eighty digit numbers on the horizon. Consequently, most of commercial RSA systems use 512-bit keys (i.e., 154-digits) which should be out of the reach of conventional computers and algorithms for quite some time.

3.2 Topics of current interest

3.2.1 Commercial integrity models

3.2.1.1 Proprietary software/database protection

The problem of the protection of proprietary software or of proprietary databases is an old and difficult one. The blatant copying of a large commercial program such as a payroll program, and its systematic use within the pirating organization is often detectable and will then lead to legal action. Similar considerations apply to large databases and for these the pirating organization has

the additional difficulty of obtaining the vendor-supplied periodic updates, without which the pirated database will become useless.

The problem of software piracy is further exacerbated in the context of personal computing. Vendors supply programs for word processing, spread sheets, game playing programs, compilers etc., and these are systematically copied by pirate vendors and by private users. While large scale pirate vendors may be eventually detected and stopped, there is no hope of hindering through detection and legal action, the mass of individual users form copying from each other.

Various technical solutions were proposed for the problem of software piracy in the personal computing world. Some involve a machine customized layout of the data on the disk. Others involve the use of volatile transcription of certain parts of the program text (A. Shamir). Cryptography employing machine or program instance customized keys is suggested, in conjunction with co-processors which are physically impenetrable so that cryptographic keys and crucial decrypted program text cannot be captured. Some of these approaches, especially those employing special hardware, and hence requiring cooperation between hardware and software manufacturers, did not penetrate the marketplace. The safeguards deployed by software vendors are usually incomplete and after a while succumb to attacks by talented amateur hackers who produce copyable versions of the protected disks. There even exist programs (Nibble) to help a user overcome the protections of many available proprietary programs. (These thieving programs are then presumably themselves copied through use of their own devices!)

It should be pointed out that there is even a debate as to whether the prevalent theft of proprietary personal computing software by individuals is sufficiently harmful to warrant the cost of development and of deployment of really effective countermeasures.

It is our position that the problem of copying proprietary software and databases discussed above, while important, lie outside the purview of system security. Software piracy is an issue between the rightful owner and the thief and its resolution depends on tools and methods, and represents a goal, which are disjoint from system security.

There is, however, an important aspect of protection of proprietary software and/or databases which lies directly within the domain of system security as we understand it. It involves the unauthorized use of proprietary software/databases by parties other than the organization licensed to use that software/database, and that within the organization’s system where the proprietary software is legitimately installed. Consider, for example, a large database with the associated complex-query software which is licensed by a vendor to an organization. This may be done with the contractual obligation that the licensee obtains the database for his own use and not for making query services available to outsiders. Two modes of transgression against the proprietary rights of the vendor are possible. The organization itself may breach its obligation not to provide the query services to others, or some employee who himself may have legitimate access to the database may provide or even sell query services to outsiders. In the latter case the licensee organization may be held responsible, under certain circumstances, for not having properly guarded the proprietary rights of the vendor. Thus, there is a security issue associated with the prevention of unauthorized use of proprietary software/database which is legitimately installed in a computing system. In our classification of security services it comes under the heading of resource (usage) control. Namely, the proprietary software is a resource and we wish to protect against its unauthorized use (say for sale of services to outsiders) by a user who is otherwise authorized to access that software.

The security service of resource control has attracted very little, if any, research and implementation efforts. It poses some difficult technical as well as possible privacy problems.

The obvious approach is to audit, on a selective and possibly random basis, accesses to the proprietary resource in question. This audit trail can then be evaluated by human scrutiny, or automatically, for indications of unauthorized use as defined in the present context. It may well be that effective resource control will require recording, at least on a spot check basis, of aspects of the content of the user's interaction with the software/database. For obvious reasons, this may encounter resistance.

Another security service that may come into play in this context of resource control is non-repudiation. The legal aspects of the protection of proprietary rights may require that certain actions taken by the user in connection with the proprietary rights may require that certain actions taken by the user in connection with the proprietary resource be such that once recorded, the user is barred from later on repudiating his connection to these actions.

It is clear that such measures for resource control if properly implemented and installed will serve to deter unauthorized use by individual users of proprietary resources. But what about the organization controlling the trusted system in which the proprietary resource is imbedded. The organization may well have the ability to dismantle the very mechanisms designed to control the use of proprietary resources, thereby evading effective scrutiny by the vendor or his representations. But the design and nature of security mechanisms is such that they are difficult to change selectively, and especially in a manner ensuring that their subsequent behavior will emulate the untempered mode thus, making the change undetectable. Thus the expert effort and people involved in effecting such changes will open the organization to danger of exposure.

At the present time there is no documented major concern about the unauthorized use, in the sense of the present discussion, of proprietary programs/databases. It may well be that in the future, when the sale of proprietary databases will assume economic significance, the possibility of abuse of proprietary rights by licenced organizations and authorized will be an important issue. At that point an appropriate technology for resource control will be essential.

3.2.2 Authentication: secure channels to users

3.2.2.1 Passwords

Passwords have been used throughout military history as a mechanism to distinguish friends and foes. When sentries were posted they were told the daily password which would be given by any friendly soldier that attempted to enter the camp. Passwords represent a shared secret that allow strangers to recognize each other and have a number of advantageous properties. They can be chosen to be easily remembered (e.g., "Betty Boop") without begin easily guessed by the enemy (e.g., "Mickey Mouse"). Furthermore, passwords allow any number of people to use the same authentication method and can be changed frequently (as opposed to physical keys which must be duplicated). The extensive use of passwords for user authentication in human-to-human interactions has led to their extensive use in human-to-computer interactions.

"A password is a character string used to authenticate an identity. Knowledge of the password that is associated with a user ID is considered proof of authorization to use the capabilities associated with that user ID." (NCSC - Password Management Guideline)

Passwords can be issued to users automatically by a random generation routine, providing excellent protection against commonly-used passwords. However, if the random password generator is not good, breaking one may be equivalent to breaking all. At one installation, a person reconstructed the entire master list of passwords by guessing the mapping from random numbers to alphabetic passwords and inferring the random number generator. For that reason, the

random generator must base its seed from a non-deterministic source such as the system clock. Often the user will not find a randomly selected password acceptable because it is too difficult to memorize. This can significantly decrease the advantage of random passwords because the user may write the password down somewhere in an effort to remember it. This may cause infinite exposure of the password thwarting all attempts of security. For this reason it can be helpful to give the user the option to accept or reject, or choose from a list. This may increase the probability that the user will find an acceptable password.

User defined passwords can be a positive method for assigning passwords if the users are aware of the classic weaknesses. If the password is too short, say 4 digits, a potential intruder can exhaust all possible password combinations and gain access quickly. That is why every system must limit the number of tries any user can make towards entering his password successfully. If the user picks very simple passwords, potential intruders can break the system by using a list of common names or using a dictionary. A dictionary of 100,000 words has been shown to raise the intruder's chance of success by 50 percent. Specific guidelines of how to pick passwords is important if the users are allowed to pick their own passwords. Voluntary password systems should guide the user to never reveal his password to another user and to change the password on a regular basis, which can be enforced by the system. (The NCSC - Guide to Password Management represents such a guideline.)

There must be a form of access control provided to prevent unauthorized persons from gaining access to the password list and reading or modifying the list. One way to protect passwords in internal storage is by encryption. The passwords of each user are stored as ciphertext produced by an approved cryptographic algorithm. When a user signs on and enters his password, the password is processed by the algorithm to produce the corresponding ciphertext. The plaintext password is immediately deleted, and the ciphertext version of the password is compared with the one stored in memory. The advantage of this technique is that passwords cannot be stolen from the computer. However, a person obtaining unauthorized access could delete or change the ciphertext passwords and effectively deny service.

The longer a password is used, the more opportunities exist for exposing it. The probability of compromise of a password increases during its lifetime. This probability is considered acceptably low for an initial time period, after a longer time period it becomes unacceptably high. There should be a maximum lifetime for all passwords. It is recommended that the maximum lifetime of a password be no greater than 1 year. (NCSC Password Guideline Management)

3.2.2.2 Tokens

- physical device assigned to a user - usually used in conjunction with password or PIN - magnetic strip cards - inexpensive - may be forged - smart cards - contains a micro-processor, memory and interface - stores user profile data - usually encrypted

3.2.2.3 Biometric Techniques

Voice - fingerprints - retinal scan - signature dynamics.

Can be combined with smart cards easily

Problems: forgery, compromise

3.2.3 Networks and distributed systems

3.2.4 Viruses

A computer virus is a program which

- is hidden in another program (called its host) so that it runs whenever the host program runs, and
- can make a copy of itself.

When the virus runs, it can do a lot of damage. In fact, it can do anything that its host can do: delete files, corrupt data, send a message with the user's secrets to another machine, disrupt the operation of the host, waste machine resources, etc. There are many places to hide a virus: the operating system, an executable program, a shell command file, a macro in a spreadsheet or word processing program are only a few of the possibilities. In this respect a virus is just like a trojan horse. And like a trojan horse, a virus can attack any kind of computer system, from a personal computer to a mainframe.

A virus can also make a copy of itself, into another program or even another machine which can be reached from the current host over a network, or by the transfer of a floppy disk or other removable medium. Like a living creature, a virus can spread quickly. If it copies itself just once a day, then after a week there will be more than 50 copies, and after a month about a billion. If it reproduces once a minute (still slow for a computer), it takes only half an hour to make a billion copies. Their ability to spread quickly makes viruses especially dangerous.

There are only two reliable methods for keeping a virus from doing harm:

- Make sure that every program is uninfected before it runs.
- Prevent an infected program from doing damage.

3.2.4.1 Keeping It Out

Since a virus can potentially infect any program, the only sure way to keep it from running on a system is to ensure that every program you run comes from a reliable source. In principle this can be done by administrative and physical means, ensuring that every program arrives on a disk in an unbroken wrapper from a trusted supplier. In practice it is very difficult to enforce such procedures, because they rule out any kind of informal copying of software, including shareware, public domain programs, and spreadsheets written by a colleague. A more practical method uses digital signatures.

Informally, a digital signature system is a procedure you can run on your computer that you should believe when it says "This input data came from this source" (a more precise definition is given below). Suppose you have a source that you believe when it says that a program image is uninfected. Then you can make sure that every program is uninfected before it runs by refusing to run it unless

- you have a certificate that says "The following program is uninfected:" followed by the text of the program, and
- the digital signature system says that the certificate came from the source you believe.

Each place where this protection is applied adds to security. To make the protection complete, it should be applied by any agent that can run a program. The program image loader is not the only

such agent; others are the shell, a spreadsheet program loading a spreadsheet with macros, a word processing program loading a macro, and so on, since shell scripts, macros etc. are all programs that can host viruses. Even the program that boots the machine should apply this protection when it loads the operating system.

3.2.4.2 Preventing Damage

Because there are so many kinds of programs, it may be hard to live with the restriction that every program must be certified uninfected. This means, for example, that a spreadsheet can't be freely copied into a system if it contains macros. If you want to run an uncertified program, because it might be infected you need to prevent it from doing damage--leaking secrets, changing data, or consuming excessive resources.

Access control can do this if the usual mechanisms are extended to specify programs, or set of programs, as well as users. For example, the form of an access control rule could be "user A running program B can read" or "set of users C running set of programs D can read and write". Then we can define a set of uninfected programs, namely the ones which are certified as uninfected, and make the default access control rule be "user running uninfected" instead of "user running anything". This ensures that by default an uncertified program will not be able to read or write anything. A user can then relax this protection selectively if necessary, to allow the program access to certain files or directories.

3.2.4.3 Vaccines

It is well understood how to implement the complete protection against viruses just described, but it requires changes in many places: operating systems, command shells, spreadsheet programs, programmable editors, and any other kinds of programs, as well as procedures for distributing software. These changes ought to be implemented. In the meantime, however, there are various stopgap measures that can help somewhat. They are generally known as vaccines, and are widely available for personal computers.

The idea of a vaccine is to look for traces of viruses in programs, usually by searching the program images for recognizable strings. The strings may be either parts of known viruses that have infected other systems, or sequences of instructions or operating system calls that are considered suspicious. This idea is easy to implement, and it works well against known threats, but an attacker can circumvent it with only a little effort. Vaccines can help, but they don't provide any security that can be relied upon.

3.2.5 Security perimeters

Security is only as strong as its weakest link. The methods described above can in principle provide a very high level of security even in a very large system that is accessible to many malicious principals. But implementing these methods throughout the system is sure to be difficult and time-consuming. Ensuring that they are used correctly is likely to be even more difficult. The principle of "divide and conquer" suggests that it may be wiser to divide a large system into smaller parts and to restrict severely the ways in which these parts can interact with each other.

The idea is to establish a security perimeter around part of the system, and to disallow fully general communication across the perimeter. Instead, there are GATES in the perimeter which are carefully managed and audited, and which allow only certain limited kinds of traffic (e.g.,

electronic mail, but not file transfers or general network datagrams). A gate may also restrict the pairs of source and destination systems that can communicate through it.

It is important to understand that a security perimeter is not foolproof. If it passes electronic mail, then users can encode arbitrary programs or data in the mail and get them across the perimeter. But this is less likely to happen by mistake, and it is more difficult to do things inside the perimeter using only electronic mail than using terminal connections or arbitrary network datagrams. Furthermore, if, for example, a mail-only perimeter is an important part of system security, users and managers will come to understand that it is dangerous and harmful to implement automated services that accept electronic mail requests.

As with any security measure, a price is paid in convenience and flexibility for a security perimeter: it's harder to do things across the perimeter. Users and managers must decide on the proper balance between security and convenience.

3.2.5.1 Application gateways

3.2.5.1.1 What's a gateway

The term 'gateway' has been used to describe a wide range of devices in the computer communication environment. Most devices described as gateways can be categorized as one of two major types, although some devices are difficult to characterize in this fashion.

- The term 'application gateway' usually refers to devices that convert between different protocols suites, often including application functionality, e.g., conversion between DECNET and SNA protocols for file transfer or virtual terminal applications.

- The term 'router' is usually applied to devices which relay and route packets between networks, typically operating at layer 2 (LAN bridges) or layer 3 (internetwork gateways). These devices do not convert between protocols at higher layers (e.g, layer 4 and above).

'Mail gateways,' devices which route and relay electronic mail (a layer 7 application) may fall into either category. If the device converts between two different mail protocols, e.g., X.400 and SMTP, then it is an application gateway as described above. In many circumstances an X.400 Message Transfer Agent (MTA) would act strictly as a router, but it may also convert X.400 electronic mail to facsimile and thus operate as an application gateway. The multifaceted nature of some devices illustrates the difficulty of characterizing gateways in simple terms.

3.2.5.1.2 Gateways as Access Control Devices

Gateways are often employed to connect a network under the control of one organization (an 'internal' network) to a network controlled by another organization (an 'external' network such as a public network). Thus gateways are natural points at which to enforce access control policies, i.e., the gateways provide an obvious security perimeter. The access control policy enforced by a gateway can be used in two basic ways:

- Traffic from external networks can be controlled to prevent unauthorized access to internal networks or the computer systems attached to them. This means of controlling access by outside users to internal resources can help protect weak internal systems from attack.

- Traffic from computers on the internal networks can be controlled to prevent unauthorized access to external networks or computer systems. This access control facility can help mitigate Trojan Horse concerns by constraining the telecommunication paths by which data can be

transmitted outside of an organization, as well as supporting concepts such as release authority, i.e., a designated individual authorized to communicate on behalf of an organization in an official capacity.

Both application gateways and routers can be used to enforce access control policies at network boundaries, but each has its own advantages and disadvantages, as described below.

3.2.5.1.2.1 Application Gateways as PAC Devices

Because an application gateway performs protocol translation at layer 7, it does not pass through packets at lower protocol layers. Thus, in normal operation, such a device provides a natural barrier to traffic transiting it, i.e., the gateway must engage in significant explicit processing in order to convert from one protocol suite to another in the course of data transiting the device. Different applications require different protocol conversion processing. Hence, a gateway of this type can easily permit traffic for some applications to transit the gateway while preventing other traffic, simply by not providing the software necessary to perform the conversion. Thus, at the coarse granularity of different applications, such gateways can provide protection of the sort described above.

For example, an organization could elect to permit electronic mail to pass bi-directionally by putting in place a mail gateway while preventing interactive login sessions and file transfers (by not passing any traffic other than e-mail). This access control policy could be refined also to permit restricted interactive login, e.g., initiated by an internal user to access a remote computer system, by installing software to support the translation of the virtual terminal protocol in only one direction (outbound).

An application gateway often provides a natural point at which to require individual user identification and authentication information for finer granularity access control. This is because many such gateways require human intervention to select services etc. in translating from one protocol suite to another, or because the application being supported is one which intrinsically involves human intervention, e.g., virtual terminal or interactive database query. In such circumstances it is straightforward for the gateway to enforce access control on an individual user basis as a side effect of establishing a ‘session’ between the two protocol suites.

Not all applications lend themselves to such authorization checks, however. For example, a file transfer application may be invoked automatically by a process during off hours and thus no human user may be present to participate in an authentication exchange. Batch database queries or updates are similarly non-interactive and might be performed when no ‘users’ are present. In such circumstances there is a temptation to employ passwords for user identification and authentication, as though a human being were present during the activity, and the result is that these passwords are stored in files at the initiating computer system, making them vulnerable to disclosure (as discussed elsewhere in the report on user authentication technology). Thus there are limitations on the use of application gateways for individual access control.

As noted elsewhere in this report, the use of cryptography to protect user data from source to destination (end-to-end encryption) is a powerful tool for providing network security. This form of encryption is typically applied at the top of the network layer (layer 3) or the bottom of the transport layer (layer 4). End-to-end encryption cannot be employed (to maximum effectiveness) if application gateways are used along the path between communicating entities. The reason is that these gateways must, by definition, be able to access protocols at the application layer, which is above the layer at which the encryption is employed. Hence the user data must be decrypted for processing at the application gateway and then re-encrypted for transmission to the destination

(or to another application gateway). In such an event the encryption being performed is not really "end-to-end."

If an application layer gateway is part of the path for (end-to-end) encrypted user traffic, then one would, at a minimum, want the gateway to be trusted (since it will have access to the user data in cleartext form). Note, however, there use of a trusted computing base (TCB) for the gateway does not necessarily result in as much security as if (uninterrupted) encryption were in force from source to destination. The physical, procedural, and emanations security of the gateway must also be taken into account as breeches of any of these security facets could subject the user's data to unauthorized disclosure or modification. Thus it may be especially difficult, if not impossible, to achieve as high a level of security for a user's data if an application gateway is traversed vs. using end-to-end encryption in the absence of such gateways.

In the context of electronic mail the conflict between end-to-end encryption and application gateways is a bit more complex. The secure messaging facilities defined in X.400 [CCITT 1989a] allow for encrypted e-mail to transit MTAs without decryption, but only when the MTAs are operating as routers rather than application gateways, e.g., when they are not performing "content conversion" or similar invasive services. The Privacy-Enhanced Mail facilities developed for the TCP/IP Internet [RFC 1113, August 1989] incorporate encryption facilities which can transcend e-mail protocols, but only if the recipients are prepared to process the decrypted mail in a fashion which smacks of protocol layering violation. Thus, in the context of electronic mail, only those devices which are more akin to routers than application gateways can be used without degrading the security offered by true end-to-end encryption.

3.2.5.1.2.2 Routers as PAC Devices

Since routers can provide higher performance, greater robustness and are less intrusive than application gateways, access control facilities that can be provided by routers are especially attractive in many circumstances. Also, user data protected by end-to-end encryption technology can pass through routers without having to be decrypted, thus preserving the security imparted by the encryption. Hence there is substantial incentive to explore access control facilities that can be provided by routers.

One way a router at layer 3 (to a lesser extent at layer 2) can effect access control through the use of "packet filtering" mechanisms. A router performs packet filtering by examining protocol control information (PCI) in specified fields in packets at layer 3 (and maybe layer 4). The router accepts or rejects (discards) a packet based on the values in the fields as compared to a profile maintained in an access control database. For example, source and destination computer system addresses are contained in layer 3 PCI and thus an administrator could authorize or deny the flow of data between a pair of computer systems based on examination of these address fields.

If one peeks into layer 4 PCI, an eminently feasible violation of protocol layering for many layer 3 routers, one can effect somewhat finer grained access control in some protocol suites. For example, in the TCP/IP suite one can distinguish among electronic mail, virtual terminal, and several other types of common applications through examination of certain fields in the TCP header. However, one cannot ascertain which specific application is being accessed via a virtual terminal connection, so the granularity of such access control may be more limited than in the context of application gateways. Several vendors of layer 3 routers already provide facilities of this sort for the TCP/IP community, so this is largely an existing access control technology.

As noted above, there are limitations to the granularity of access control achievable with packet filtering. There is also a concern as to the assurance provided by this mechanism. Packet filtering

relies on the accuracy of certain protocol control information in packets. The underlying assumption is that if this header information is incorrect then packets will probably not be correctly routed or processed, but this assumption may not be valid in all cases. For example, consider an access control policy which authorizes specified computers on an internal network to communicate with specified computers on an external network. If one computer system on the internal network can masquerade as another, authorized internal system (by constructing layer 3 PCI with incorrect network addresses), then this access control policy could be subverted. Alternatively, if a computer system on an external network generates packets with false addresses, it too could subvert the policy.

Other schemes have been developed to provide more sophisticated access control facilities with higher assurance, while still retaining most of the advantages of router-enforced access control. For example, the VISA system [Estrin 1987] requires a computer system to interact with a router as part of an explicit authorization process for sessions across organizational boundaries. This scheme also employs a cryptographic checksum applied to each packet (at layer 3) to enable the router to validate that the packet is authorize to transit the router. Because of performance concerns, it has been suggested that this checksum be computed only over the layer 3 PCI, instead of the whole packet. This would allow information surreptitiously tacked onto an authorized packet PCI to transit the router. Thus even this more sophisticated approach to packet filtering at routers has security shortcomings.

3.2.5.1.3 Conclusions

Both application gateways and routers can be used to enforce access control at the interfaces between networks administered by different organizations. Application gateways, by their nature, tend to exhibit reduced performance and robustness, and are less transparent than routers, but they are essential in the heterogeneous protocol environments in which much of the world operates today. As national and international protocol standards become more widespread, there will be less need for such gateways. Thus, in the long term, it would be disadvantageous to adopt security architectures which require that interorganizational access control (across network boundaries) be enforced through the use of such gateways. The incompatibility between true end-to-end encryption and application gateways further argues against such access control mechanisms for the long term.

However, in the short term, especially in circumstances where application gateways are required due to the use of incompatible protocols, it is appropriate to exploit the opportunity to implement perimeter access controls in such gateways. Over the long term, we anticipate more widespread use of trusted computer systems and thus the need for gateway-enforced perimeter access control to protect these computer systems from unauthorized external access will diminish. We also anticipate increased use of end-to-end encryption mechanisms and associated access control facilities to provide security for end-user data traffic. Nonetheless, centrally managed access control for inter-organizational traffic is a facility than may best be accomplished through the use of gateway-based access control. If further research can provide higher assurance packet filtering facilities in routers, the resulting system, in combination with trusted computing systems for end users and end-to-end encryption would yield significantly improved security capabilities in the long term.

4 References

Bell, D. and LaPadula, J. 1976. *Secure Computer System: Unified Exposition and Multix Interpretation*. ESD-TR-75-306, MITRE Corp, Bedford, MA.

Biba, K. J. 1975. *Integrity Considerations for Secure Computer Systems*, Report MTR 3153, MITRE Corp., Bedford, MA.

Birrell, A. et al. 1987. A global suthenticaion service without global trust. *IEEE Symposium on Security and Privacy*, Oakland, CA, pp 223-230.

Boebert, E. et al. 1985. Secure Ada Target: Issues, system design, and verification. *IEEE Symposium on Security and Privacy*, Oakland, CA, pp 176-183.

CCITT 1989a. *Data Communications Networks Message Handling Systems*, Vol VIII, Fasicle VIII.7, Recommendations X.400-X.420, CCITT, Geneva.

CCITT 1989b. *Data Communications Networks Directory*, Vol VIII, Fasicle VIII.8, Recommendations X.500-X.521, CCITT, Geneva.

Clark, D. and Wilson, D. 1987. A comparison of commercial and military computer security policies. *IEEE Symposium on Security and Privacy*, Oakland, CA, pp 184-194.

Computers & Security 1988. Special Supplement: Computer viruses, **7**, 2, Elsevier, April.

Cullyer, W. 1989. Implementing high integrity systems: The Viper microprocessor. *IEEE AES Magazine*, pp 5-13.

Davies, D. and Price, W. 1984. *Security for Computer Networks: An Introduction to Data Security in Teleprocessing and Electronic Funds Transfers*. Wiley, New York.

Denning, D. 1976. A lattice model of secure information flow. *Comm. ACM* **19**, pp 236-243.

Denning, D. et al. 1988. The SeaView security model. *IEEE Symposium on Security and Privacy*, Oakland, CA, pp 218-233.

Department of Defense. 1985. *Department of Defense Trusted Computer System Evaluation Criteria*. Standard DOD 5200.28-STD (The Orange Book).

Dewdney, A. 1989. Of Worms, viruses, and core war, *Scientific American*, March, pp 110-113.

Diffie, W., and Hellman, M. 1976. New directions in cryptography. *IEEE Trans. Information Theory* **IT-22**, 16 (Nov), pp 644-654.

Estrin, D. and Tsudik, G. 1987. VISA scheme for inter-organization network security. *IEEE Symposium on Security and Privace*, Oakland, CA, pp 174-183.

Gasser, M. et al. 1989. The Digital distributed system security architecture, *12th National Computer Security Conference*, NIST/NCSC, Baltimore, MD, pp 305-319.

Gasser, M. 1988. *Building a Secure Computer System*. Van Nostrand, New York.

Hellman, M. 1979. The mathematics of public-key cryptography. *Scientific American* **241**, 2 (Aug.), pp 146-157.

Kahn, D. 1967. *The Codebreakers*. Macmillan, New York.

Lampson, B. 1973. A note on the confinement problem. *Comm. ACM* **16**, 10 (Oct.), 613-615.

Lampson, B. 1985. Protection. *ACM Operating Systems Review* **19**, 5 (Dec.), 13-24.

Lunt, T. 1988. Automated audit trail analysis and intrusion detection: A survey. *11th National Computer Security Conference*, NIST/NCSC, Baltimore, MD.

Meyer, C. and Matyas, S. 1983. *Cryptography: A New Dimension in Computer Data Security*. Wiley, New York.

Morris, R. and Thompson, K. 1979. Password security: A case history. *Comm. ACM* **22**, 11 (Nov.), pp 594-597.

National Bureau of Standards. 1977. *Data Encryption Standard*. Federal Information Processing Standards Publ. 46. Government Printing Office, Washington, D.C.

NIST 1988. *Smart Card Technology: New Methods for Computer Access Control*, NIST Special Publication 500-157, NIST, Gaithersburg, MD.

National Research Council, 1991. *Computers at Risk: Safe Computing in the Information Age*, Computer Science and Telecommunications Board, National Academy Press, Washington, DC.

Needham, R. and Schroeder, M. 1978. Using encryption for authentication in large networks of computers. *Comm. ACM* **21**, 12 (Dec.), pp 993-998.

Parnas, D. 1972. On the criteria for decomposing a system into modules. *Comm. ACM* **15**, 12 (Dec.), pp 1053-1058.

Rivest, R., et al. 1978. A method for obtaining digital signatures and public-key cryptosystems. *Comm. ACM* **21**, 2 (Feb.), pp 120-126.

Saltzer, J. and Schroeder, M. 1975. The protection of information in computer systems. *Proc. IEEE* **63**, 9 (Sept.), pp 1278-1308.

Saydjari, O. 1987. Locking computers securely. *10th National Computer Security Conference*, National Bureau of Standards, pp 129-141.

Schlichting, R. and Schneider, R. 1983. Fail-stop processors: An approach to designing fault-tolerand computing systems, *ACM Transactions on Computer Systems* **1**, 3 (Aug.), pp 222-238.

Simmons, G. 1988. A survey of information authentication. *Proc. IEEE* **76**, 5 (May), pp 603-620.

Steiner, J. et. al. 1988. Kerberos: An authentication service for open network systems. *Proc. Usenix Winter Conference*, pp 191-202.

Thompson, K. 1984. Reflections on trusting trust. *Comm. ACM* **27**, 8 (Aug.), pp 761-763.

U.S. Congress Office of Technology Assessment. 1986. *Federal Government Information Technology: Management, Security and Government Oversight*. OTA-CIT-297, Government Printing Office, Washington, D.C.

U.S. Congress Office of Technology Assessment. 1987. *Defending Secrets, Sharing Data: New Locks and Keys for Electronic Information*. OTA-CIT-310, Government Printing Office, Washington, D.C.

Voydock, V. and Kent, S. 1983. Security mechanisms in high-level network protocols. *ACM Computing Surveys* **15**, 2 (June), pp 135-171.

5 Glossary

Access control	Granting or denying a principal's access to an object according to the security model
Access control list	A list of the principals that are permitted to access an object, and the access rights of each principal.
Access level	Part of the Orange Book security level.
Accountability	Enabling individuals to be held responsible for things that happen in a system.
Identity based	An access control rule based only on the identity of the subject and object. Contrasted with 'rule based'. See 'discretionary'.
Rule based	An access control rule based on some properties of the subject and object beyond their identities. Contrasted with 'identity based'. See 'mandatory'.
User directed	Access control in which the user who owns an object controls who can access it. Contrasted with 'administratively directed'. See 'discretionary'.
Administratively directed	Access control in which a small number of administrators control who can access which objects. Contrasted with 'user directed'. See 'mandatory'.
Assurance	Establishing confidence that a system meets its security specification.
Auditing	Recording each operation on an object and the subject responsible for it.
Authentication	Determining who is responsible for a given request or statement.
Authorization	Determining who is trusted for a given purpose.
Availability	Assuring that a given event will occur by a specified time.
Bell-LaPadula	An information flow security model couched in terms of subjects and objects.
Capability	Something which is accepted as evidence of the right to perform some operation on some object.
Category	Part of the Orange Book security level.
Challenge-response	An authentication procedure that requires calculating a correct response to an unpredictable challenge.
COMPUSEC	Computer security.
COMSEC	Communications security.
Confidential	Synonym for secret.
Confidentiality	Keeping information confidential.
Countermeasure	A mechanism that reduces the vulnerability of or threat to a system.
Covert channel	A communications channel that allows two cooperating processes to transfer information in a manner that violates security policy.
Crypto ignition key	A key storage device that must be plugged into an encryption device to enable secure communication.
Delegate	To authorize one principal to exercise some of the authority of another.
Depend	A system depends on another system if you can't count on the first system to work properly unless the second one does.
Denial of service	The opposite of availability.
DES	The Data Encryption Standard secret key encryption algorithm.

Digital signature	Data which can only be generated by an agent that knows some secret, and hence is evidence that such an agent must have generated it.
Discretionary access control	An access control rule that can be freely changed by the owner of the object.
Emanation	An signal emitted by a system which is not explicitly allowed by its specification.
Gateway	A system connected to two computer networks which transfers information between them.
Group	A set of principals.
Implementation	The mechanism that causes the behavior of a system, which with luck matches its specification.
Information flow control	Ensuring that information flows into an object only from other objects with lower security levels.
Integrity	Keeping system state from changing in an unauthorized manner.
Kernel	A trusted part of an operating system on which the rest depends.
Key	An input, usually a string of a few dozen or hundred characters, that makes a general encryption or decryption algorithm suitable for secure communication among parties that know the key.
Label	The security level of an object.
Level	see Security level.
Mandatory access control	An access control rule that the woner of the object can't make more permissive; often, a rule based on the security levels of the resource and the requesting subject.
Model	An expression of a security policy in a form that a system can enforce.
Mutual authentication	Authenticating each party to a communication to the other; specifically, authenticating the system that implements an object and verifying that it is authorized to do so.
Non-discretionary	Synonym for mandatory.
Non-repudiation	Authentication which remains credible for a long time.
Object	Something to which access is controlled.
Operating system	A program intended to directly control the hardware of a computer, and on which all the other programs running on the computer must depend.
Orange book	Department of Defense Trusted Computer System Evaluation Criteria.
Password	A secret that a principal can present to a system in order to authenticate himself.
Perimeter	A boundary where security controls are applied to protect assets.
Policy	A description of the rules by which people are given access to information and resources, usually broadly stated.
Principal	A person or system that can be authorized to access objects or can make statements affecting access control decisions.
Protected subsystem	A program that can act as a principal in its own right.
Public key	The publicly known key of a key pair for a public key encryption algorithm
Public key encryption	An encryption algorithm that uses a pair of keys; one key of the pair decrypts information that is encrypted by the other.

Receivers	Principals reading from a channel.
Reference monitor	A system component that enforces access controls on an object.
Requirement	see Policy; a requirement is often more detailed.
RSA	The Rivest-Shamir-Adelman public key encryption algorithm.
Secrecy	Keeping information secret.
Secret	Known at most to an authorized set of principals.
Secret key	A key for a secret key encryption algorithm.
	The secret key of a key pair for a public key encryption algorithm
Secret key encryption	An encryption algorithm that uses the same key to decrypt information that is used to encrypt it.
Secure channel	An information path where the set of possible senders can be known to the receivers, or the set of possible receivers can be known to the senders, or both.
Security level	Data that expresses how secret some information should be; one level may be higher than another if information at the higher level is supposed to be more secret.
Senders	Principals writing to a channel.
Separation of duty	An authorization rule that requires two different principals to approve an operation.
Signature	see Digital signature.
Simple security property	An information flow rule that a subject can only read from an equal or higher level object.
Smart card	A small computer in the shape of a credit card.
Specification	A description of the desired behavior of a system.
Star property	An information flow rule that a subject can only write to an equal or higher level object.
Subject	An active entity that can make request to obtain information from an object or change the state of an object.
System	A state machine: a device that, given the current state and some inputs, yields one of a set of outputs and new states.
	An interdependent collection of components that can be considered as a unified whole.
TCB	Trusted computing base.
Tempest	US government rules for limiting compromising signals from electrical equipment.
Threat	Any circumstance or event with the potential to cause harm to a system.
Token	A pocket-sized computer which can participate in a challenge-response authentication scheme.
Trap door	A hidden mechanism in a system that can be triggered to circumvent the system’s security mechanisms.
Trojan horse	A computer program with an apparently or actually useful function that also contains a trap door.
Trust	Confidence that a system meets its specifications.
Trusted computing base	The components of a system that must work for the system to meet its security specifications.
Vaccine	A program that attempts to detect viruses and disable them.

Virus	A self-propagating program.
Vulnerability	A weakness in a system that can be exploited to violate the system's security policy.