Computer system security evaluation

by PETER G. NEUMANN

SRI International

Menlo Park, California

INTRODUCTION

This paper considers the problem of attaining computer systems and applications programs that are both highly secure and highly reliable. It contrasts two current alternative approaches, one remedial, the other preventive. A remedial approach is outlined based on a classification of software security violations suggested by Bisbey, Carlstedt, and Hollingworth at ISI. This remedial analysis is then related to a preventive approach, illustrated here by the formal SRI Hierarchical Development Methodology. Evaluation of system security is then considered by combining concepts from the preventive and remedial approaches. This combination of techniques seems to have significant potential in the attainment and evaluation of computer system security. Illustrations are given for three types of systems, the first two being systems explicitly designed with security in mind, and the first of those being designed according to a formal methodology. The first system is the SRI design for a Provably Secure Operating System (PSOS), the second is Multics, and the third is UNIX. (The reader familiar with security may wish to skim the next two sections.)

BACKGROUND

Computer systems and applications are increasingly being called on to provide reliable security, although most existing commercial computer systems are incapable of supporting various security-critical applications. Until recently, the task of obtaining highly secure systems has typically been considered to be very difficult, and avoided on the grounds that good solutions would be very expensive. There has been little deep understanding of either how to develop secure systems and applications, or how to assess security; however, there have been efforts to detect insecurity. The interactions between security on the one hand and reliability and survivability on the other have also been largely ignored, as have the interactions with dynamic auditing of security. The term "system defensiveness" is used here to imply security, reliability, survivability, and the auditability of any events that might compromise these aspects of computer behavior.

Recently, the situation has been improving (e.g., see the survey in Shankar). Two approaches to attaining better security are considered here, one basically remedial, the other basically preventive. The first approach involves assessing the security of computer systems (particularly existing ones) and attempting to patch around any security flaws thereby uncovered. A variety of techniques have been used to detect flaws that reduce security. These include searching code for certain patterns of program statements corresponding to classes of would-be security flaws (e.g., Bisbey), as well as more traditional experimental penetration attempts (e.g., McPhee). The second approach involves designing new systems that are intrinsically secure and whose security can in some way be convincingly established. This approach might rely on a suitable design methodology (e.g., Robinson et al.), combining formal specifications (e.g., Parnas, Robinson et al.), suitable design structures, the use of suitable programming languages (e.g., see Lühr), and supporting tools. Examples of such approaches that also support formal verification are found in Robinson et al. and Good.

With respect to the remedial approach, experience in attempting to penetrate allegedly secure systems leads to several observations. First, penetrators do not generally have to work very hard to find major security flaws in traditionally developed systems. Second, patches that attempt to remove such flaws are themselves often flawed. Third, this approach, is intrinsically limited because attempts to characterize the still undetected security flaws are speculative, at best. In general, attempting to retrofit security into a basically insecure system is of limited effectiveness, and inevitably leaves much in doubt.

In the long run, use of the preventive approach is likely to be significantly more productive. This approach should proceed with security as a fundamental design goal from the outset. It may include explicit statements of requirements and formal specifications of the design. It should make use of a modern programming language, and might employ formal proofs of significant properties regarding system behavior, such as security (e.g., see Neumann et al.). On the basis of recent research, it is now possible to assess the security of a precisely specified design, prior to implementation, and then subsequently to assess the security provided by the implementation.

The conclusions of Glasean et al. are extremely pessimistic, at least for the near future, with respect to obtaining quantitative assessments of the cost (risk) associated with various security violations. Similar conclusions are drawn
in the present writing, with regard to attaining meaningful measures of security in conventionally designed systems. However, there are some hopes for attaining secure systems. First, it is helpful to assess insecurity—although it is not very reassuring to know just that another security flaw has been found. Second, it is possible to consider relative measures of insecurity for different classes of would-be violators such as skilled system programmers, data processing supervisors, system operators, skilled users, and casual users. Some security flaws may provide threats only from users in certain of these classes. Knowledge of the would-be violators and the degree to which they are trusted is thus fundamental to any evaluation of risk. Third, there is some short-term hope in a combination of the remedial and preventive approaches in redesigning an existing system so that its security-relevant portions can be isolated into a small and cleanly defined collection of system programs (a "security kernel" plus a few trusted processes, jointly responsible for the security of the system). However, the effectiveness of this combination can be reduced by restrictions imposed on the required redesign, such as the need to maintain compatibility with the original (insecure) system interface. It also may confuse policy (e.g., security strategies) with mechanism (e.g., the protection mechanisms upon which the policy is implemented), potentially embedding high-level policy issues inflexibly into low-level mechanisms. In any case, it is felt that the preventive approach must be involved whenever new systems are to provide substantially more security than existing systems. In particular, the use of a formal development methodology for design and implementation can provide a powerful basis for intrinsic and quantitatively assessable security.

SYSTEM DEFENSES

Defensiveness is considered here as a generalized notion of security, and implies the nonexistence of inappropriate behavior—insofar as possible. For present purposes, defensiveness consists of related and partially overlapping components, namely security, reliability, availability (including recoverability), and auditability.

(1) Security as used here involves the protection of the system, its applications, and the shared resources from misuse. The context considered is purposely a broad one. It includes the notions of preventing unauthorized acquisition and unauthorized modification of information, that is, assuring "confidentiality" and "integrity," respectively. It is intended to cover both system security and user data security. [In usage by MITRE, viz., Biba10 "(data) integrity" is a precise formal dual of multilevel "(data) security" (i.e., confidentiality), the former referring to the writing of data, the latter to the reading of data.] In usage here, security also includes the prevention of denial of service (maintaining the integrity of resources) and the prevention (where possible, or otherwise the limitation) of information leakage through clandestine channels (such as signalling through shared resources or through shared timing channels). (See Lampson, Lipner, Neumann et al. 4) In certain applications the notion of security may refer to or include multilevel (e.g., military) security with levels such as TOP SECRET, SECRET, etc. (See Bell and LaPadula, Feiertag et al. 14)

(2) Reliability, Availability, and Recovery involve (respectively) the correctness of system security (including data integrity) and other system functions; the maintenance of a secure running system; and the restoration of a secure running system following any error, accident, willful damage, or disaster that caused a loss of service or security.

(3) Auditability involves the monitoring of the continued existence of system security and reliability, including the detection of anomalous or potentially threatening behavior.

Continuous maintenance of security depends on appropriate system hardware and software, including concepts of fault-tolerant architecture and reliable software. It also depends on an appropriate operational environment. It requires global planning and management. The implications of an unreliable system on security are particularly insidious, and thus must be considered along with the more typical considerations of security.

It is a difficult task to assure dynamically the continuous absence of all would-be security violations, e.g., to audit misuse of an intrinsically insecure system. It seems much more fruitful to work on the design of better systems, and then to develop auditing procedures as an integral part of the design. (For example, auditing should obey security policies wherever possible.) However, once a breach of system defenses is discovered, it should be fixed as rapidly as possible. In the sense that the effects of a security violation may already have leaked out, it may not be possible to provide dynamic recovery in the strict sense. Thus auditing is important to help limit the propagation. Good design should anticipate the needs of both auditing and recovery, with auditing that strongly limits the propagation effects of detected violations and that simplifies recovery therefrom.

Given an existing system not designed to be secure, the elimination of potential threats becomes more and more difficult after the first few threats are removed. The process is time-consuming, frustrating, unpredictable, and ultimately limited by the knowledge that although fewer threats are being discovered, many more may still remain undiscovered. It is particularly perverse that the changes necessary to remove discovered threats may themselves lead to new violations, particularly in conventional systems.

Given an arbitrary system, it would be useful to have some assessment of the penetrability of the system. However, even the most innocent-looking flaw may in fact be a hole in the dike that leads to total inundation. In general, the risks from system violations can best be avoided by a combination of good design, good implementation, good operations, and good management, all done with defensiveness in mind. A combination of testing and penetration studies,
and possibly some formal verification is recommended, commensurate with the desired system requirements, the expected difficulty of penetration, and the cost of having violations. Formal verification for the most security-critical portions of the design and its implementation is expected to be cost-effective in the near future. Recent techniques for verifying design properties, independent of any implementations, are particularly promising.

The remainder of this paper is organized as follows. First, a categorization of various typical system violations is given, based on the work of Bisbey, Carlstedt, and Hollingworth at ISI. This provides the framework for a remedial analysis of any particular system. A collection of preventive criteria is then given whose presence in the design, implementation, evolution, and operation of new computer systems may result in systems that are far more defensive than the conventional systems available today.

It is felt that these categories of violations and criteria for defensiveness should be very useful for evaluating the development of new systems and applications, as well as existing ones. They are used here first for a consideration of the design of PSOS (a Provably Secure Operating System), intended to be demonstrably secure. Multics and UNIX are then considered as two (extreme) examples of existing systems. It is seen that a system developed according to the preventive criteria intrinsically avoids many of the flaws indicated by the remedial approach.

VIOLATIONS OF SYSTEM DEFENSES

Nielsen et al. have analyzed over 300 cases of computer misuse (from the files of Donn Parker) and have identified seven types of violations to system defenses, including cases of disaster, accidents, hoaxes, threats and extortion. (Defensiveness is therein called "integrity.") However, system violations are involved in the preponderance of cases (e.g., undesired acquisition, modification, insertion, or destruction of data, along with various other forms of system penetration). Some of the cases involve undesired denial of service. The sophistication of leakage through clandestine channels was apparently unnecessary, given the ease of penetration by simpler means. Nevertheless, in deference to Murphy's Law ("If it can happen, it will"), all plausible violations should be anticipated, and covered by various safeguards whenever the threats are deemed significant.

Classes of system penetrations

Bisbey, Carlstedt, and Hollingworth at the USC Information Sciences Institute (ISI) have studied techniques for identifying potentially vulnerable sections of code, including some techniques that could be carried out automatically. In so doing, they identified 10 categories of system flaws, each capable of producing security violations. These categories have evolved over the last few years, with each violation observed at ISI fitting into at least one of these categories. However, there is no pretense that these categories are either canonical or complete. (Although they were conceived primarily to be applicable to software, most of them are in fact also applicable to hardware.)

Two of the 10 ISI categories are closely related and have been lumped together here. For present purposes, the resulting nine topical categories are grouped into four generic groups of flaws. Each has both design and implementation aspects.

(A) improper protection (initialization and enforcement);
(B) improper validation;
(C) improper synchronization;
(D) improper choice of operation or operand.

The nine topical categories can be associated with these groups as follows.

PROTECTION (initialization and enforcement):
(1) improper choice of initial protection domain;
(2) improper isolation of implementation detail;
(3) improper change (e.g., a value or condition changing between its time of validation and its time of use);
(4) improper naming;
(5) improper (incomplete) deallocation or deletion;
VALIDATION:
(6) improper validation;
SEQUENCING:
(7) improper indivisibility;
(8) improper sequencing;
OPERATION CHOICE:
(9) improper operation or operand selection.

These categories are illustrated in Table I. They cover many different security violations, including undesired reading, writing, and deleting, undesired denial of service, and to some extent undesired leakage through clandestine information channels. For example, visibility of implementation detail (category 2) often leads to potential leakage channels. However, leakage channels also exist in systems that are otherwise secure, because of the natural visibility of elapsed time.

It is not claimed that the four generic categories are orthogonal. Nevertheless, they provide very useful typical cases. Within the first generic category ("improper protection"), the five topical categories exhibit considerable overlap. (Category 1 deals with initialization, while categories 2-5 provide paradigms of improper enforcement.) In fact, there exists a sequence of flaws that exhibits an ordering of (1) $\rightarrow$ (2) $\rightarrow$ (3) $\rightarrow$ (4) $\rightarrow$ (5), as follows. An incorrect choice of protection partition (1) can result in the inability to isolate the implementation of an abstraction from the use of that abstraction (2). Such inability can result in a value presented at the time of call (and expected subsequently to remain constant) being changed during execution (3). Such a flaw can result in a naming problem (4), in which two different paths to the (apparently) same data can give different results. Finally, a naming problem may create a residue problem (5), e.g., if a local symbolic name is not unbound when the object is deleted with reference to its global name—or vice versa. (Since each category may reappear at different levels
TABLE I.—Categories of Protection Flaws (and examples) (Based on Bisbey, Carlstedt, Hollingworth at ISI)

1. Incorrect choice of protection domain or security partition
   (a security-critical function manipulating critical data directly accessible to the user; incorrect initial assignment of security or integrity level at system generation, configuration, or initialization)
2. Exposed representations or implementation detail
   (bypassing an abstraction, e.g., direct manipulation of a hidden data structure such as an unmediated user modification of a directory entry; user use of an absolute I-O address; note that the visibility of timing information provides a generic leakage channel, e.g., drawing inferences from page fault activity)
3. Inconsistency of data over time
   (noninvariance of parameters, e.g., change in value of a parameter in a call by reference; change in a file accessible to different processes, e.g., in an improperly protected, shared process directory)
4. Naming problems
   (aliasing, e.g., two distinct names for the same object not being treated identically; ambiguity resulting from use of the same local name for two distinct objects [which may also involve a residue problem, e.g., if name persists in some system table])
5. Residues in allocation and deallocation
   (incomplete deletion, revocation, or deallocation, e.g., such that an apparently deleted value is still accessible—in core, disk, archive store, etc.; incomplete cleanup on abort; ignoring terminal hangup)
6. Nonvalidation of critical conditions and operands
   (invalid or unconstrained parameters such as an out-of-bounds virtual address or absolute I-O address; lack of strong type checking, e.g., a pointer to a structure of the wrong type; absence of quota limit stops such as bounds on queue sizes or number of processes, with overflows resulting in possible system or user crashes)
7. Indivisibility problems (in multiprocessing)
   (interrupted atomic operations, e.g., incomplete interrupt handling [quit during login resulting in partial success, or in continual lockup of interlocked data]; faulty read-after-rewrite in hardware)
8. Serialization problems (in multiprocessing, multiprogramming)
   (incorrect sequencing [e.g., wrong order], improper isolation of atomic operations from one another [e.g., reading during writing, or concurrency among different directory commands on the same directory]; critical race conditions in implementation; deadlocks and deadly embraces)
9. Incorrect choice of operation or operand
   (use of the wrong function, producing incorrect results; use of an unfair scheduling algorithm, producing correct results for each scheduled process, but denying service completely to certain users)

In a hierarchical design, other such orderings may also exist. Thus the most primitive descriptor of these protection flaws is something like "improper domain selection and enforcement."

There is some overlap between the two topical categories of sequencing flaws. The indivisibility and serialization categories, (7) and (8), respectively, are related when considered at different levels of abstraction, since (logically) atomic operations appearing at one level may in fact be implemented out of (logically) atomic operations at a lower level, with appropriate interlocks. That is, a serialization problem at one level may apparently manifest itself as an indivisibility problem at the next higher level.

Validation problems usually involve the omission of a check on argument validity or on quota limits. The inconsistency of data over time (category 3 above) might conceivably be viewed as a validation problem. (This is the time-of-check-to-time-of-use flaw, or "TOCTTOUT," considered by McPhee, Weissman, and others.) However, this view is somewhat like locking the barn door after the horse has escaped: the primitive flaw is not a lack of validation, but rather the lack of either protection or interlocks that permitted the value to change in the first place.

Categories 3, 5, and 6 above have been the subject of experimental study at ISI, including the development of computer tools that search given programs for specific types of violations—notably interprocedure data dependencies, potential inconsistencies of data values over time, and potential residues. (See Bisbey et al.,16,17 Carlstedt et al.18 Carlstedt,19 and Hollingworth and Bisbey.19) However, the analysis of most categories is not generally amenable to systematic investigation.

It is useful here to identify some of the symptoms that may underly flaws of each particular category. Such an identification is attempted in Table II, in which are listed various conditions that can (but not always do) cause flaws in security.

DEFENSIVE SYSTEMS

In general, good design of the hardware and software and good implementation are highly important. In existing sys-

TABLE II.—Symptoms of Potential Protection Flaws, by Category

1. Domain choice. All programs or human actions relating to the initialization or interpretation of protection information are suspect, e.g., any setting or changing of a security level, particularly any action that lessens security (e.g., downgrading).
2. Exposed representations. Any direct visibility or use of implementation detail is suspect. Any use of absolute addresses for memory or input-output. Nonvirtual resources. Direct access to a data structure that is normally used as an abstract data object. Serial dependence within logically combinational functions.
3. Data inconsistency. A called procedure fetches the value of a parameter more than once, or fetches a value it just stored. A parameter is passed by name or by reference. (The value may change between call and return.) An output value is overlaid on top of an input value. Reference is made to a value that is self-modifying upon being accessed. (See Bisbey et al.,20 Carlstedt et al.)
4. Naming. Any object for which two different names can exist is suspect. Use of a local name in one context, a global name in another, e.g., a virtual and a nonvirtual name. Any use of a table index where protection is expected.
5. Residues. Physical deletion of contents is suspect whenever deferred beyond logical deletion, e.g., deferred until reuse of media space. Readable free pools. Accessible backup storage. Reuse of an index or slot number after deletion of entry. (See Hollingworth and Bisbey,20 which enables isolation of all allocations and deallocations, and searches for potential residues.)
6. Nonvalidation. The absence of any checks on protection information upon access to sharable data is usually indicative of a flaw. The absence of any checks on an input variable or parameter, on its type or value range, or even on the existence of data is suspect. Lack of quotas on real resources or on different virtual partitions of resource usage can provide a leakage channel across partitions. Validation of status at the time of a request for which status may change, without revalidation on completion. (See Carlstedt,19 Bisbey et al.)
7. Indivisibility. Any allegedly noninterruptable or indivisible operation is suspect, as is the mechanism for achieving indivisibility.
8. Serialization. Any overlapping of operations using the same data is suspect, either with different uses of the same operation, or simultaneous uses of different operations on the same data base.
9. Choice of operation or operand. This category is very hard to formalize. Potentially any operation can be improperly chosen. Operations and data types that do not correspond are suspect, as is the use of mismatched type declarations.
tems, it is usually bad design and bad implementation that lead to violations of system defenses. Thus, good design and good implementation should themselves be considered as (meta-)safeguards relating to software. The a priori use of good design practices and good implementation practices is in general enormously more effective than the a posteriori retrofitting of patches that attempt to strengthen the defenses.

Factors influencing development of defensive systems

The factors summarized in Table III are relevant to improving the defensiveness of a design and its implementation. Many of these factors have profound effects throughout the development process, involving requirements specifications, design, implementation, system maintenance, long-term evolution, and operation.

Use of a formal development methodology for secure system development

As an indication of how systems can fare with respect to the presence or absence of potential security flaws, a system is considered here that has been conceived under rather unusual—and hopefully optimal—conditions. This system is PSOS, a Provably Secure Operating System. PSOS has been designed from the outset to be able to support advanced security requirements beyond those of existing systems, and has taken advantage of essentially all of the above factors influencing good design and implementation. In particular, it has been designed according to formally stated requirements, has been formally specified, and has been subjected to formal proofs of its critical design properties. It is designed according to the SRI Hierarchical Design Methodology (HDM), described in Robinson et al., Neumann et al., and Robinson and Levitt. The design is represented in a formal language, SPECIAL (A SPECification and Assertion Language, Roubine and Robinson). The system has not yet been implemented, but the nature of would-be implementations has been characterized. The design has been oriented toward provability of the design and of subsequent implementations. (HDM is also being used in the design and implementation of the SIFT computer system for an ultrareliable commercial aircraft application [for which proofs of elementary fault-tolerance properties have been undertaken], the design of a real-time operating system, the design of a family of related message processing systems, and the development of provable security kernels.)

PSOS is a capability-based system in which each capability acts as a protected name or token for an object. A capability for an object is protected, in that its creation and its transfer to other users or processes can be controlled, and once created, it can never be altered. It protects the object to which it refers, in that its presentation is required for that object to be accessed, including access rights appropriate to the operation being performed. (Certain PSOS applications take advantage of store-limited capabilities, which cannot be transferred out of the containing process.)

<table>
<thead>
<tr>
<th>TABLE III.—Factors Influencing Defensiveness in Systems and Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Well-defined and well-understood requirements, established clearly and agreed upon in advance</td>
</tr>
<tr>
<td>Good design (e.g., modularly structured, especially hierarchically, with strict isolation of application programs and system programs, strongly typed operations, unified treatment of storage, input-output, e.g., mapped virtual access)</td>
</tr>
<tr>
<td>Suitable implementation languages (e.g., strong typing, avoidance of aliasing, constrained argument passing [such as use of call by value where data inconsistency may be a problem], hiding of implementation detail and device dependence wherever possible, clean control structures, encapsulation of data types)</td>
</tr>
<tr>
<td>Well-defined and understandable specifications for the system hardware and software</td>
</tr>
<tr>
<td>Structured implementation, reflecting the modularity of the design wherever appropriate, and structured initialization (e.g., hierarchical)</td>
</tr>
<tr>
<td>Systematic handling of exception conditions and quota limits</td>
</tr>
<tr>
<td>Auditing and recovery integrated into system design, e.g., hierarchical</td>
</tr>
<tr>
<td>Careful debugging, testing, verification</td>
</tr>
<tr>
<td>Good management of system development (e.g., respecting these factors)</td>
</tr>
<tr>
<td>Lessening the need for management as a result of simplifications resulting from use of these factors</td>
</tr>
<tr>
<td>Good management of system operation (e.g., rigid adherence to system generation and evolution protocols)</td>
</tr>
<tr>
<td>Nonreliance on secrecy of design and implementation</td>
</tr>
<tr>
<td>Awareness of the user community (e.g., enforcing the use of random pronounceable passwords rather than guessable ones)</td>
</tr>
</tbody>
</table>

If formal verification of the design or its implementation is desired, then the following also contribute, both separately and collectively:

<table>
<thead>
<tr>
<th>Formally stated requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Formally specified design, including specifications of modules and their interrelationships (e.g., data representations)</td>
</tr>
<tr>
<td>Formal proofs of correspondence between design specifications and requirements</td>
</tr>
<tr>
<td>Formal axiomatization of the programming language</td>
</tr>
<tr>
<td>Formal proofs of consistency of programs with design specifications</td>
</tr>
<tr>
<td>Formal axiomatization of the hardware/microcode</td>
</tr>
<tr>
<td>Formal proofs of consistency of hardware/microcode with hardware specifications</td>
</tr>
</tbody>
</table>

An entity analogous to a capability exists in SPECIAL, called a designator. A designator serves as a protected name of an object. Its uniqueness in SPECIAL is part of the specification language. In the implementation of PSOS, this uniqueness can be easily guaranteed by the use of capabilities. (Other solutions exist in other systems, such as descriptors.)

Table IV gives an indication of how the use of the methodology together with the use of a suitable programming language can overcome each of the nine categories of flaws summarized in Table II. It is seen that the use of SPECIAL has a very significant impact on the avoidance of these flaws in design. Most of the comments on the design specification in the table are generally applicable to any system specified in SPECIAL. Similarly, the use of the hierarchical design methodology has significant impact on the avoidance of these flaws in implementation.

The use of a suitable modern programming language can also contribute considerably. A language such as Euclid, Modula, Texas’ Gypsy, or SRI’s HDM-compatible ILPL
TABLE IV.—The Influence of the SRI Methodology and Programming Languages (PL) on Avoiding Characteristic Flaws, both in General and in PSOS (P:)

<table>
<thead>
<tr>
<th>Category of Flaw</th>
<th>Design (in SPECIAL)</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Domain choice</td>
<td>Design proofs.</td>
<td>Program proofs.</td>
</tr>
<tr>
<td>2. Exposed representations</td>
<td>Hidden by hierarchical levels of abstraction.</td>
<td>PL encapsulation; proofs.</td>
</tr>
<tr>
<td></td>
<td>P: Extended types further aid hiding.</td>
<td>P: Extended type mechanism.</td>
</tr>
<tr>
<td>3. Data inconsistency</td>
<td>None. No overwrite. Conceptual call by value only.</td>
<td>PL call by value; proofs.</td>
</tr>
<tr>
<td></td>
<td>P: Argument values on stack unalterable.</td>
<td>P: Residues protected.</td>
</tr>
<tr>
<td>4. Naming</td>
<td>Unique designators, nonbypassable, only means of naming.</td>
<td>PL no aliasing; proofs.</td>
</tr>
<tr>
<td></td>
<td>P: Capabilities unique.</td>
<td>P: Uniqueness and nonbypassability of capabilities aided by tagging.</td>
</tr>
<tr>
<td>5. Residues</td>
<td>None in specifications. Deallocated values become UNDEFINED, are thereafter unnamable.</td>
<td>PL strong typing; compile or run-time checking.</td>
</tr>
<tr>
<td></td>
<td>P: Residues protected.</td>
<td>P: nonreusable identifiers; contents zeroed if desirable on deallocation.</td>
</tr>
<tr>
<td></td>
<td>Strong type checking, explicit subtypes, explicit exception and error conditions.</td>
<td>Completion or non-effect exception provable.</td>
</tr>
<tr>
<td>7. Indivisibility</td>
<td>Specifications for each function are logically indivisible.</td>
<td>Benignness of overlap provable.</td>
</tr>
<tr>
<td>9. Operation choice</td>
<td>Faulty specifications detected by spec proofs. Type checking also helps.</td>
<td></td>
</tr>
</tbody>
</table>

Many of the characteristic flaws can be avoided in design by the use of a methodology such as the SRI methodology, and in implementation by the use of a suitable programming language.

Note: "PL" refers to language features found in such languages as Euclid, ILPL, and the emerging DoD/1 languages. "P:" refers to PSOS. (See text.)

(due to Larry Robinson, and documented in Neumann et al.) would be particularly helpful, as might the languages emerging from the DoD/1 effort. (ILPL is an extremely basic intermediate-level programming language that gains its simplicity from the power of the supporting methodology, whose formal specifications provide [for example] its data structures as a by-product of the hierarchical levels of abstraction.) Some of the major contributions that such languages can make are indicated generically by the "PL" entries in Table IV. In most cases these contributions result from intrinsic properties of the language. In other cases they may result from simply enforceable language restrictions. Note that not all of the above languages have all of the desired facilities. (At present, Euclid and ILPL [the latter taken together with HDM] are probably the most complete in this respect. Modula and Gypsy are currently being extended, and support for Gypsy and ILPL is currently being developed.)

Finally, for many of these flaws, relatively simple proofs are possible to demonstrate that the particular flaw is absent in various explicit forms. The design aspects result from properties of the specifications, e.g., they follow from proofs of consistency between the specifications and formal requirements, or are intrinsic to the methodology and the specification language. The implementation aspects result from proofs of program correctness, or are intrinsic to the use of the programming language. With this methodology, it is sufficient to show consistency between the programs and their specifications. However, many proofs are possible without having to prove program correctness in general. Nevertheless, program proofs are becoming feasible as tools to support them continue to be developed.

The comments in Table IV are generically applicable to systems designed and implemented according to the methodology. In addition, those prefixed with a "P:" are particularized to the design and the would-be implementation of PSOS (e.g., in one of the above-mentioned languages). It is seen that a system designed according to a formal methodology like HDM is likely to avoid the characteristic flaws without much additional effort. In particular, PSOS has none of these flaws intrinsic to its design, and it is believed that an implementation essentially free of these flaws can be developed—and if desired, proved.

Above and beyond merely avoiding the nine characteristic problems, the use of HDM has a significant impact on the integrity of PSOS in other ways. These include the explicit statement of formal requirements for security, formal specifications for the design and its hierarchical structure, a unified design, a systematic approach to the handling of exception conditions and quota limits, and a structured implementation capable of taking advantage of the hierarchical design without losing much in efficiency. These factors contribute to the suitability of the design and implementation, and to the possibility of carrying out formal proofs, if desired. This additional impact is summarized in Table V.

Application of this evaluative approach to other systems

The evaluation of PSOS with respect to security flaws is clearly favorable. This is not surprising, in that PSOS was designed with security in mind from the outset, using the formal SRI methodology (which itself evolved simultaneously as a result of its application to PSOS). However, it is instructive to consider two other types of systems: the first which designed with security in mind, but without a formal methodology; the second designed with security only superficially in mind for its implementation. The examples taken are Multics and UNIX. (Conventional commercial operating systems are ignored here, as they are for the most
Formal specifications of the design. These provide precise specifications for the system. Their consistency with the formal requirements can be formally proved. They provide the basis for implementation, as well as the basis for proofs of consistency of the implementation and the design. Proven specifications also provide a strong basis for compatible alternate implementations on the same or different hardware.

Hierarchical structure of the design. This encourages separation of policy and mechanism, and simplifies the initial implementation significantly. Recovery is done hierarchically, as is initialization. Recovery is therefore predictable and controllable. This increases the persistence of security during failure (partial or total).

Unified design. Auditing functions conform to the security requirements. Provable layered security kernel designed (albeit in 1965) to take advantage of what was then known about advanced design and implementation techniques. UNIX is a system that was developed at Bell Labs to take advantage of then recent operating system experience (e.g., Multics), on a much smaller scale—with judicious choice of simplifications intended to give high efficiency. However, UNIX was designed to be a system operating in a cooperative user environment, and was not conceived as a system providing any rugged sense of security. Nevertheless, it is a widely used and highly useful system. The discussion here is not intended in any way as a condemnation of UNIX as an insecure system, but rather as a simple illustration of how insecure a system can be in the absence of a pervasive concern for security from the outset.

Table VI presents a summary of how Multics and UNIX fare with respect to the nine characteristic problems. It is clear from the table that Multics is relatively secure. It is equally clear that UNIX is not.

Table VI presents a comparison in terms of the methodological concepts of Table V. After considerable improvement upon the implementation of a basically sound design, Multics has reached a level of significant security, and has become difficult to penetrate. Although it was subjected to various penetration efforts on its earlier hardware, and was indeed penetrated, the current hardware and software have long since remedied the flaws permitting those penetrations.

UNIX has thus far apparently been used only in benign part intrinsically insecure.) In essence, Multics is a large-scale general-purpose system with security superior to other commercial systems, with some emphasis on its being able to recover from outage, and some emphasis on audit. UNIX, on the other hand, makes little pretense of being secure, except for its attempts to provide access control and its encryption of the password file. It is fundamentally insecure. Multics, like PSOS, is a system that was innovatively designed (albeit in 1965) to take advantage of what was then known about advanced design and implementation techniques.

<table>
<thead>
<tr>
<th>Category of Flaw</th>
<th>Multics</th>
<th>UNIX</th>
</tr>
</thead>
<tbody>
<tr>
<td>2. Exposed representation</td>
<td>Access within a ring all or nothing. Considerable hiding via ring mechanism.</td>
<td>Argument itself is copied, but only by convention. Processes-temporary files are alterable in mid-computation by user or other users.</td>
</tr>
<tr>
<td>3. Data inconsistency</td>
<td>Argument pointer is copied onto stack by call. Arguments are copied within the system code to avoid this flaw internally.</td>
<td>Default on directory entries: unprotected. Memory aliases (&quot;/dev/mem&quot;). Trojan horses abound.</td>
</tr>
<tr>
<td>4. Naming</td>
<td>Collisions of local names: Trojan horse!</td>
<td>Core zeroed only before reallocation, but is not virtually addressable after deallocation. Disk never zeroed, just overwritten. Residues after crash.</td>
</tr>
<tr>
<td>5. Residues</td>
<td>Core zeroed only before reallocation, but is not virtually addressable after deallocation. Disk never zeroed, just overwritten. Residues after crash.</td>
<td>Core zeroed only before reallocation, and is until then still readable. Disk never zeroed, just overwritten. Residues after crash.</td>
</tr>
<tr>
<td>6. Nonvalidation</td>
<td>Hardware checking at ring and segment levels. All kernel args validated. Compile-time data-type checking.</td>
<td>Very few checks on resource or I-O bounds, interuser ops. Easy to crash.</td>
</tr>
<tr>
<td>8. Serialization</td>
<td>Locks prevent overlap. Lock hierarchy used to avoid deadlocks.</td>
<td>OK? System functions logically synchronous (until IPC installed).</td>
</tr>
<tr>
<td>9. Op choice</td>
<td>Possible problems?</td>
<td>Possible problems?</td>
</tr>
</tbody>
</table>
environments, and is at present simple to penetrate. It appears that major renovation would be required to substantially improve its security (although a planned new release of UNIX is expected to eliminate a few of the problems).

From the methodological considerations of Table VII, Multics again appears to be better off than UNIX. However, from this vantage point, UNIX does not seem to be irreparably insecure. For example, a reimplementation with the imposition of constraints on C (or modifications to the language) and better handling of exceptions would be beneficial.

It is interesting to note that each of these three systems has been considered as a basis for supporting multilevel security. The PSOS design is augmented with a multilevel security policy manager that can be efficiently implemented on top of PSOS (or within it, if that is to be the only policy). Multics and UNIX both have experimental redesigns retrofitting a kernel responsible for system security into the existing design. MIT, Honeywell, and the Air Force have undertaken the restructuring of the existing Multics system to improve (among other things) its security. As a component of that work, Honeywell has designed a Multics security kernel, for which formal requirements and formal specifications exist (using the SRI methodology). However, the funding for that project (Guardian) has expired before the kernel could be implemented. UCLA27 and MITRE have independently designed and implemented prototype security kernels to be retrofitted into UNIX. These prototypes have provided the impetus for a competitive design effort between two groups (FORD-Aerospace and SRI as one group, TRW as the other), expected to result in a demonstrably secure kernel upon which is built a secure version of UNIX.

CONCLUSIONS

This document attempts to take a broad view of the development of secure systems, combining a remedial approach with a preventive approach. In particular, it characterizes various common flaws that can lead to security violations. It then uses this characterization to evaluate a constructive methodological approach to computer system development, and shows how such a methodology can intrinsically tend to avoid the characteristic flaws. This combination is thought to be useful for analyzing existing systems and for developing new systems.

Given a computer system whose design did not originally have security as a major goal, and which consequently may be flawed, it is in general extremely difficult to converge on a secure system by successively patching flaws as they are recognized. Nevertheless, the search for such flaws is desirable. In general, given a system that has evolved after extensive penetrate-and-patch efforts, or given a computer system that has been expressly designed to be secure, it is still a difficult matter to assess how secure the system really is. The evaluative approach presented here is felt to be useful. However, compatibly with this approach, recent advances in formal proof seem increasingly applicable. Such advances indicate the feasibility of proofs of the security of a design, given a suitable formal representation of the design and formal statements of security requirements. These proofs may be carried out prior to and independent of any particular implementation of that design. In addition, proofs of selected critical properties are feasible. Furthermore, the technology for handling proofs of consistency between design specifications and programs implementing those specifications is progressing well. Various semiautomatic computer tools for carrying out such proofs now exist and are being integrated. However, the approach outlined here is aimed at improving the resulting system dramatically, even if no program proofs are ever carried through.

A goal that emerges here is to be able to gain increasing confidence in the design and implementation throughout a system development, and to minimize reimplementation late in the development process by avoiding fundamental design flaws early in the process. The combination of approaches discussed here is seen to be relevant. A highly secure system can be attained, with a well-conceived design, careful implementation, selected formal proofs (particularly proofs of design properties), and the use of penetration efforts. However, it should be emphasized that a completely secure system is unlikely in any case, partly because of the intrinsic problems presented by leakage through timing channels.

PSOS is an example of a new system design whose stated goals from the beginning included support for advanced security requirements and provability. The experience gained in that and related efforts is used here to establish criteria that appear to be useful in helping to evaluate other efforts. For example, the ISI criteria and the methodological considerations clearly show some of the strengths of PSOS and Multics. It is hoped that this paper will lead to further refinements in security evaluation, and that it will be useful in evaluating other systems. It is also hoped that it will influence the subsequent development of secure computer systems.

ACKNOWLEDGMENTS

The writing of this paper has been supported by National Science Foundation Grant No. DCR 74-23774. The author is grateful to Richard Bisbey, Jim Carlstedt, and Dennis Hollingworth for sharing some of their experience, and to
REFERENCES

2. Private communication. See also other references below under Bisbey, Carlstedt, and Hollingworth.