A TAXONOMY OF SECURITY FAULTS IN THE UNIX OPERATING SYSTEM

A Thesis

Submitted to the Faculty

of

Purdue University

by

Taimur Aslam

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science

August 1995

This thesis is dedicated to my parents

ACKNOWLEDGMENTS

I would like to thank my thesis advisor Dr. Eugene Spafford for guiding me through my reaserch. Dr. Spafford provided me the insight to tackle different problems, read several drafts of the thesis and made invaluable comments. I also thank him for being so patient and willing to work with my odd schedule. I also thank my committee members: Dr. Aditya Mathur, Dr. Samuel Wagstaff, Dr. Michal Young. Dr. William Gorman provided invaluable feedback to improve the format of this thesis.

I am greatly indebted to my parents for their support and encouragement over the years. I would not have been able to make it this far without their support. I am also grateful to my sister, for her support in so many different endeavors and for being a great friend.

I thank Dan Schikore, Mark Crosbie, Steve Lodin, and Muhammad Tariq, who proofread manuscripts of my thesis and made invaluable comments. Thanks to all my friends and everybody in the COAST lab, who provided much needed support and made sure that I maintained my sanity. Thanks to Frank Wang for implementing the front-end to the database prototype.

This acknowledgement would not be complete without mentioning Sadaf and some very special people who have always been an inspiration to me.

This work was supported, in part, by: gifts from Sun Microsystems, Bell Northern Research, and Hughes Research Laboratories; equipment loaned to the COAST group by the U.S. Air Force; and a contract with Trident Data Systems. This support is gratefully acknowledged.

THIS PAGE IS INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Taimur Aslam, M.S., Purdue University, August 1995. A Taxonomy of Security Faults in the Unix Operating System. Major Professor: Eugene H. Spafford.

Security in computer systems is important to ensure reliable operation and protect the integrity of stored information. Faults in the implementation can be exploited to breach security and penetrate an operating system. These faults must be identified, detected, and corrected to ensure reliability and safe-guard against denial of service, unauthorized modification of data, or disclosure of information.

We define a classification of security faults in the Unix operating system. We state the criteria used to categorize the faults and present examples of the different fault types.

We present the design and implementation details of a database to store vulnerability information collected from different sources. The data is organized according to our fault categories. The information in the database can be applied in static audit analysis of systems, intrusion detection, and fault detection. We also identify and describe software testing methods that should be effective in detecting different faults in our classification scheme.

Security is defined relative to a policy and involves the protection of resources from accidental or malicious disclosure, modification or destruction [web88]. Computer security can be broadly categorized as:

Physical security: Physical security is the protection of computer facilities against physical threats and environmental hazards. It involves the use of locks, identification badges, guards, personnel clearances, and other administrative measures to control the ability to approach, communicate with, or otherwise make use of any material or component of a data processing system [A^+76].

Information security: Information security is the protection of data against accidental or unauthorized destruction, modification or disclosure. It involves both physical security measures and controlled access techniques [A^+76].

Interest in computer security stemmed from protecting information related to matters of national security and grew as computers were used to store medical and financial records of individuals, and proprietary information for business organizations [Zwa92].

The security of a computer system is defined by security requirements and is implemented using a security policy. The requirements vary according to the level of security desired. Computer systems operate with *implicit trust*: the reliability of a computer system depends on the reliability of its components. The security of a computer system thus involves security of its hardware and software components.

The critical role played by operating systems in the operation of computer systems makes them prime targets for malicious attacks. An operating system is penetrated if an unauthorized user gains access to the system, or if an authorized user causes a security breach by exploiting a flaw in the source code. A *total system penetration* occurs when an unprivileged user gains the ability to execute privileged commands [H^+80]. A penetration of a system can lead to one or more of the following consequences [Den83, BS88, Lin75]:

- Denial of service: Users may be denied access to legitimate services.

- Degradation of performance: Performance can be so poor that the system is not usable.
- Disclosure of information: A user gains unauthorized access to protected information.
- Modification of data: A user modifies information in an unauthorized manner.

Security breaches result from operational faults, coding faults, or environment faults. Operational faults consist of configuration errors and policy errors. Coding faults include programming logic errors, faults of ommision and faults of commision. Environment faults occur when programmers do not pay sufficient attention to the execution environment. These faults include errors that result from limitations of the operational environment, and errors introduced when some functional modules operate in an unexpected manner.

To ensure the reliable operation of a computer system, the implementation should conform to the security requirements and should not contain any vulnerabilities: faults that may be exploited to breach security. Faults introduced during software development can be exposed by software testing techniques. Faults can be prevented in software by using formal methods of verification at each stage of development. These two aspects of developing reliable software are discussed in the following sections.

0.1 An Overview of Software Testing Methods

The classic software engineering life-cycle paradigm for software engineering, also known as the “waterfall model,” is a sequential approach to the software development process. The process begins at the system level and progresses through requirements analysis, design, coding, testing and maintenance phases [Pre92, GJM91]. These steps may not be followed in the specified order and some may be omitted. The phases of development are briefly described below.

Systems engineering and analysis: Often the software being developed is part of a larger system. Systems engineering establishes the requirements for all the different system components and defines how the software must interface with other components.

Requirements analysis: During this phase, requirements-gathering is focused on the software being developed and performance and interfacing requirements for software are defined. During requirements analysis, a *requirements specification document* is developed that describes what the requirements phase has accomplished [GJM91]. The requirements specifications must be stated in a precise, complete, consistent and understandable manner. This clarity ensures that the requirements specifications document encompasses all the expectations of the users and enables designers to develop software that meets the requirements [IEE90, GJM91]. During requirements analysis, a *functional specifications document* may also be produced that describes how the system will perform the intended task [IEE90].

Design: Software design focuses on four attributes of the program: data structures, software architecture, procedural detail, and interface characterization [Pre92]. The design phase results in a *design specifications document*, that contains a description of the system architecture: a description of what each module in the system is intended to do and the relationships among modules [GJM91]. During this phase, the design process is documented and is often translated in a representation that can be assessed for quality.

Coding: During the coding phase, the design is translated using one or more appropriate programming languages.

Testing: A fault is induced by an error (a mistake by a human) and may lead to failure [IEE90]. During the testing phase, the focus is on finding faults in the software that may have been introduced during the design or coding phases.

Maintenance: After the software has been developed and released, it may undergo changes to adapt to different environments and errors may be introduced in the process. Software maintenance re-applies each of the preceding life-cycle steps to an existing program.

The time spent in the testing phase is often dependent on the project. Pressman [Pre92] and Ghezzi [GJM91] report that for typical projects about 30-40% of the total development time is spent in testing and debugging. According to Myers [Mye76] the maintenance and testing costs for typical installations of IBM OS/360 exceeded 80% of the total cost.

The objective of the testing phase is to discover any faults that may have been introduced during the coding or design phase. Although it is referred to as the testing phase, fault detection is not restricted to conventional testing methods and any fault detection technique may be employed.

Fault detection techniques can be classified as *dynamic analysis* or *static analysis*. Dynamic analysis techniques require the execution of the program being tested and most conventional testing methods belong to this category [YT91]. Static analysis consists of code walk-throughs, and inspection of requirements and specification documents to find flaws in the implementation. Figure 0.1 from [How81a, How81b] illustrates the dichotomy of the analysis techniques.

Given a program P and specifications S, it is undecidable to determine if P conforms to S [YT91]. The presence of faults is generally an undecidable property. Thus, it is not even theoretically possible to devise a fault detection technique that is applicable to arbitrary programs [YT91]. Every practical technique incorporates a compromise between accuracy and effort. Figure 0.2 adapted from [YT91] shows the relationship between effort expended and different types of testing methods.

	Static	Dynamic
Requirements	Informal checklists Formal modeling	Functional testing Testing by classes of input data Testing by classes of output data
Design	Static analysis of design documents	Design-based testing
Programs	General information Static error analysis Symbolic execution	Structural testing Expression testing Data-flow testing

Figure 0.1 A Conventional Taxonomy of Software Modeling and Analysis Techniques

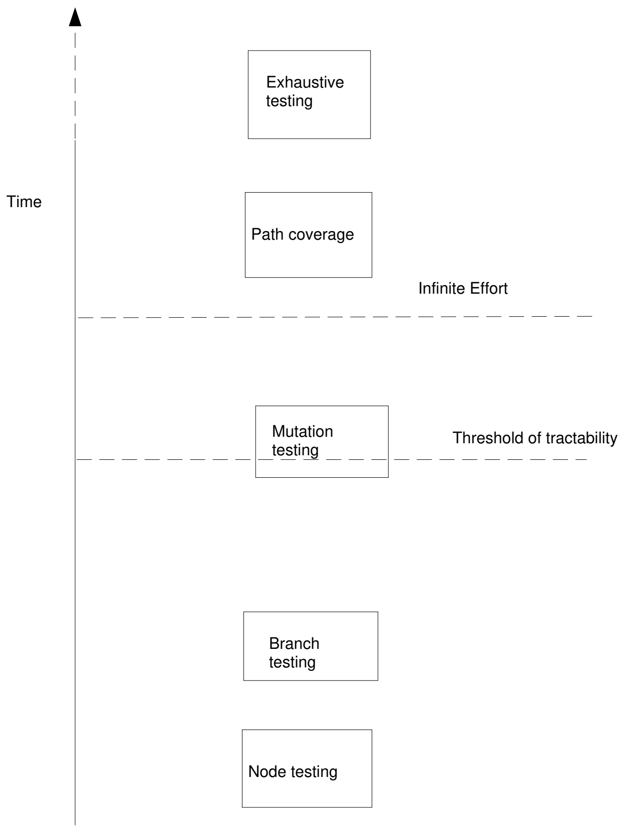

Figure 0.2 Effort vs. Testing Methods

The objective of software testing is not to show that a given program is free from faults but to demonstrate their presence [Mye79]. Exhaustive testing of a program would require infinite effort and time to find all the faults in a program. As a compromise between time and effort, conventional testing methods rely on an *adequacy* criteria to determine that adequate testing has been done and the test data being used is *adequate*. A test data set is adequate if the program under test runs successfully on the data set and all incorrect programs run incorrectly [DMMP87].

Several techniques for measuring the adequacy of test data have been proposed. Path testing associates the adequacy of test data with the traversal of a subset of all paths through the program under test [CPRZ89]. Mutation testing is another method for determining the adequacy of test data and is based on inducing faults in the program under test [DMMP87].

Another method of classifying dynamic fault detection techniques is as:

Functional testing: In functional (black box) testing, the internal structure and behavior of the program is not considered. The objective is to find out solely when the input-output behavior of a program does not agree with its specifications [DMMP87].

Structural testing: In this approach, the structure of the program is examined and test cases are derived from the program's logic. Structural testing is also known as white box testing [DMMP87].

Software testing is further refined into several categories by Myers [Mye76] as described below. Fig 0.3 from [Mye76] illustrates the relationship between different types of testing methods and their relationship to the design process.

Unit testing: Unit testing is a white box oriented approach and focuses verification efforts on a per module basis. It uses a detailed design description to exercise important control paths within the module boundaries.

Integration testing: Integration testing is used to uncover interfacing errors between unit-tested modules.

External testing: External testing is the verification of the external system functions as stated in the system specifications.

System testing: System testing attempts to fully verify the complete functionality of the system. System testing is a verification process when it is done in a simulated environment; it is a validation process when it is performed in a live environment.

Acceptance testing: Acceptance testing is the validation of the system or program according to the user's requirements.

Installation testing: Installation testing is the validation of each particular installation of the system with the intent of discovering any errors made while installing the system.

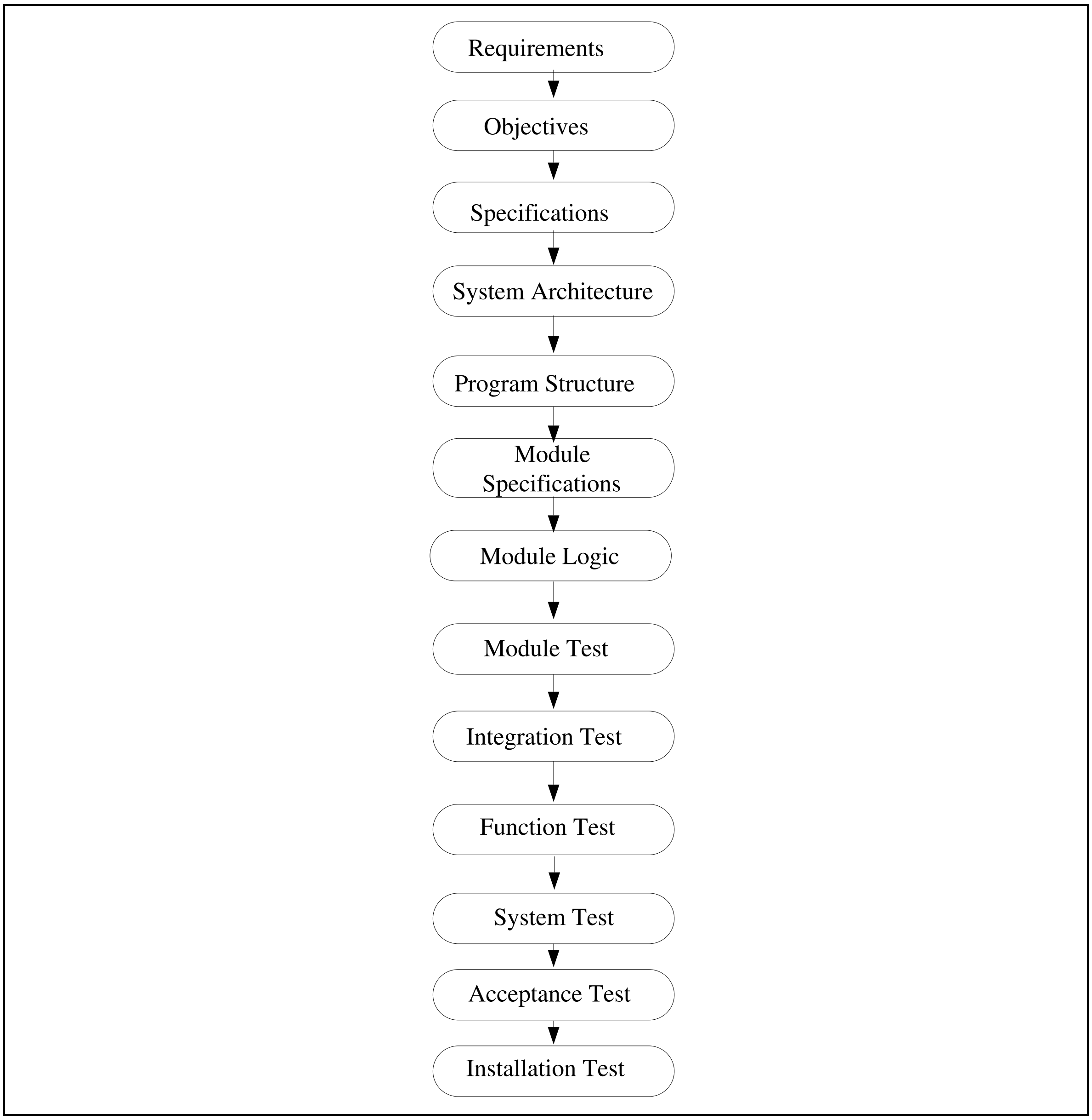

Figure 0.3 Relationship between Testing Methods and the Design Process

0.2 Provable Security and Formal Methods

Software testing techniques can only establish the presence of faults not their absence [Mye79]. Formal methods of verification are used to ensure that faults are not introduced during different stages of development and the implementation is formally proven to conform to the requirements and specifications. The applicability of formal methods has been mostly restricted to proving small sections of programs to make the process tractable. Hall in his introduction to formal methods [Hal90] states:

> "Formal methods are controversial. Their advocates claim that they can revolutionize [software] development. Their detractors think they are impossibly difficult. Meanwhile, for most people, formal methods are so unfamiliar that it is difficult to judge the competing claims."

Security kernels were designed to provide provable security and evolved from the idea of a reference monitor described by Anderson [And72]. A reference monitor is an abstraction of access checking mechanisms [Den83]. The idea behind security kernels is to have a small nucleus of software that is responsible for administering the security policy of the entire system. The implementation of a security kernel is formally verified to demonstrate that it conforms to the security requirements.

The requirements of a security kernel are stated in precise mathematical terms and are known as the *criteria* for the system. The requirements are followed by stating the specifications that precisely describe the function of the software in mathematical terms. These specifications are independent of any implementation details and the system is coded using a high order language (HOL). Finally, the implementation is formally demonstrated to conform to the original requirements.

Several security kernels were developed to provide provable system security. These included the Kernelized Secure Operating System [MD79], MITRE security kernel [Sch75], MULTICS with AIM [SCS77], and the UCLA Data Secure Unix (DSU) [PKKe79]. However, security kernels failed to gain wide-spread popularity because of high development-costs, mediocre performance, and difficulties encountered in maintaining the systems.

0.3 Security Testing

Software testing is a cost effective method to detect faults in software [Mye79]. Several software testing techniques have been proposed and have been studied widely [Bei83, Mye76, Mye79]. These techniques provide cost-effective and systematic methods to detect faults in software. Similarly, security testing is intended to assess the trustworthiness of the security mechanisms [Bei83]. Security testing is often regarded as a special case of system testing [Bei83]. The emphasis of security testing is not to establish the functional correctness of the software but to establish some degree of confidence in the security measures [Bei83].

At present there is no systematic approach to security testing. Traditional methods of detecting security faults include penetration analysis, and formal verification of security kernels [Lin75, MD79]. Penetration analysis is conducted by a *tiger team*, which is comprised of personnel familiar with the abstract and implementation details of operating systems [Lin75]. Penetration analysis has proven to be a successful method in assessing the security of a system [Lin75, H$^+$80, Wil81, AMP76]. But the success of this method depends on the competence of the tiger team members and their knowledge of the system.

The importance of security in computer systems cannot be ignored. It is needed for reliable operation, and to maintain the integrity of stored information. However, there does not exist a systematic approach to detect or prevent security faults in operating systems. Traditional security assessment methods are based on a "penetrate and patch paradigm" where faults are fixed as they are discovered. The shortcoming of this approach is succinctly summed up by Schell [Sch79a] as:

> "Penetrate and patch is a losing approach to assure secure systems because the hacker needs to find only one flaw, whereas the vendor must find and fix all the flaws."

In the following section we discuss the motivation for developing a fault classification scheme, and its applications in making the process of fault prevention and detection more systematic.

0.4 Applications of Fault Categories

In this thesis we define a classification of security faults in the Unix operating system and present applications of the classification scheme. Faults can occur in different sections of source code, and can result in a wide range of consequences. A fault classification scheme can be used to categorize faults that share a common characteristic. The categories can be used to collect statistics about faults and devise methods for fault prevention and detection. Beizer [Bei83] summarized the importance of fault classifications as:

> "It is important to establish categories for bugs if you take the goal of bug prevention seriously. If a particular kind of bug recurs or seems to dominate the kinds of bugs you have, then it is possible through education, training, new controls, revised controls, documentation, inspection, and a variety of other methods to reduce the incidence of that kind of bug. If you have no statistics on the frequency of bugs, you cannot have a rational perspective on where and how to allocate your limited bug prevention resources."

The objective of our fault classification scheme was to unambiguously classify security faults into non-overlapping categories. We applied the fault categories for data organization in a vulnerability database and it was necessary to derive non-overlapping categories.

The vulnerability database we designed can be used to store vulnerability information from different sources, and maintain security patch information. We identified several potential applications of the vulnerability information in the database. The information can be used to analyze commonly occurring faults and trace their origin to a design or programming practice. If the fault cannot be corrected, defensive mechanisms may be programmed in a system that take preventive actions when the fault occurs.

The vulnerability information can also be used to configure audit tools that attempt to find known vulnerabilities in a system [FS91, SSH93]. The information in the database can also be used to write signatures that can be used by an intrusion detection system to detect intrusions in real-time [KS94]. Furthermore, the database may be used to devise thought experiments during the flaw hypothesis stage of a penetration analysis of a system [Lin75].

As security testing is a special case of software testing [Bei83], the fault categories can be used to devise software testing techniques that expose faults in each category. For instance, DeMillo and Mathur [DM91] used fault categories to detect errors reported by Knuth in the development of TEX[Knu89]. Based on fault characteristics, we identify software testing techniques that can be used to detect these characteristics and expose faults in each category. These testing techniques can provide a more systematic approach to fault detection as compared to the "penetrate and patch" paradigm. This testing can be performed as part of system acceptance testing after the software has been developed. Alternatively, these techniques can be used as part of a penetration analysis to detect security faults, or as a guideline to develop new attacks during the flaw hypothesis stage.

0.5 Organization of the Thesis

In Chapter 1 we present a summary of related work in fault classification and penetration analysis. We present a revised taxonomy of security faults in the Unix operating system in Chapter 2. The design and implementation of a vulnerability database is discussed in Chapter 3. In Chapter 4 we discuss software testing methods that can be applied to detect security faults. In Chapter 6 we present our conclusions and outline future work.

1. RELATED WORK

Faults in operating system software can lead to security breaches. Knowledge of the different types of faults, their general characteristics, and the time they enter the system is important to ensure reliable operation and to preserve the integrity of stored information. These topics were the focus of fault classification studies that were conducted to make computer systems secure and to improve the reliability of software. Another approach to find security faults is to assume the role of an intruder and assess the security mechanisms by trying to break into the system. This approach is the Flaw Hypothesis Methodology used in penetration analysis studies.

In the following sections we outline the previous research conducted on security fault analysis, penetration analysis, and software fault classification schemes. We conclude with a discussion of the shortcomings of the previous fault classification schemes.

1.1 Protection Analysis Project

The Protection Analysis (PA) Project conducted research on protection errors in operating systems during the mid-1970s. The group published a series of papers, each of which described a specific type of protection error and presented techniques for finding those errors. The proposed detection techniques were based on pattern-directed evaluation methods, and used formalized patterns to search for corresponding errors [CBP75]. The results of the study were intended for use by personnel working in the evaluation or enhancement of the security of operating systems [BPC75]. The objective of the study was to enable anyone with little or no knowledge about computer security to discover security errors in the system by using the pattern-directed approach. The errors that were found in the systems were classified into four representative categories as follows:

- Domain errors, including errors of exposed representation, incomplete destruction of data, incomplete destruction of content, and incomplete destruction of context

- Validation errors, including boundary condition errors and failure to validate operands

- Naming errors, including aliasing and incomplete revocation of access to a deallocated object.

- Serialization errors, including multiple reference errors and interrupted atomic operations

The PA project did not achieve its goal of fully automating the error detection process by using the pattern-directed approach. The proposed techniques could not be automated easily and their vulnerability database was never published. However, the group's research was an important step in formalizing the process of error detection. Some pertinent work is outlined below.

Inconsistency of a Single Datum Over Time

Bibsey et al. [BPC75] described a general class of problems in which a value becomes inconsistent after it is verified. These errors are also referred to as time-of-check-to-time-to-use (TOCTTOU) errors. These type of errors could be classified as validation errors in the error classification scheme proposed in the final report of the PA project [BH78].

If the value of a variable in a critical function is inadvertently or intentionally replaced with a different value after it has been verified, the operating system can be fooled into using this new value. This can lead to a security breach if the variable is part of a critical check because the entire check will be invalidated.

[BPC75] illustrated how a critical value change can lead to a security breach by a user-callable routine shown below.

```
connect:PROCEDURE(directory, password, code);
          CALL password_check(directory, password, code);
          IF code = 'ok' THEN
                      user_directory = directory;
          END;
```

`connect` is a user-callable supervisor procedure that allows users to change their current directory to a different one than that was specified at login. The procedure requires as arguments the name and password of the new directory and returns the result of the request to the user in the variable `code`. `connect` calls another supervisor procedure `password_check` to verify the correctness of the {directory,password} pair. A return code of `ok` indicates that the directory was changed, otherwise an error code is returned.

The `connect` procedure assumes that the value of `directory` in line 4 is the same as that checked by `password_check`. It also assumes that the value of `code` in line 3 is the same as that returned by `passwd_check`. The entire password check would be invalidated if the values of `code` and `passwd_check` were changed after they had been checked but before being used. The value of the variables could be changed in one of the following ways [BPC75].

1. If the parameter was passed by name or reference, the value remains in the caller's address space throughout the execution of the program. Any asynchronous process that has write access to the caller's address space can change the value.

2. If the parameter was passed by name or reference, the calling procedure could arrange that the input and output variables occupy the same memory location.

When the called procedure stored into the output location, it would overwrite the value of the input parameter.

The consistency of a data value between successive operations may be formulated as the policy :

(B,M,X) $\Longrightarrow$ for some operation L occurring before M, either

[for operation L which does not modify Value(X),

Value(X) before L = Value(X) before M], or

Value(X) after L = Value(X) before M.

Informally, process B can perform operation M on variable X only if the value of X when M is performed is equal to the value of X either before or after some operation L which occurs before M.

The error statement corresponding to the policy statement is given as:

B: M(X) and for some operation L occurring before M,

[for operation L which does not modify Value(X),

Value(X) before L != Value(X) before M], and

Value(X) after L != Value(X) before M.

Process B performs operation M on variable X and the value of X at the time operation M is performed is not equal to the value of X either before or after some operation L which occurs before M.

The error statement could be used to form a search pattern to detect if such an error exists in the source code. Bibsey [BPC75] reported that forty-seven MULTICS routines were searched using an error pattern and seven of them were found to contain errors that could be classified as TOCTTOU errors.

Allocation/Deallocation of Residuals

This section describes errors in operating systems that result from exposed content and incomplete revocation of access to storage elements. Errors of this type are classified as naming errors in the classification proposed in the final report of the PA project [BH78].

A common strategy used in penetration attacks is *scavenging* or searching for residual information from other processes [HB76]. Residuals can be classified into [HB76]:

Content residual: Contents of file space or memory are allocated to a new process without the data from the previous allocation having been purged.

Attribute residual: Information such as the size or position in the free pool of resource is made available to a different process.

If attribute and content residuals are preserved, they can be exploited by communicating processes to get information about the activity of the process [HB76]. Two important attribute residuals that can disclose security related information are cell[1] size, and inter and intra-cell relationships. For example, if the cell size information is available to a user, it may be used to guess the maximum length of the password field, which can reduce the number of permutations required in a brute force attack. Similarly, the relative position of different cells in the free pool can be used to deduce that the previous cell allocations were part of a larger process-defined cell.

Content residual information can be prevented from being disclosed by formulating a deallocation policy that either destroys the contents of storage cells when a cell is added, or after the cell is extracted from the free pool. The latter is less desirable because a functional error in the system may lead to the contents being exposed while the cell is in the free pool. The deallocation mechanism can also be designed such that particular attribute information is destroyed before a cell is added to the free pool.

In addition to content residuals, access residual information also poses a security problem and should be prevented. An access residual occurs when a process can still access a storage cell after it has deallocated it, or an access path to it has been established as a side effect of another request.

[1] a unit of storage

Hollingworth [HB76] did not propose a formal search strategy to detect allocation/deallocation errors. It was suggested that the allocation and deallocation mechanisms should be manually checked to ensure that content, attribute, or access residuals are not preserved.

Serialization

Serialization refers to the ordering of operations relative to one another. Serialization errors result from errors in ordering specifications, either at the policy or mechanism level [Car78]. Serialization errors qualify as protection errors because they can be exploited to corrupt data objects that consequently result in invalid operations. However, serialization errors are underestimated as security errors, because they usually do not manifest themselves directly and rarely occur during the operation of a system [Car78]. Carlstead [Car78] provides an in-depth explanation of serialization errors, including a theoretical treatment of the subject.

1.2 RISOS Project

The RISOS project was a study of computer security and privacy conducted in the mid-1970s [A+76]. The project was aimed at understanding security problems in existing operating systems and to suggest ways to enhance their security. The systems whose security features were studied included IBM's OS/MVT, UNIVAC's 1100 Series operating system, and Bolt Beranek and Newman's TENEX system for the PDP-10.

The final report of the project discussed several issues related to data security in general. It suggested administrative actions that could prevent unauthorized access to the system and methods to prevent disclosure of information. The main contribution of the study was a classification of integrity flaws found in the operating systems studied.

The fault categories proposed by researchers of RISOS [A+76] are the following.

- Incomplete parameter validation

  Processes in an operating system communicate with each other using procedure calls. To maintain reliable operation, the parameters must be validated for data types, number and order, value and range, access rights to storage location, and consistency [A+76]. These checks must specially be enforced when a procedure call is made by a process that requests services from another process with a higher set of privileges. Failure to validate the parameters completely or thoroughly may result in an amplification of access rights of a subject, which may be exploited to cause a security breach.

- Inconsistent parameter validation

  Inconsistent parameter validation is a design error and different from parameter validation errors. If there are multiple sets of validation criteria in a system that are not consistent, an inconsistent parameter validation error occurs when a routine evaluates a condition that it considers valid.

- Implicit sharing of privileged/confidential data

  If information from a higher privileged process or user becomes available to a lesser privileged process or user, an implicit disclosure of information has taken place. To preserve the integrity of stored information and security of the system, it is important that such errors be prevented from occurring in the operating system software.

- Asynchronous-validation/Inadequate-serialization

  This category is comprised of errors that occur if serialization is not enforced during the time a value is stored and the time it is referenced (time-of-check-to-time-to-use errors). These errors can usually be exploited to penetrate a system during a small timing window.

- Inadequate identification/authentication/authorization

Authorization refers to the controlled granting of access rights and is based on authentication of individuals and resources [A+76]. Operating system security may be compromised if:

  - it does not require authorization for an individual or process
  - it does not uniquely identify the resources it is dealing with

- Violable prohibition/limit

  Operating system code contains many fixed-sized data structures such as tables, queues, and stacks. If these data structures overflow, the system may behave in unexpected ways that may lead to a security compromise.

- Exploitable logic errors

  Some errors in operating system software may exist because of improbable timing conditions, incorrect error handling, or instruction side-effects. For example, handling an error condition before an error signal is raised, or using half word arithmetic to store a return address from a routine can be regarded as exploitable logic errors.

1.3 Flaw Hypothesis Methodology

System Development Corporation (SDC) conducted penetration analyses of seven government and industrial installations to assess the secure-worthiness of their systems. This led to the formulation of the Flaw Hypothesis Methodology that was successfully used in these penetration studies.

Linde [Lin75] summarizes the importance of penetration analysis as:

> "In the absence of more formal correctness proof techniques, penetrations are the most cost effective method for assessing vulnerabilities. Exhaustive testing of an operating system's security controls is different than subjecting them to a penetration attack. System testing is intended to

discover implementation errors; whereas penetration tests are used to examine an implementation and from these analyses infer areas of possible design weaknesses."

The Flaw Hypothesis Methodology is a strategy for penetrating an operating system as well as for isolating generic functional flaws that can be used for determining weaknesses in the functional areas of the operating system design [Lin75]. The four stages of the methodology are outlined below.

Knowledge of system's control structures: For a successful penetration, a penetrator should have an understanding of how the users interact with the system, what services are available to them and the constraints placed on those services. In addition, a penetrator should have an abstract as well as implementation level knowledge at the following levels:

- inter-module knowledge
- access control mechanism
- control object hierarchy
- intra-module design
- implementation

Flaw hypothesis: During the flaw hypothesis stage, the penetration team makes a hypothesis about the existence of a vulnerability in the system by studying system source code and any related documentation.

Flaw hypothesis confirmation: This stage of the methodology involves confirming the flaw hypothesis by using live attacks or by writing programs to exploit the potential vulnerability.

Flaw generalization: In this stage, the penetration team makes a generalized hypothesis about the existence of a flaw based on their knowledge of errors in other similar systems, or by analyzing flaws discovered in different functional modules of the system under study.

Based on the successful penetrations, Linde [Lin75] observed that some modules were found to be more vulnerable than others and it is recommended that these should be inspected in a penetration analysis. The functional modules that should be inspected for security flaws include:

- Input/Output control
- Program and data sharing
- Access Control
- Installation management/operation control

The main contribution of this study was the formulation of the Flaw Hypothesis Methodology for system penetration. This methodology was later successfully used in several penetration analysis studies [H+80, AMP76, Wil81]. Despite its usefulness, the methodology has some drawbacks. It does not provide a systematic approach to testing the security mechanisms. Also, the success of a penetration analysis depends on the knowledge and expertise of the tiger team conducting the study. As with most testing techniques, penetration testing can only show the existence of security flaws — not their absence.

1.4 Case Study: Penetration Analysis of the Michigan Terminal System

Presented in this section is a case study of a penetration analysis conducted by a graduate level computer science class at the University of Michigan, Ann Arbor [H+80]. The goals of this study were for an authorized and knowledgeable but unprivileged user to gain unauthorized access to files and data, and crash the system.

MTS was a general purpose operating system for IBM 360/370 computers and supported both batch and interactive computation. The system under study maintained about 2,500 user accounts and could support 250 users concurrently. The members of the tiger team conducting the penetration analysis were students enrolled in the course and had several years of experience in using the system. The team members had access to all user-level documentation and manuals, as well as internal system documentation.

The penetration team employed the Flaw Hypothesis Methodology to find flaws in the operating system and discovered a number of flaws in some of the system subroutines. It was found that the parameter checking mechanisms were weak and would allow a user to store arbitrary bit sequences in the system segment. The system segment in MTS contained much of the system's accounting and protection information. Thus, by storing an arbitrary bit sequence a user could modify the system's accounting information, change his privilege level and completely disable the protection mechanisms. As a consequence, users could execute privileged instructions, disable hardware storage protection and modify shared segments. In MTS, the supervisor code is also contained in the shared segment. Thus, by forcing an appropriate bit pattern to be stored in the shared segment, users could switch the system to supervisor mode and gain complete control over the system. This is the also known as total system penetration and is the ultimate goal of a penetration study [H$^+$80].

Some other flaws were also found in the system that resulted in non-total system penetrations. One such flaw allowed users to bypass the normal accounting update and get free computer time by forcing a job into abnormal termination. It was also discovered that some shared memory segments contained sensitive information such as user passwords and tape identification that could lead to disclosure of information. In the denial of service attacks, the system could be forced to loop forever while in supervisor mode, when presented with a particular instruction sequence.

The study concluded that penetration analysis, if properly done, can provide an actual estimate of the amount of time and effort needed to penetrate a system [H$^+$80]. Even though this work factor may be subjective and imprecise, it still can give a valid assessment of a system's overall security and highlight its areas of strength and weaknesses. The study also highlighted the interdependence of different modules as an important area of consideration because system routines execute with an implicit trust on each other. Any security violation at the lowest security level can lead to a cascading effect that can cause security of the entire system to be compromised.

1.5 Software Fault Studies

Marick [Mar90] published a survey of software fault studies from the software engineering literature. Most of the studies reported faults that were discovered in production quality software. Outlined in this section is a summary of the results from various software fault studies. For brevity, the citation index is used for reference to the study. A brief description of these studies is included in Appendix A.

- Faults in programming logic are common.

  Programming logic errors refer to errors that are introduced in the software because of a misunderstanding of the problem area [Mar90]. These errors may be introduced in the software because of incorrect design, or incorrect logic.

  Studies by Dniestrowski [DGM78], Lipow [Lip79], and Motley [MB77] found that programming logic faults were the largest type of faults. In other studies, Rubey [Rub75] and Glass [Gla81] also regarded programming logic faults as being serious.

- Faults of omission are important.

  Most faults in the survey were faults of commission and few were faults of omission [Mar90]. Glass [Gla81] studied faults that were discovered after delivery of the software. He observed that faults of omission are the most likely faults to survive. Basili and Rombach [BR87] found that most interface faults were omissions, while data handling and computation faults were mostly faults of commission.

- Data handling is more error-prone than computation.

Data handling refers to the process of initializing and changing variables, to distinguish from computation faults, which include evaluation of arithmetic and boolean expressions [Mar90].

Basili [BR87] and Glass [Gla81] reported that data handling faults were numerous in their study. Studies by Rubey [Rub75] Basili and Perricone [BP84], Lipow [Lip79], Motley [MB77], and Ostrand and Weyuker [OW84] reported such faults to be of average frequency.

- Static and dynamic detection techniques are approximately equally effective.

  Neither detection technique can detect all the faults in the implementation. Static analysis techniques have proven to be successful at exposing 30-70% of logic and design errors in a typical program [DMMP87]. Dynamic techniques can expose run-time errors that may not be detected by static analysis. According to four studies reported by Marick [Mar90] there is no convincing evidence that one technique outperforms the other. The percentage of errors detected using the two techniques is shown in Table 1.1.

Study	**Static Analysis**	**Dynamic Analysis**
[Rub75]	44%	56%
[SB75]	55%	45%
[DGM78] Weak complexity modules	70%	30%
[DGM78] Greater complexity modules	50%	50%
[WB85]	29%	40%

Table 1.1 Percentage of Errors Detected

Note: In the study by Dniestrowski et al. [DGM78] weak complexity modules refer to modules containing mostly arithmetic and boolean expressions. Greater complexity modules contain I/O drivers, and real-time management code.

- There is evidence that both small and large modules are more error-prone than medium sized modules.

  Studies by Basili and Perricone [BP84] and Shen et al. [Y+85] observed that smaller modules have higher fault rates. This observation is also confirmed by Withrow [Wit90] in her study of Ada programs.

- No conclusion about development phases is possible.

  The percentage of faults introduced during the different phases of the software life cycle vary from 26% to 83%, making it difficult to draw any conclusive decisions about the time a fault was introduced.

1.5.1 Fault Categories

Rubey [Rub75] presented several fault categories and the percentage of fault occurrence in a study of small commercial real-time control programs. The results are shown in Table 1.2.

Fault Category	**Percentage**
Incomplete or erroneous specifications	28
Intentional deviation from specification	12
Violation of programming standards	10
Erroneous data access	10
Erroneous decision logic or sequencing	12
Erroneous arithmetic computations	9
Invalid timing	4
Improper handling of interrupts	4
Wrong constant or data value	3
Inaccurate documentation	8

Table 1.2 Fault Categories by Rubey

Weiss [WB85] reported 143 faults in a project to develop a hardware architecture simulator coded in 10,000 lines of Fortran. The faults included in the study were reported during development and after delivery. The fault categories by Weiss [WB85] are shown in Table 1.3. In the categories proposed by Weiss [WB85] language refers to misunderstanding a Fortran language construct, and clerical and careless ommision refers to a misunderstanding of the design concept.

Fault Category	**Percentage**
Requirements	6
Design	19
Interface	6
Coding Specifications	13
Language	8
Coding Standards	2
Careless ommision	10
Clerical	36

Table 1.3 Fault Categories by Weiss

Table 1.4 presents results from a comparative study published by Potier [PAFB82]. Potier collected over 1000 faults found in a compiler for TLR (a high level language). Most of the faults were detected during testing. Potier et al. compare their findings with those published in [Lip79] and [MB77].

Category	**[PAFB82]**	**[Lip79]**	**[MB77]**
Computational	6%	9%	9%
Logic	38%	26%	26%
I/0	2%	14%	16%
Data handling	15%	18%	18%
Interface	19%	16%	17%
Data definition	19%	3%	1%
Data base	1%	7%	4%
Others	0%	7%	9%

Table 1.4 Comparative Results by Potier

Beizer [Bei83] collected a sample of 126,000 program statements (mostly executable) and 2070 faults from fault studies conducted by Dniestrowski et al. [DGM78], Endres [End75], Rubey et. al [RDB75], and Schneidewind [Sch79b]. Based on these statistics, Beizer proposed a classification scheme and categorized the faults into the different categories. These categories and the fault statistics are shown in Table 1.5.

Category	Number	Percentage of occurrence
FUNCTIONAL		
Specification	404	
Function	147	
Test	7	
Total	558	27
SYSTEM		
Internal Interface	29	
Hardware(I/O devices)	63	
Operating System	2	
Software Architecture	193	
Control or Sequence	43	
Resources	8	
Total	338	16
PROCESS	27	
Arithmetic	141	
Initialization	15	
Control or Sequence	271	
Static Logic	13	
Other	12	
Total	560	27
DATA	36	
Type	34	
Structure	34	
Initial Value	51	
Other	120	
Total	201	10
CODE	78	4
DOCUMENTATION	103	5
STANDARDS	166	8
OTHER	62	3

Table 1.5 Bug Statistics by Beizer

1.6 Errors of TeX

TeX is a typesetting software developed by D.E. Knuth. Knuth presented a chronological record of errors discovered during the development process and proposed a classification scheme to organize the errors [Knu89]. The classification scheme was based on essential functionality rather than on the external form of the program [Knu89]. This error classification is included in our study because it is representative of errors made by competent and experienced programmers. A revised classification of these errors was used by DeMillo and Mathur [DM91] to detect errors using mutation testing.

The fault categories proposed by Knuth are as follows.

A. an algorithm gone awry

B. a blunder or a mental typo

C. a clean-up for consistency or clarity

D. a data structure debacle

E. an efficiency enhancement

F. a forgotten function

G. a generalization or growth of ability

I. an interactive improvement

L. a language liability

M. a mismatch between modules

P. a promotion of portability

Q. a quest for quality

R. reinforcement of robustness

S. a surprising scenario

T. a trivial typo

In Knuth's classification, A,B,D,F,L,M,R,S, and T represent bugs which had to be fixed for the correct functioning of the program [Knu89]. Categories C,E,G,I,P, and Q represent enhancements which were introduced to improve performance [Knu89].

1.7 A Taxonomy of Computer Program Security Flaws

Landwehr et al. [L+93] published a collection of security flaws in different operating systems and classified each flaw according to its genesis, or the time it was introduced into the system, or the section of code where each flaw was introduced. This study was motivated by the observation that the history of software failures is mostly undocumented and a study that cataloged system failures could help system designers in building better and secure systems. The authors remark that:

> "Knowing how systems have failed can help us build systems that resist failure."

An outline of the three categories in the taxonomy is presented below.

By Genesis: The genesis of a flaw is categorized as malicious or non-malicious. Malicious flaws are intentionally introduced in the system to cause a security violation. These take the form of viruses, worms, Trojan horse, time bombs, and trap doors in the code [L+93]. Non-malicious flaws are introduced because of an implementation error, missing requirements, or misunderstanding of design logic. Most non-malicious flaws can belong to one of the following categories.

- Validation errors
- Domain errors
- Serialization/aliasing errors

- Errors that result from inadequate identification/authentication
- Boundary condition errors
- Logic errors

By time of introduction: Flaws are categorized according to the time in the software life cycle that they were introduced into the system. Time of introduction includes flaws that were introduced during development, maintenance, and operation.

By location: In this category, flaws are classified based on their location in operating system routines, support software, or user programs. Flaws in operating system are further classified into categories corresponding to the functional modules where they are discovered.

1.8 Comparison of Security Fault Classification Schemes

In chapter 2 we present a taxonomy of security faults in Unix. The objective of our fault classification scheme was to unambiguously classify faults into non-overlapping categories. This classification was then used for data organization in the design of a vulnerability database and to identify fault detection techniques. In this section we discuss the shortcomings of the existing security fault classification schemes in applying them for data organization.

The research conducted by the Protection Analysis Project was aimed at formulating pattern matching techniques to detect security faults in the system source code [CBP75]. The search strategies were based on the use of formalized patterns and analyzed the syntactic structure of the program. An error was detected if a sequence of operations matched the corresponding error pattern. The final report of the project proposed four representative categories of faults [BH78]. These fault categories were designed to group faults based on the syntactic structure and were too broad to be used for data organization. Each category in the classification consisted of several types of faults. This would have made it difficult to unambiguously classify faults and would not have allowed specific queries to be performed on our database.

Seven generic fault categories were published in the final report of the RISOS project [A+76]. These fault categories were developed with the goal of better understanding the faults and to classify and analyze new faults as they were discovered [A+76]. The fault categories proposed by researchers of RISOS were general enough to classify faults from several operating systems. But the generality of the fault categories would have prevented fine-grained queries to be performed on our database. Also, the criteria used for the classification was not clearly stated. This could lead to ambiguities when classifying different faults and a fault could be classified in more than one category.

Landwehr et al. [L+93] proposed a taxonomy of flaws found in different operating systems. The objective of this taxonomy was to identify the time during the software life-cycle that a fault was introduced in the system. The taxonomy categorized flaws according to genesis, location in the system software, and time of introduction. These three fault categories were further divided into sub-categories. This fault classification was not used for data organization in our database because faults could not be categorized into Landwehr's proposed categories based only on their description. For example, to classify a flaw as either being malicious or non-malicious involves a decision about the intentions of the programmer. This is difficult to judge, without access to the source code and knowledge of the programming environment. Also, it is not possible to correctly predict the time a flaw was introduced in the system based only on its description [Mar90]. Such a decision could not be made without knowledge of detailed progress reports and milestones met during a project. Similarly, the location where a flaw was discovered cannot be easily classified without access to the source code.

2. A TAXONOMY OF SECURITY FAULTS IN THE UNIX OPERATING SYSTEM

In this chapter we present a taxonomy of security faults in the Unix operating systems. We present the motivation for developing the taxonomy, followed by a discussion of the different fault categories in the classification.

Security errors in operating systems are software faults that are intentionally or inadvertently introduced in the system and can cause a security breach: an action that violates a security policy. The operating system plays an important role in the operation of a computer system and the presence of a security fault poses a potential threat to disrupt reliable operation of the system. A security fault may be exploited to cause denial of service, disclosure of information, or degrade performance of the system. To prevent unauthorized and authorized users from disrupting reliable operation, the security mechanisms of an operating system should be able to withstand malicious attacks and the implementation should not contain any faults that may be exploited to breach security.

Security of computer systems is important so as to maintain reliable operation and to protect the integrity of stored information. With the wide-spread use of computers and increasing computer knowledge it is no longer possible to implement security through obscurity [GS91]. To ensure that computer systems are secure against malicious attacks, we need to analyze and understand the characteristics of faults that can subvert security mechanisms. A classification scheme can help in the understanding of fault characteristics by categorizing faults that share common characteristics. This classification can be used to organize vulnerability data, gather statistics about frequency of faults, and devise prevention or detection techniques.

2.1 A Taxonomy of Security Faults

In this chapter we present a taxonomy of security faults in the Unix operating system. The motivation of the study was to unambiguously classify security faults into distinct categories. This classification was used as the basis for data organization of a vulnerability database that facilitated different queries to be performed on the stored data. The specific characteristics of faults in each category have also been used to identify software testing techniques that can detect those faults. These testing methods can provide a systematic approach to detect security faults as compared to the traditional "penetrate and patch" paradigm.

Our fault classification scheme is not unique. Several security fault classification schemes have been proposed that categorized faults according to different criteria. These include the classifications proposed in the final report of the Protection Analysis Project [BH78], a classification proposed by researchers of the RISOS project [A+76], and a taxonomy of flaws proposed by Landwehr et al. [L+93]. However, these classifications are not suitable for data organization because the categories are too generic, and do not clearly specify the criteria used for the classification. The latter often leads to ambiguities that can result in a fault being classified in more than one category. Our taxonomy addresses these shortcomings. The fault categories we present are specific and distinct. We also specify a selection criteria for each fault category.

Our taxonomy was derived from security faults in the Unix operating system that led to a security compromise. Other taxonomies discussed earlier [A+76, L+93, BH78] were based on faults collected from several different operating systems. We focused on Unix because of its wide-spread popularity in academia and industry. Thus, a large number of users would benefit from the results of a study based on Unix vulnerabilities. The security faults were collected from a number of different sources including Computer Emergency Response Team (CERT) advisories, vulnerabilities reported on an electronic security mailing list, and a survey of the literature.

After collecting the security faults, we gathered information on how each fault was exploited, the functional area a fault was found in, versions of operating systems that each fault was present, and the consequences that occurred when the fault was exploited. This information was collected from references in literature, descriptions of faults in the response team advisories, and discussions with experienced personnel. After understanding the details, we broadly classified faults as either coding faults, operational faults, or environment faults.

Personnel, communication, physical, and operations security play an essential role in the reliable operation of computer systems [ISV95]. Vulnerabilities in any of these classes may be exploited to cause a security compromise. We have focused on faults that are embodied in the software. We have not included vulnerabilities that involved personnel, communication, physical, or operations security. Examples of vulnerabilities in these security classes include:

Personnel security: An employee is bribed to reveal sensitive information.

Communication security: A wiretap is installed to monitor information on a channel.

Physical security: An intruder or an employee steals a disk or tape containing sensitive information.

Operations security: An operator fails to run software that could detect potential security weaknesses.

Coding faults are comprised of faults that were introduced during software development. These faults could have been introduced because of errors in programming logic, missing or incorrect requirements, or design errors. These observations are drawn from different software fault studies that list the aforementioned errors as potential problem areas [Rub75, WB85, PAFB82, Bei83, Knu89]. Operational faults result from improper installation of software. Most policy errors can be classified as operational faults. Environment faults result when a programmer fails to completely understand the limitations of the run-time environment or interactions between functionally correct modules. These faults are dependent on the operational environment.

Coding and operational fault categories were further refined into distinct subcategories. For coding faults, we studied the faults to identify the specific coding error that led to the fault. We tried to abstract each implementation error to a level that would maintain the specific characteristics of each category yet hide the implementation details. This approach can be beneficial in classifying faults in more than one programming language. The fault categories were then used to group different faults that shared similar characteristics. For operational faults, we identified the specific operation that had led to the fault and used it to categorize the faults.

In the following sections, we present our taxonomy of faults. For each fault category we present a description of the fault, the criteria used, and examples of the different fault types.

The taxonomy of faults is comprised of the following categories:

Operational Faults

- Configuration errors to include errors resulting from:
  - programs/utilities installed in the wrong place
  - programs/utilities installed with incorrect setup parameters
  - programs/utilities or secondary storage objects installed with access permissions that violate the security policy

Coding Faults

- Synchronization errors to include:
  - race condition errors
  - errors resulting from improper or inadequate serialization of operations
- Condition validation errors to include errors resulting from:
  - missing conditions
  - incorrectly specified conditions

– missing predicates in the condition

Environment Faults

Environment faults include:

– errors resulting from a limitation of the operational environment

– errors introduced by a faulty compiler or operating system

– interaction errors between functionally correct modules

– errors introduced becuase exception handling mechanisms are different than expected

Figure 2.1 shows a diagrammatic representation of our fault taxonomy.

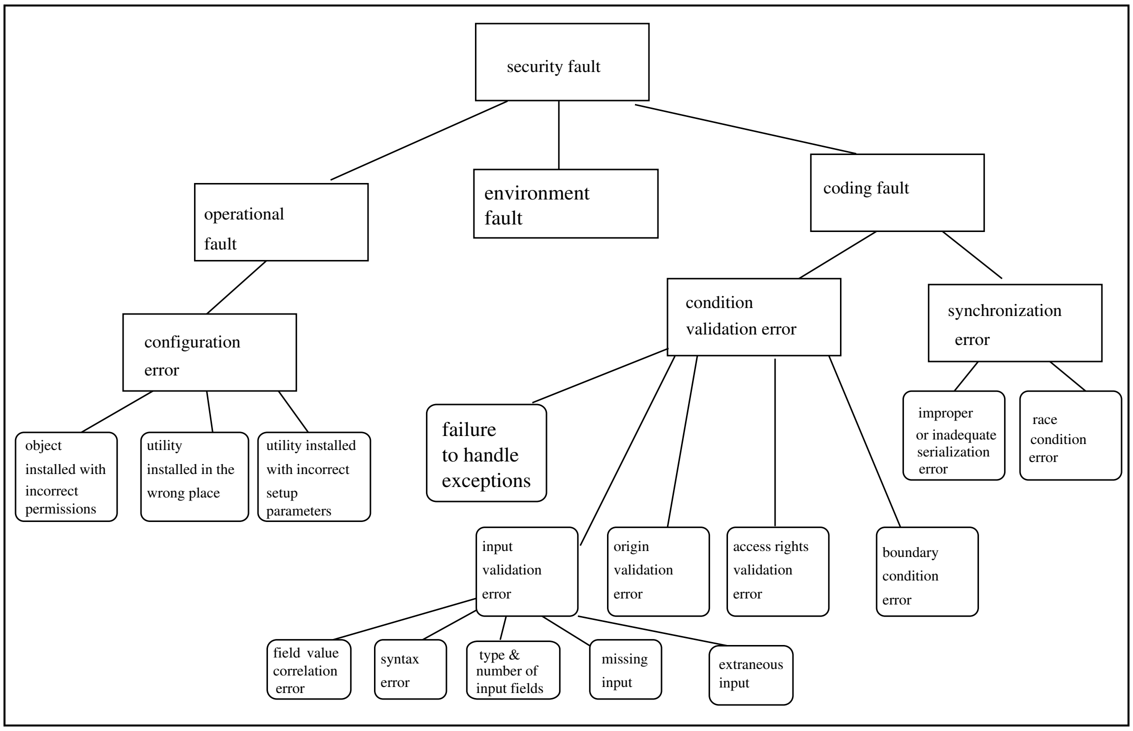

Figure 2.1 Taxonomy of Faults

2.2 Configuration Errors

The configuration of a system can be broadly defined as consisting of the hardware and software resources [web88]. Thus, the system utility programs, and various service routines that a computer system provides can be regarded as being part of the configuration. In our taxonomy, a fault can be classified as a configuration if:

- a program/utility is installed in the wrong place
- a program/utility is installed with incorrect setup parameters
- a secondary storage object or a program/utility is installed with incorrect permission that violates the security policy

The following examples illustrate how configuration errors have been exploited to compromise security.

2.2.1 Examples

The trivial file transfer protocol (`tftp`) is a network utility based on the User Datagram Protocol (UDP), and is commonly used by diskless workstations to download kernel image files from a server [Com88]. At some sites running different versions of Unix, the `tftp` daemon was enabled in such a way that it allowed any user on the Internet to access any file on the machine running `tftp` [CA-91a, CA-91b]. This could result in disclosure of information and security could be compromise if sensitive files were read. For instance, if an intruder obtained access to the password file it could be used to crack passwords and gain unauthorized access to the system. The problem occurred because `tftp` was not properly installed. If `tftp` had to be enabled, it should have been installed such that it only had restricted access to the file system through the `chroot` command.

In Unix BSD 4.2, `sendmail` was installed with the "Wizard mode" enabled [CA-90a]. The "Wizard mode" option was left over from a test scaffolding. It was included to enable system programmers to debug the program without having to go through a lengthy authentication process. If `sendmail` was installed with "Wizard mode" enabled, any user could gain access to the system by connecting to the mail port and providing `WIZ` as a command. The "Wizard mode" came with a default password of `wizzywoz`. By using this default password, an intruder could bypass the system authentication procedure and gain access to the system.

2.3 Synchronization Errors

Carlstead [Car78] studied serialization errors in detail as part of the Protection Analysis Project. Some of the ideas presented in this section are borrowed from his results. Two operations are data related if the evaluation of conditions by one can affect the other by changing the relative ordering of the operations. Two operations are control related if a change in the relative order of execution can produce different results. Serialization is concerned with two types of relationships between operations: data flow and control flow. The purpose of serialization is to maintain the appropriateness of the latter with the former [Car78].

In our taxonomy, a fault can be classified as a synchronization error if:

- a fault can be exploited because of a timing window between two operations
- a fault results from improper serialization of operations

Synchronization errors in our taxonomy include both race condition errors (errors that occur because of a timing window) and serialization errors (error that result from improper serialization of operations).

2.3.1 Example

A Unix specific example is presented here to illustrate race condition errors. `xterm` provides a window interface in the X window system. A vulnerability was found in many versions of the `xterm` program, which if exploited allowed users to create and delete arbitrary files in the system. If `xterm` operated as a setuid or setgid process, then a race condition between the access check permissions to the logging file, and the logging itself, allowed users to replace any arbitrary file with the logging file [CA-93a]. The following code explains how the vulnerability could be exploited.

```
mknod foo p              # create a FIFO file and name it foo
xterm -lf foo            # start logging to foo
mv foo junk              # rename file foo to junk
ln -s /etc/passwd foo    # create a symbolic link to password file
cat junk                 # open other end of FIFO
```

This error occurs because of a timing window that exists between the time access permissions of the logging file are checked and the time actual logging is started. This timing window could be exploited by creating a symbolic link from the logging file to a target file in the system. If `xterm` ran as setuid root, this could be used to create new files or destroy existing files in the system.

2.4 Condition Validation Errors

Most operations in an operating system can be classified as [Car78]:

prohibitive: an operation is allowed to proceed only if the conditions under which it must be allowed to proceed hold, otherwise the action is aborted.

or

inhibitive: an operation is delayed until the conditions under which it can proceed are met.

For example, access checking mechanisms are prohibitive because an action is denied if a user/process invokes it on an object outside its access domain. Synchronization mechanisms are inhibitive because an operation is delayed until the required condition is satisfied.

Conditions are usually specified as a conditional construct in the implementation language. An expression corresponding to the condition is evaluated and an execution path is chosen based on the outcome of the condition. In this discussion, we assume that an operation is allowed to proceed only if the condition evaluated to true. A condition validation error occurs if:

- Condition is missing: This allows an operation to proceed regardless of the outcome of the condition expression.
- Condition is incorrectly specified: If a condition is incorrectly specified it would allow execution to proceed along an alternate path. This may allow an operation to proceed regardless of the outcome of the condition expression and completely invalidate the check.
- A predicate in the condition expression is missing: This would evaluate the condition incorrectly and allows an alternate execution path to be chosen.

Condition errors are coding faults that occur because a programmer misunderstood the requirements, or made a logic error when a condition was specified.

In our taxonomy, a fault is classified as a condition error if one of the following conditions is missing or not specified correctly.

Check for limits: Before an operation can proceed, the system must ensure that it can allocate the required resources without causing starvation or deadlocks. For input/output operations, the system must also ensure that a user/process does not read or write beyond its address boundaries.

Check for access rights: The system must ensure that a user/process can only access an object in its access domain. The mechanics of this check depend on how access control mechanisms are implemented.

Check for valid input: Any routines that accept input directly from a user or from another routine must check for its validity. This includes checks for:

- Field-value correlation

- Syntax
- Type and number of parameters or input fields,
- Missing input fields or delimiters
- Extraneous input fields or parameters

Failure to properly validate input may indirectly cause other functional modules to fail and cause the system to behave in an unexpected manner.

Check for the origin of a subject: In this context, subject refers to a user/process, host, and shared data objects. The system must authenticate the subject to prevent identity compromise attacks.

Check for exceptional conditions: The system must be able to handle exceptional conditions that arise from failure events or malfunction of a functional module or device.

2.4.1 Examples

In Unix `/etc/exports` specifies a lists of trusted remote hosts that are allowed to mount the file system. In SunOS 4.1.x, if a host entry in the file was longer than 256 characters, or if the number of hosts exceeded the cache capacity, a buffer overflow allowed any non-trusted host to mount the file system [CA-94]. This allowed unauthorized users read and write access to all files on a system. This error occurred because the system failed to check that it had read more than 256 characters or that it had exhausted the cache capacity.

The Unix operating system allows users to copy a file from or to a remote machine via the `rcp` utility. A vulnerability was discovered in a version of Unix that allowed remote users of a trusted machine to execute root-privilege commands via `rcp` [CA-89]. This problem occurred because of an authentication error in the name lookup protocol that binds the high level addresses used for identification to Internet Protocol (IP) addresses. This error is representative of a number of other similar flaws that can be exploited using a weakness in the Domain Name Service protocol [Sch93]. The error occurred because the origin of the message (hostname) was not authenticated.

In SunOS 4.1 and SunOS 4.1.1, any user could redirect characters away from any other user's terminal device. The error occurred because the input/output routine failed to properly check access rights of the user who had invoked the operation. The fix involved adding a check to allow a read or write request to proceed only if the user had the correct access rights and reject the request otherwise [CA-90b].

`uux` is a Unix utility that allows users to remotely execute a limited set of commands. A flaw in the parsing of the command line allowed remote users to execute arbitrary commands on the system [Bis86]. The command line to be executed was received by the remote system, and parsed to see if the commands in the line were among the set of commands that could be executed. `uux` read the first word of the line, and skipped characters until a delimiter character (;,^, `|`) was read. `uux` would continue this way until the end of the line was read. However, two delimiters (&, ‘) were missing from the set, so a command following these characters would never be checked before being executed. For example, a user could execute any command by executing the following sequence.

```
uux remote_machine ! rmail anything & command
```

In `uux` the command after & would not be checked before being executed. This allowed users to execute unauthorized commands on a remote system. This error occurred because `uux` failed to check for missing delimiters.

`fsck` is a program that checks and repairs file system consistency. Some of the inconsistencies that `fsck` corrects include: too-large link counts in inodes, missing blocks in the free list, blocks appearing in the free list and also in files, or incorrect counts in the super block. On many Unix systems, `fsck` is run as part of the system bootup procedure to correct filesystem inconsistencies. If `fsck` failed during bootup in Solaris 2.x, a privileged shell was started on the console. This enabled any user with physical access to gain root privileges [CA-93b].

2.5 Environment Faults

Environment faults are introduced when specifications are translated to code but sufficient attention is not paid to the run-time environment [Spa90]. These faults are dependent on the operational environment and result when software is executed in a different environment than the programmer had in mind.

Some typical examples of environmental faults include [Spa90]:

- the value used to initialize "uninitialized" pages or segments of virtual memory may introduce unsuspected problems
- interpertation of supposed constant values may produce faults, such as porting code that deferences a constant pointer
- exception handling and reporting may be different than expected
- system errors: a compiler may produce incorrect code, the hardware may execute instructions differently than expected under some circumstance, and the operating system may introduce intermittent errors not related to the user code

Environment faults can also occur when different modules interact in an unanticipated manner. Independently the modules may function according to specifications but an error occurs when they are subjected to a specific set of input in a particular execution environment.

2.5.1 Example

The `exec` system call overlays a new process image over an old one. The new image is constructed from an executable object file or a data file containing commands for an interpreter. When an interpreter file is exec'd, the arguments specified in the `exec` call are passed to the interpreter. Most interpreters take "-i" as an argument to start an interactive shell.

In SunOS version 3.2 and earlier, any user could create an interactive shell by creating a link with the name "-i" to a setuid shell script. `exec` passed "-i" as an argument to the shell interpreter that started an interactive shell. Both the `exec` system call and the shell interpreter worked according to specifications. The error resulted from an interaction between the shell interpreter and the `exec` call that had not been considered.

2.6 Selection Criteria

In this section we describe the selection criteria that can be used to decide membership of security faults into different categories. The motivation is to avoid any ambiguities and distinctly classify each fault.

For each fault category, we present a series of questions that are used to determine membership in a specific category. An answer in the affirmative to a question in that series qualifies the fault to be classified in the corresponding category. We assume that sufficient details about the faults are known.

Condition Validation Errors

The following sets of questions can be used to determine if a fault can be classified as condition validation error.

Boundary Condition Errors

- Did the error occur when a process attempted to read or write beyond a valid address boundary?
- Did the error occur when a system resource was exhausted?
- Did the error result from an overflow of a static-sized data structure?

Access Validation Errors

- Did the error occur when a subject invoked an operation on an object outside its access domain?

- Did the error occur as a result of reading or writing to/from a file or device outside a subject's access domain?

Origin Validation Errors

- Did the error result when an object accepted input from an unauthorized subject?
- Did the error result because the system failed to properly or completely authenticate a subject?

Input Validation Errors

- Did the error occur because a program failed to parse syntactically incorrect input?
- Did the error result when a module accepted extraneous input fields?
- Did the error result when a module did not handle missing input fields?
- Did the error result because of a field-value correlation error?

Failure to Handle Exceptional Conditions

- Did the error because the system failed to handle an exceptional condition, generated by a functional module, device, or user input?

Synchronization Errors

This section presents the criteria that can be used to decide if a fault can be classified as a synchronization error.

Race Condition Errors

- Is the error exploited during a timing window between two operations?

Serialization Errors

- Did the error result from inadequate or improper serialization of operations?

Environment Errors

This section presents a series of questions that be used to decide if a fault can can be classified as an environment error.

- Does the error result from an interaction in a specific environment between functionally correct modules?
- Does the error occur only when a program is executed on a specific machine, under a particular configuration?
- Does the error occur because the operational environment is different from that the software was designed for?

Configuration Errors

The following questions can be used to determine if a fault can be classified as a configuration error.

- Did the error result because a system utility was installed with incorrect setup parameters?
- Did the error occur by exploiting a system utility that was installed in the wrong place?
- Did the error occur because access permissions were incorrectly set on a utility such that it violated the security policy?

Figure 2.2 shows a pictorial representation of the decision tree derived from questions used for deciding membership for faults. The internal nodes of the decision tree represent the series of questions that are used for each category. The leaf nodes in the tree represent the different fault categories. The decision tree shows two nodes that are labeled *unclassifiable errors*. These nodes represent a *catch-all* category for the coding, operational, and environment faults. The *unclassifiable error* categories are included in our classification scheme to account for new faults that cannot be readily classified using our existing criteria. The new fault category and the associated criteria can be added as a parent node of the *unclassifiable error* node. The ability to add new fault categories can be useful if the taxonomy is extended to different operating systems that have different structures.

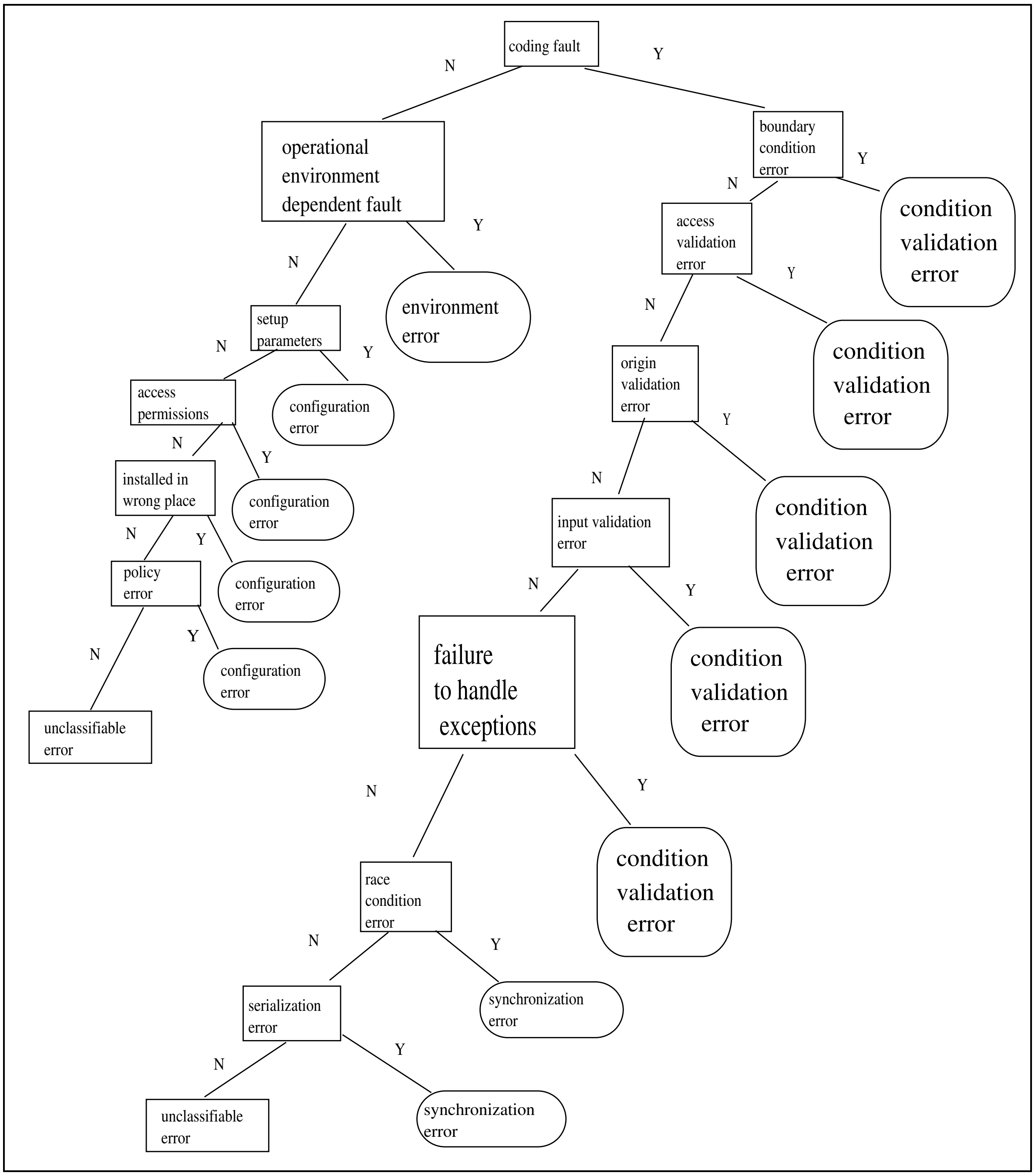

Figure 2.2 Selection Criteria for Fault Classification

3. DESIGN OF THE VULNERABILITY DATABASE

In this chapter we present the conceptual design of a vulnerability database. The vulnerability database can be used to organize information about security faults based on our taxonomy. This chapter discusses the various design issues and the rational behind the design choices. The design of the database is presented in terms of relational model entities and relational algebra functions. This approach is used to separate the implementation details from the abstract design.

3.1 Motivation

Landwehr et al.[L+93] observe that the history of software failure has been mostly undocumented and knowing how systems have failed can help us design better systems that are less prone to failure. The design of this vulnerability database is one step in that direction. The database can serve as a repository of vulnerability information collected from different sources. This information can be organized to allow useful queries to be performed on the data. The information in the database can be useful to the system designer in identifying areas of weaknesses in the design, requirements, or implementation. Personnel involved with testing security mechanisms of an operating system may also find the information to be useful. The database also demonstrates that our taxonomy can be applied to classify actual vulnerabilities from different versions of Unix.

Our database can also be used to maintain vendor patch information, vendor and response team advisories, and catalog the patches applied in response to those advisories. This information can be helpful to system administrators maintaining a legacy system.

3.2 Entity Relation Model

As part of the conceptual design process, an entity relation model is often used to illustrate the relationships between the different entities in the database model. An entity is a basic object in the entity relation model with an independent existence in the real world [EN89]. An entity can either have a physical or a conceptual existence.

Based on the queries collected in the requirements analysis phase, the following three entities were identified.

Vulnerability: The vulnerability entity type corresponds to the description of a potential security flaw in the operating system.

Operating system: The operating system entity type corresponds to the operating system that contains a particular vulnerability.

Patch Info: Patch info is a conceptual entity that provides information about patches that were applied to a system and the vulnerabilities they fix.

3.3 Entity Attributes

An attribute is a particular property possessed by an entity. Associated with an attribute is a value, which is the information stored in that attribute. Attributes can be either single valued or multi-valued. The single valued attributes are associated with the value set, which specifies the domain of values that may be assigned to the attribute.

The attributes associated with the three entity types are outlined below. We have also included a brief description, and the value set of the attribute.

Attribute	Description	Value Set
Name/identification number	Unique identification (primary key)	string
Source	Where the vulnerability was reported	string
Description	A brief description of how it was or could have been exploited	text
Detection technique	A brief description of the testing strategy that can be used to detect this vulnerability	text
Classification	Representative category in the taxonomy	string
Work around	Any known work-arounds	text
Description of fix	A description of how the vulnerability was fixed	text
Literature	Any reference to the vulnerability	text
Consequence	A generic consequence class	string

Table 3.1 Vulnerability Entity Attributes

Attribute	Description	Value Set
System name	Unique name of the operating system (primary key)	string
Family of system	Family of the system e.g. Unix	string
Vendor	Vendor's name and address if available	string

Table 3.2 Operating System Entity Attributes

Attribute	Description	Value Set
Patch number	Unique patch number (primary key)	string
Description	A brief description of the patch	string
Fixes vulnerability	Vulnerability/vulnerabilities fixed by this patch	text

Table 3.3 Patch Info Attributes

3.3.1 Entity Relation Diagram

The entity-relation diagram (Figure 3.1) shows two M:N relationships that exist between the entities in the database. These relationships cannot be easily represented in the hierarchical or the network model without replication of data records. As the amount of stored information in the vulnerability database can be large, we opted for the relational model minimize data replication. Each M:N relationship is represented by an additional relation that contains the primary keys of the entities on both sides.

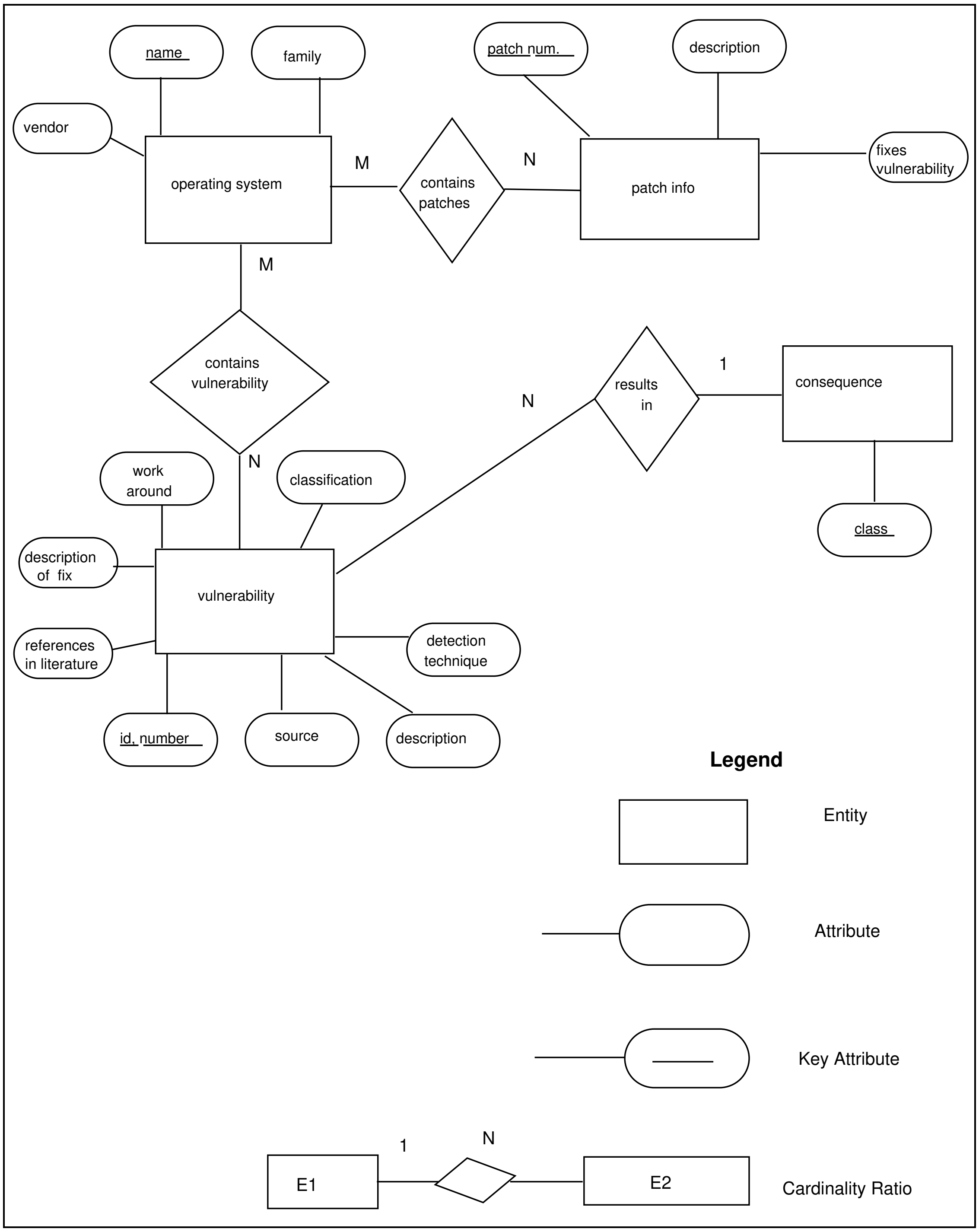

Figure 3.1 Entity Relation Diagram

The following schema are derived from the entity-relation diagram and show the different attributes. The primary key attributes of the schema are underlined.

Operating System Schema

name	family	vendor

Vulnerability Schema

vulner id	source	description	detection technique	classification	work around	description of fix	references in literature	consequence

Patch Info Schema

patch number	description	fixes vulnerability

Contains_patch Schema

operating system name	patch number

Contains_vulnerability Schema

operating system name	vulnerability id

3.3.2 Entity Integrity Constraints

The entity integrity constraints state that primary keys used to uniquely identify a tuple in a relation should not be null. Entity integrity constraints should be enforced on all update, add, and modify operations.

3.3.3 Referential Integrity Constraints

Referential integrity constraints are specified between two relations and are used to maintain consistency between the tuples. Referential integrity constraints should be maintained on add, delete, and modify operations. The referential integrity constraints in our vulnerability database are diagrammatically depicted in Figure 3.2.

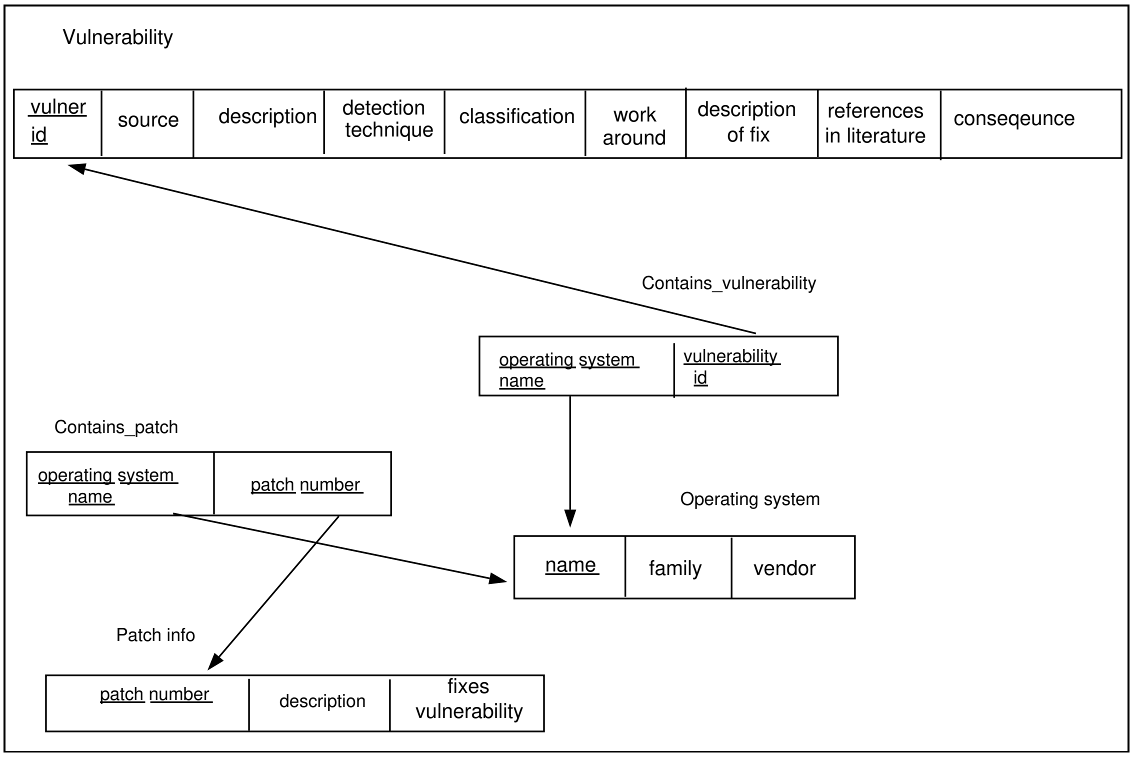

Figure 3.2 Referential Integrity Constraints

3.4 Operation on the Relations

Relational Algebra is a collection of operations that are used to manipulate entire relations. It includes operations to select tuples from individual relations, and to combine related tuples from several relations for the purpose of specifying a query. The complete set of relational operations consists of SELECT, PROJECT, UNION, DIFFERENCE, CARTESIAN PRODUCT. Any other relational operation can be performed as a combination of these five operations[EN89]. These operations are briefly described in Table 3.4.

In the following sections we present the implementation details of the vulnerability database prototype.

The prototype was developed to demonstrate that vulnerabilities from different versions of Unix can be classified in our taxonomy of faults.

The three major components of the vulnerability database are:

- Database kernel
- Vulnerability database interface
- User-interface

Each of the three components are discussed in detail in the following sections. The information and data flow information between the three components is shown in Figure 3.7.

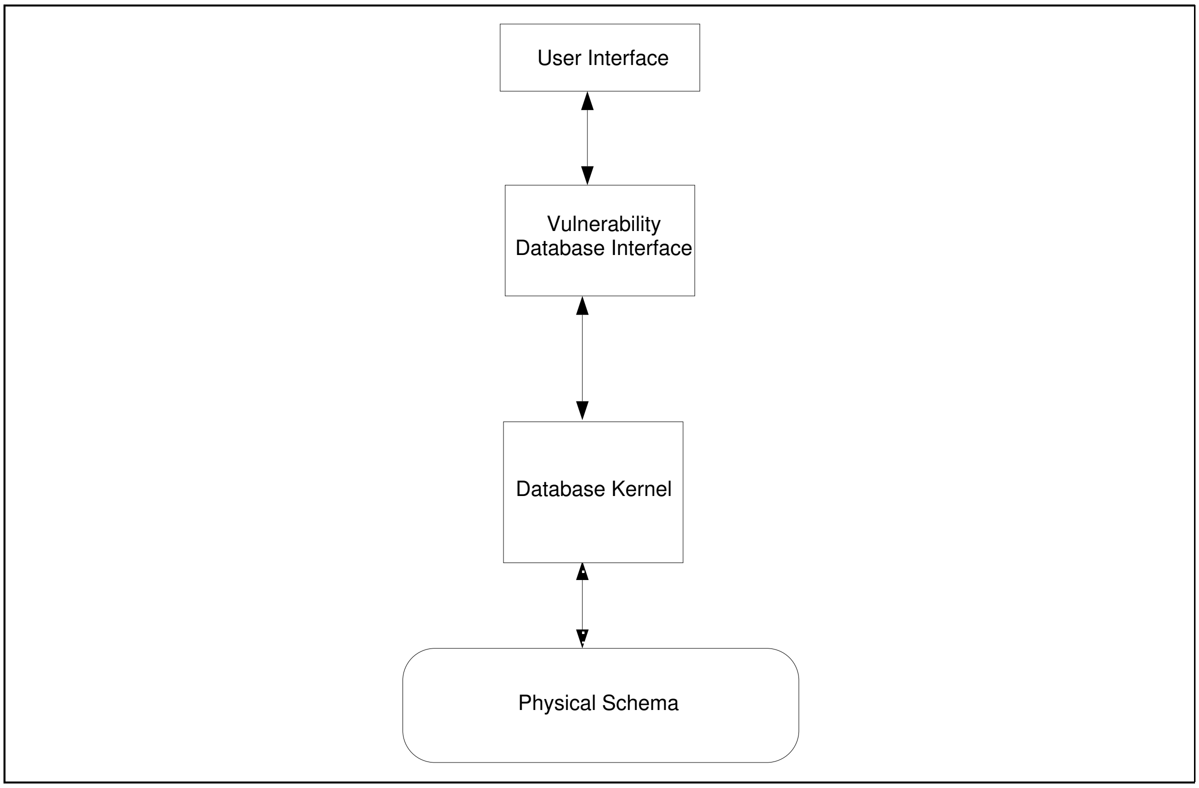

Figure 3.3 Vulnerability Database

3.5 Requirements Analysis

As part of the requirements analysis, we identified two major uses for the vulnerability database. It can be used to maintain up-to-date security related information about different operating systems that can be helpful to system designers and researchers. Also, the vulnerability information can be used during security assessment studies of operating systems. In the second phase of requirements analysis, we asked several people involved with security-related projects to comment on a small set of queries we had proposed. We asked them to supplement the list with any additional queries that may be performed on the database. We then constructed a small but comprehensive set of queries that encompass a wide range of attributes. The collection of queries helped us identify the information that should be included in different attributes of the data objects.

Although the collection of vulnerabilities provided useful information, some of the queries were discarded and the attributes they accessed are not included in the database. For example, a query about the time that a vulnerability was introduced into the system cannot be answered by using the stored information. Similarly, queries about programming practices are not supported. Such queries cannot be answered without access to program source code, program requirements, functional specifications and knowledge of the program development environment. Furthermore, such queries are difficult to answer because they require a qualitative evaluation of information.

The set of queries that can be supported in the initial prototype of the database is shown below.

1. List all the vulnerabilities in system X.
2. List all vulnerabilities in system X that occur because of Y (e.g. race condition)
3. List all vulnerabilities which result in Z (e.g. denial of service)
4. Perform any combination of boolean operations on 3 and 2. e.g.: $Y \wedge Z$, $Y \vee Z$, $Y \wedge (\neg Z)$
5. Given vulnerability V, classify it.
6. Given vulnerability V, find all systems that have this vulnerability.
7. List all vulnerabilities in system X that will result in Z.
8. List all systems that have a vulnerability that will cause Z.
9. Given vulnerability V and system X, what were (or could have been) the consequence if exploited.
10. Give a brief description of how V was (could have been) exploited.
11. List any reference to vulnerability V in the literature.
12. Give a description of the fix for vulnerability V.
13. List all patches applied to system X.
14. For a vulnerability V, are there any practical work-arounds?
15. For system X, version Y, what vulnerabilities have no supported patch?
16. For all systems that have a vulnerability V, list their version numbers, and vendors.
17. List any references to V in the literature.

Some queries collected during requirements analysis are not supported because they require a qualitative analysis of the data. For completeness, these queries are also shown below.

1. Given vulnerability V and system X, which part of the operating system was it found in?

2. Does patch P completely fix a given vulnerability?

3. What is the level of difficulty/expertise required to patch vulnerability V?

4. What is the initial access/setup required to exploit vulnerability V (e.g., requires a user account, requires access to port P)?

5. Given a coding practice, list all vulnerabilities resulting from this coding practice?

6. Given V, how could have it been prevented?

7. For a class of vulnerability C, which software engineering practice P would have prevented the introduction of this vulnerability?

3.6 Design Issues

The two major factors that influenced the design of our database were scalability and ease of maintaining the integrity of information.

A poor design that did not scale could lead to degradation of performance, waste storage space and create problems in maintenance. Even though performance was not a critical factor in the design, we keep a response time for the queries. We paid particular attention to ensure that little or no data replication took place. Our initial design was based on a hierarchical model, but the M:N relationships between the different entities could not be modeled without data replication. At that point, a relational schema was adopted that facilitated the representation of the M:N relationships with less data replication.

Maintenance of a large repository of information is a difficult task and it is not clear if any particular organization scheme can make it any easier. We have identified entity integrity constraints as well as referential integrity constraints that should be helpful to maintain the integrity of the stored information.

3.7 Database Kernel

Our vulnerability database is based on a relational schema model that consists of both physical and conceptual entities. These entities are represented as relations (tables) in the model. Relational algebra defines the operations that can be performed on the the relations. It also defines a set of basis functions such that any query in the relational model can be specified only in terms of these functions. The basis functions in the relational model are: SELECT, PROJECT, UNION, DIFFERENCE, and CARTESIAN PRODUCT.

The vulnerability database kernel is comprised of the five basis functions that directly operate on the physical schema. In addition, it also contains the following functions:

Create: Creates a new relation in the database.

Modify: Modify all the fields of a tuple in a relation.

Update: Change a specified field of a tuple in a relation.

Delete: Delete a tuple from a relation.

Destroy: Remove a relation from the database.

The kernel also provides a layer of abstraction to the upper level layers, which do not directly interface with the physical schema. This enables us to restrict details of the physical schema to the kernel functions.

3.7.1 Representation of Relations

In the vulnerability database, each relation is contained in a separate file. Each tuple in the relation is represented as a line of text, where the different fields are separated by a colon. An instance of the operating system relation is shown below.

```
os1:un*x:some vendor
os2:x*n:vendor 2
```

3.8 Vulnerability Database Interface

The database provides an interface to the physical schema through a set of kernel functions. All of the queries listed in Section 3.13 can be specified using only the kernel functions. However, this approach has two disadvantages. First, the underlying details of the physical schema are exposed to the user. Second, specifying the queries only in terms of the kernel functions becomes cumbersome and exposes details of the physical schema. As an alternative, a separate layer of abstraction called the *vulnerability database interface* is provided. This layer provides a set of form-based queries that interact with the kernel functions. The set of queries currently supported in the vulnerability database layer of the prototype are listed in Table 3.6.

Query	**Interface Function**
List all the vulnerabilities in system X.	list_vulnerability $< system >$
List all vulnerabilities in system X that occur because of Y.	vulner_conseq $< vulner >< system >$
List all vulnerabilities which result in Z.	list_conseq $< conseq >$
Given vulnerability V, classify it.	classify $< vulner >$
Given vulnerability V, find all systems that have this vulnerability.	find_system $< vulner >$
List all vulnerabilities in system X that will result in Y.	vulner_result $< system >< conseq >$
List all systems which have a vulnerability that will cause Y.	list_systems $< systems >$
Given vulnerability V and system X, what were (or could have been) the consequence if exploited.	conseq $< vulner >< system >$
Give a brief description of how V was (could have been) exploited.	exploit $< vulner >$
List any reference to V in the literature.	descr_fix $< vulner >$
Give a description of the fix .	literature $< vulner >$
List all patches applied to system X.	patches_to $< system >$
For a vulnerability V, are there any practical work-arounds?	work-around $< vulner >$
For system X, version Y, what vulnerabilities have no supported patch?	nopatch $< system >$
For all systems that have a vulnerability V, list their vendors.	sysinfo $< vulner >$

Table 3.5 Vulnerability Database Interface Functions

3.9 User Interface

A user can interact with our database either by using command line arguments or through a graphical user interface. The graphical interface was written using the Tk toolkit [Ous94] and provides a flexible interface to the vulnerability database functions. All of the functions listed in Table 3.6 can be invoked through the interface.

Figure 3.8 shows the main commands screen that shows all the functions that can be invoked. After a function has been selected, it may be invoked using the run button. This presents another window with entry fields for the needed arguments. Figure 3.9 shows the dialogue window for the `vulner_conseq` function. It shows the entry fields for the two needed arguments. A template representation for the input fields was chosen so that a user would not need to remember the order or number of arguments required by each function. A user can also obtain a help window for a selected function by using the help command in the main screen. The help screen contains a brief description of the function, the arguments required and the usage syntax. Figure 3.10 shows a help screen for the `vulner_conseq` function.

Operation	Description	Notation
SELECT	Select all tuples that satisfy the selection condition from a relation R.	SELECT < selection condition > R
PROJECT	Produce a new relation with only some of the attribute of R and remove duplicate tuples	PROJECT <attribute list > R
UNION	Produce a new relation that includes all tuples in R_1 or R_2 or both R_1 and R_2; R_1 and R_2 must be union compatible	R_1 UNION R_2
DIFFERENCE	Produce a relation that includes all the tuples in R_1 that are not in R_2	R_1 DIFFERENCE R_2
CARTESIAN PRODUCT	Produce a relation that has the attributes of R_1 and R_2 and includes as tuples all possible combinations of tuples from R_1 and R_2	R_1 CARTESIAN PRODUCT R_2

Table 3.4 Operations on Relations

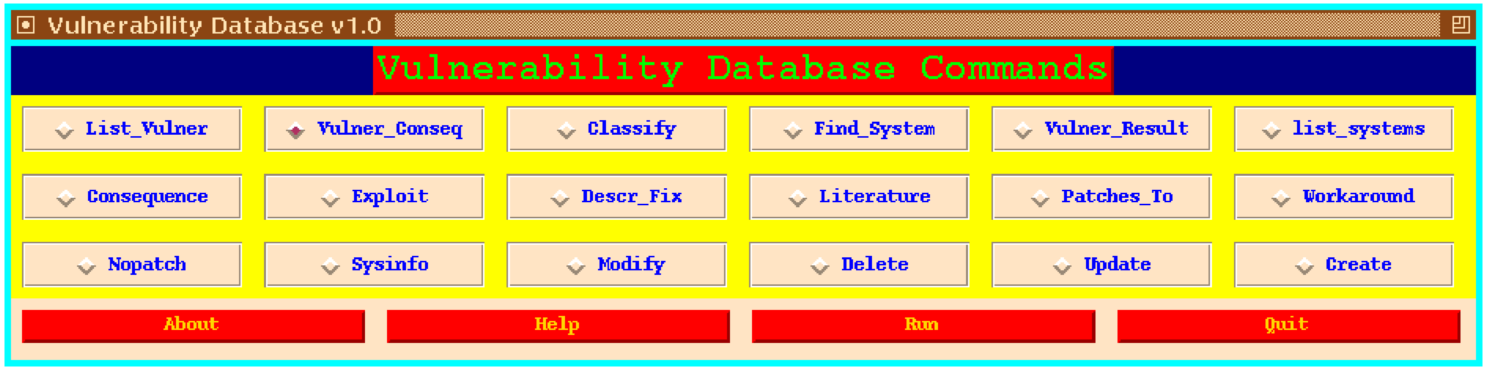

Figure 3.4 Vulnerability Database Commands Screen

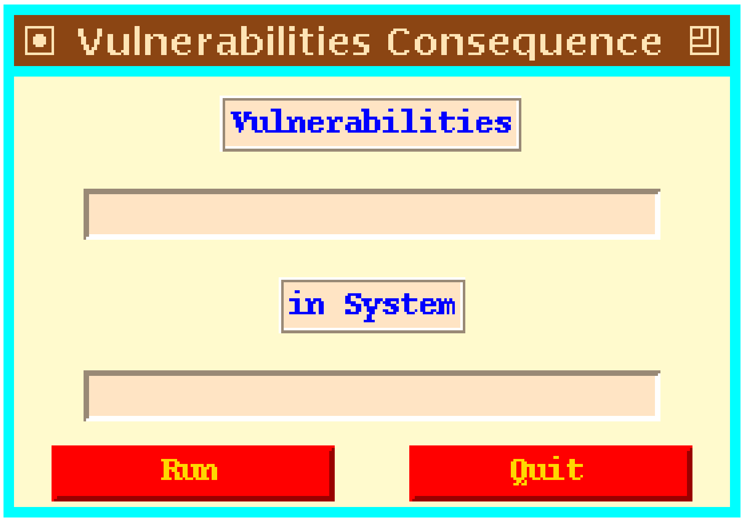

Figure 3.5 Function Interface

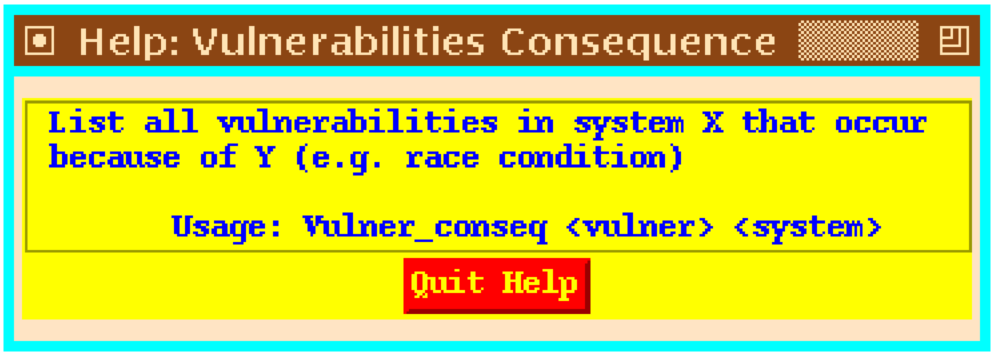

Figure 3.6 Help Window

3.10 Integrity Constraints

In a relational database, entity and referential integrity constraints must be maintained when creating, modifying or updating relations to maintain the integrity of stored information. The referential and entity integrity constraints in the vulnerability database were discussed in section 3.3.2 and 3.3.3. This section describes how these constraints are implemented in the database.

3.10.1 Entity Integrity Constraints

Entity integrity constraints specify that the primary key of a relation, which is used to uniquely identify a tuple, is always defined. In the vulnerability database, entity integrity constraints are explicitly specified in the "integrity.conf" file. The format of entries in the files is as shown below:

```
vulner:vulner_id
contains_vulner:os:vulner_id
os:name
contains_patch:os:patch_number
patch_info:patch_number
```

The name of the relation is followed by the fields that serve as the primary key(s). This format allows more than one key to be specified in the integrity check and allows the maintainer to change the keys if desired. The entity integrity check can be invoked every time a relation is created, modified, or when a tuple is modified or deleted from the relation.

The primary keys that are used in the database are:

vulnerability number: For vulnerabilities obtained from a response team, the primary key is the same as the unique identifier used by a response team to advertise the vulnerability. For other vulnerabilities a unique number is assigned to each vulnerability tuple.

operating system name: The primary key for the operating system entity is formed by concatenating the name of the operating system to its version number. This field serves as a unique identifier and also implicitly stores the version number.

patch number: Patch number is the same unique key used by vendors to identify a distributed patch.

3.10.2 Referential Integrity Constraints

Referential integrity constraints are specified between tuples of two relations to maintain consistency among the tuples. In the vulnerability database, referential integrity constraints are explicitly specified in the "chk.conf" file. The tuples that have referential integrity constraints are specified as:

```
relation1:fieldA ==> relation2:fieldB
```

In the instance shown, `fieldA` in `relation1` specifies a referential integrity constraint on `fieldB` in `relation2`. Any changes to `fieldB` in `relation2` should be reflected in `fieldA` in `relation1`. Referential integrity checks should be invoked when tuples in the relations are modified, deleted, or updated.

3.11 Template Representation

The vulnerability database provides flexibility to change fields in the tuples contained in the relations, or to change the integrity constraints through an editable template representation. The different fields of the tuples are specified in the "db.conf" file using a format shown below.

```
#vulnerability#: id: source: classification
```

The example above shows a relation "vulnerability" that consists of three fields: "id", "source", and "classification".

The templates in "db.conf" are used by the database kernel routines when new tuples are added, modified, or updated. A template representation allows the maintainer of the database to easily add or delete fields from the tuples. The changes are then reflected in any subsequent CREATE, MODIFY, or UPDATE operations.

Integrity constraints in the vulnerability database are also specified as editable templates. This also allows maintainers to change primary keys, add new keys, or change referential integrity constraints by simply editing the templates in the integrity configuration files.

3.12 Data Sources

We have started populating the vulnerability database with attacks that led to a security compromise. The information was gathered from advisories issued by the Computer Emergency Response Team (CERT), information reported on an electronic security mailing list, and a survey of the literature. At present the database contains information on fourty nine known vulnerabilities. These vulnerabilities have been categorized based on our taxonomy, and include descriptions of how each was exploited, and the operating systems that it was found in.

In the following sections we present the implementation details of the vulnerability database prototype.

The prototype was developed to demonstrate that vulnerabilities from different versions of Unix can be classified in our taxonomy of faults.

The three major components of the vulnerability database are:

- Database kernel
- Vulnerability database interface
- User-interface

Each of the three components are discussed in detail in the following sections. The information and data flow information between the three components is shown in Figure 3.7.

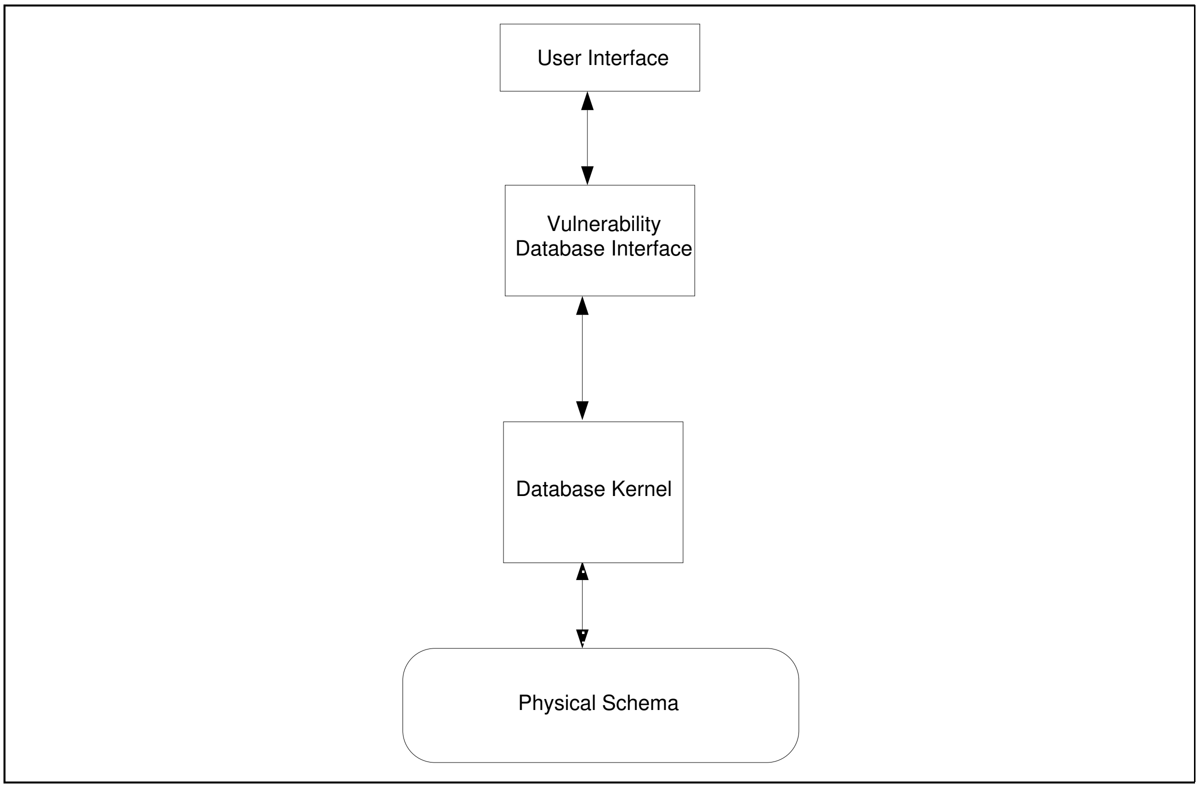

Figure 3.7 Vulnerability Database

3.13 Requirements Analysis

As part of the requirements analysis, we identified two major uses for the vulnerability database. It can be used to maintain up-to-date security related information about different operating systems that can be helpful to system designers and researchers. Also, the vulnerability information can be used during security assessment studies of operating systems. In the second phase of requirements analysis, we asked several people involved with security-related projects to comment on a small set of queries we had proposed. We asked them to supplement the list with any additional queries that may be performed on the database. We then constructed a small but comprehensive set of queries that encompass a wide range of attributes. The collection of queries helped us identify the information that should be included in different attributes of the data objects.

Although the collection of vulnerabilities provided useful information, some of the queries were discarded and the attributes they accessed are not included in the database. For example, a query about the time that a vulnerability was introduced into the system cannot be answered by using the stored information. Similarly, queries about programming practices are not supported. Such queries cannot be answered without access to program source code, program requirements, functional specifications and knowledge of the program development environment. Furthermore, such queries are difficult to answer because they require a qualitative evaluation of information.

The set of queries that can be supported in the initial prototype of the database is shown below.

1. List all the vulnerabilities in system X.
2. List all vulnerabilities in system X that occur because of Y (e.g. race condition)
3. List all vulnerabilities which result in Z (e.g. denial of service)
4. Perform any combination of boolean operations on 3 and 2. e.g.: $Y \wedge Z$, $Y \vee Z$, $Y \wedge (\neg Z)$
5. Given vulnerability V, classify it.
6. Given vulnerability V, find all systems that have this vulnerability.
7. List all vulnerabilities in system X that will result in Z.
8. List all systems that have a vulnerability that will cause Z.
9. Given vulnerability V and system X, what were (or could have been) the consequence if exploited.
10. Give a brief description of how V was (could have been) exploited.
11. List any reference to vulnerability V in the literature.
12. Give a description of the fix for vulnerability V.
13. List all patches applied to system X.
14. For a vulnerability V, are there any practical work-arounds?
15. For system X, version Y, what vulnerabilities have no supported patch?
16. For all systems that have a vulnerability V, list their version numbers, and vendors.
17. List any references to V in the literature.

Some queries collected during requirements analysis are not supported because they require a qualitative analysis of the data. For completeness, these queries are also shown below.

1. Given vulnerability V and system X, which part of the operating system was it found in?
2. Does patch P completely fix a given vulnerability?
3. What is the level of difficulty/expertise required to patch vulnerability V?
4. What is the initial access/setup required to exploit vulnerability V (e.g., requires a user account, requires access to port P)?
5. Given a coding practice, list all vulnerabilities resulting from this coding practice?
6. Given V, how could have it been prevented?
7. For a class of vulnerability C, which software engineering practice P would have prevented the introduction of this vulnerability?

3.14 Design Issues

The two major factors that influenced the design of our database were scalability and ease of maintaining the integrity of information.

A poor design that did not scale could lead to degradation of performance, waste storage space and create problems in maintenance. Even though performance was not a critical factor in the design, we keep a response time for the queries. We paid particular attention to ensure that little or no data replication took place. Our initial design was based on a hierarchical model, but the M:N relationships between the different entities could not be modeled without data replication. At that point, a relational schema was adopted that facilitated the representation of the M:N relationships with less data replication.

Maintenance of a large repository of information is a difficult task and it is not clear if any particular organization scheme can make it any easier. We have identified entity integrity constraints as well as referential integrity constraints that should be helpful to maintain the integrity of the stored information.

3.15 Database Kernel

Our vulnerability database is based on a relational schema model that consists of both physical and conceptual entities. These entities are represented as relations (tables) in the model. Relational algebra defines the operations that can be performed on the the relations. It also defines a set of basis functions such that any query in the relational model can be specified only in terms of these functions. The basis functions in the relational model are: SELECT, PROJECT, UNION, DIFFERENCE, and CARTESIAN PRODUCT.

The vulnerability database kernel is comprised of the five basis functions that directly operate on the physical schema. In addition, it also contains the following functions:

Create: Creates a new relation in the database.

Modify: Modify all the fields of a tuple in a relation.

Update: Change a specified field of a tuple in a relation.

Delete: Delete a tuple from a relation.

Destroy: Remove a relation from the database.

The kernel also provides a layer of abstraction to the upper level layers, which do not directly interface with the physical schema. This enables us to restrict details of the physical schema to the kernel functions.

3.15.1 Representation of Relations

In the vulnerability database, each relation is contained in a separate file. Each tuple in the relation is represented as a line of text, where the different fields are separated by a colon. An instance of the operating system relation is shown below.

```
os1:un*x:some vendor
os2:x*n:vendor 2
```

3.16 Vulnerability Database Interface

The database provides an interface to the physical schema through a set of kernel functions. All of the queries listed in Section 3.13 can be specified using only the kernel functions. However, this approach has two disadvantages. First, the underlying details of the physical schema are exposed to the user. Second, specifying the queries only in terms of the kernel functions becomes cumbersome and exposes details of the physical schema. As an alternative, a separate layer of abstraction called the *vulnerability database interface* is provided. This layer provides a set of form-based queries that interact with the kernel functions. The set of queries currently supported in the vulnerability database layer of the prototype are listed in Table 3.6.

Query	**Interface Function**
List all the vulnerabilities in system X.	list_vulnerability $< system >$
List all vulnerabilities in system X that occur because of Y.	vulner_conseq $< vulner >< system >$
List all vulnerabilities which result in Z.	list_conseq $< conseq >$
Given vulnerability V, classify it.	classify $< vulner >$
Given vulnerability V, find all systems that have this vulnerability.	find_system $< vulner >$
List all vulnerabilities in system X that will result in Y.	vulner_result $< system >< conseq >$
List all systems which have a vulnerability that will cause Y.	list_systems $< systems >$
Given vulnerability V and system X, what were (or could have been) the consequence if exploited.	conseq $< vulner >< system >$
Give a brief description of how V was (could have been) exploited.	exploit $< vulner >$
List any reference to V in the literature.	descr_fix $< vulner >$
Give a description of the fix .	literature $< vulner >$
List all patches applied to system X.	patches_to $< system >$
For a vulnerability V, are there any practical work-arounds?	work-around $< vulner >$
For system X, version Y, what vulnerabilities have no supported patch?	nopatch $< system >$
For all systems that have a vulnerability V, list their vendors.	sysinfo $< vulner >$

Table 3.6 Vulnerability Database Interface Functions

3.17 User Interface

A user can interact with our database either by using command line arguments or through a graphical user interface. The graphical interface was written using the Tk toolkit [Ous94] and provides a flexible interface to the vulnerability database functions. All of the functions listed in Table 3.6 can be invoked through the interface.

Figure 3.8 shows the main commands screen that shows all the functions that can be invoked. After a function has been selected, it may be invoked using the run button. This presents another window with entry fields for the needed arguments. Figure 3.9 shows the dialogue window for the `vulner_conseq` function. It shows the entry fields for the two needed arguments. A template representation for the input fields was chosen so that a user would not need to remember the order or number of arguments required by each function. A user can also obtain a help window for a selected function by using the help command in the main screen. The help screen contains a brief description of the function, the arguments required and the usage syntax. Figure 3.10 shows a help screen for the `vulner_conseq` function.

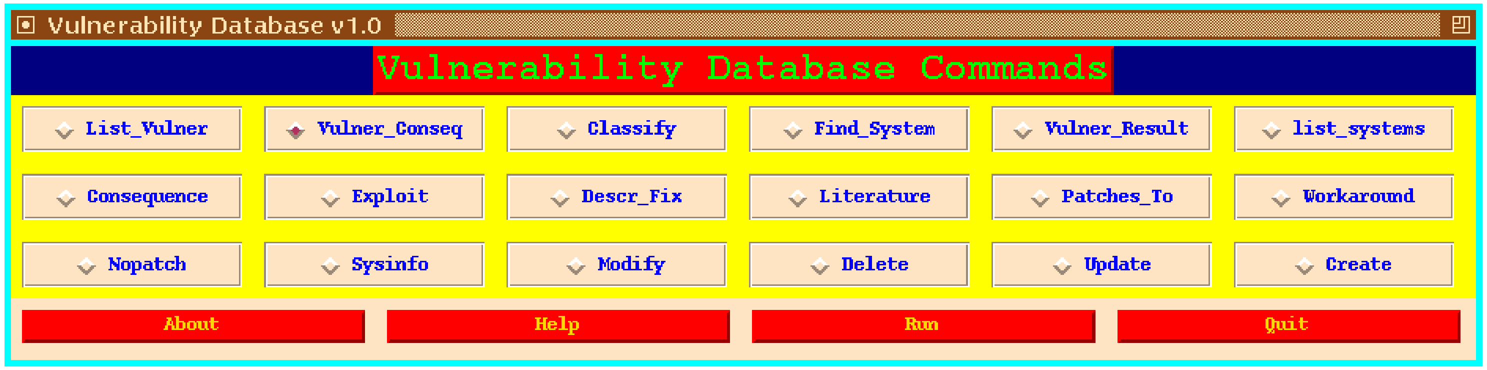

Figure 3.8 Vulnerability Database Commands Screen

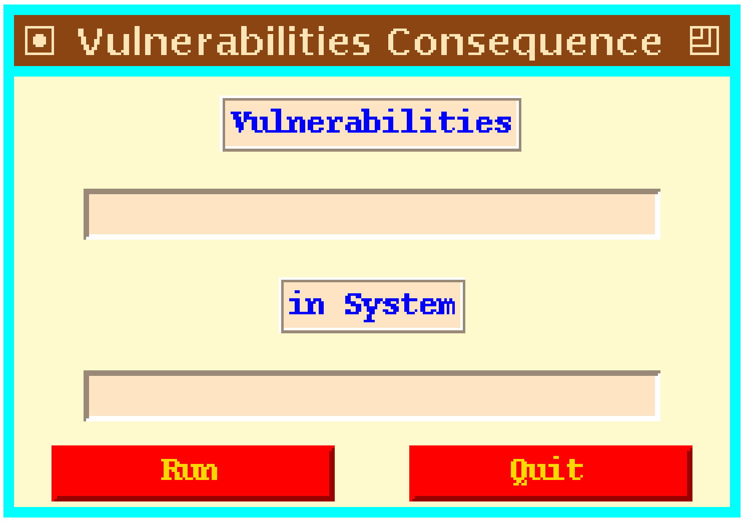

Figure 3.9 Function Interface

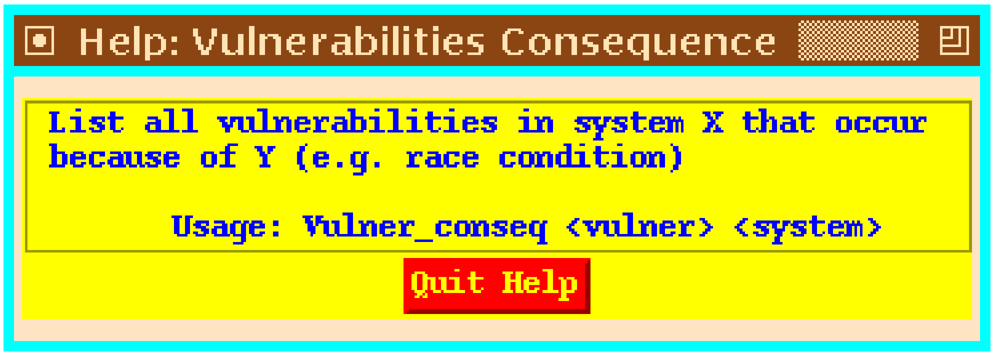

Figure 3.10 Help Window

3.18 Integrity Constraints

In a relational database, entity and referential integrity constraints must be maintained when creating, modifying or updating relations to maintain the integrity of stored information. The referential and entity integrity constraints in the vulnerability database were discussed in section 3.3.2 and 3.3.3. This section describes how these constraints are implemented in the database.

3.18.1 Entity Integrity Constraints

Entity integrity constraints specify that the primary key of a relation, which is used to uniquely identify a tuple, is always defined. In the vulnerability database, entity integrity constraints are explicitly specified in the "integrity.conf" file. The format of entries in the files is as shown below:

```
vulner:vulner_id
contains_vulner:os:vulner_id
os:name
contains_patch:os:patch_number
patch_info:patch_number
```

The name of the relation is followed by the fields that serve as the primary key(s). This format allows more than one key to be specified in the integrity check and allows the maintainer to change the keys if desired. The entity integrity check can be invoked every time a relation is created, modified, or when a tuple is modified or deleted from the relation.

The primary keys that are used in the database are:

vulnerability number: For vulnerabilities obtained from a response team, the primary key is the same as the unique identifier used by a response team to advertise the vulnerability. For other vulnerabilities a unique number is assigned to each vulnerability tuple.

operating system name: The primary key for the operating system entity is formed by concatenating the name of the operating system to its version number. This field serves as a unique identifier and also implicitly stores the version number.

patch number: Patch number is the same unique key used by vendors to identify a distributed patch.

3.18.2 Referential Integrity Constraints

Referential integrity constraints are specified between tuples of two relations to maintain consistency among the tuples. In the vulnerability database, referential integrity constraints are explicitly specified in the "chk.conf" file. The tuples that have referential integrity constraints are specified as:

```
relation1:fieldA ==> relation2:fieldB
```

In the instance shown, `fieldA` in `relation1` specifies a referential integrity constraint on `fieldB` in `relation2`. Any changes to `fieldB` in `relation2` should be reflected in `fieldA` in `relation1`. Referential integrity checks should be invoked when tuples in the relations are modified, deleted, or updated.

3.19 Template Representation

The vulnerability database provides flexibility to change fields in the tuples contained in the relations, or to change the integrity constraints through an editable template representation. The different fields of the tuples are specified in the "db.conf" file using a format shown below.

```
#vulnerability#: id: source: classification
```

The example above shows a relation "vulnerability" that consists of three fields: "id", "source", and "classification".

The templates in "db.conf" are used by the database kernel routines when new tuples are added, modified, or updated. A template representation allows the maintainer of the database to easily add or delete fields from the tuples. The changes are then reflected in any subsequent CREATE, MODIFY, or UPDATE operations.

Integrity constraints in the vulnerability database are also specified as editable templates. This also allows maintainers to change primary keys, add new keys, or change referential integrity constraints by simply editing the templates in the integrity configuration files.

3.20 Data Sources

We have started populating the vulnerability database with attacks that led to a security compromise. The information was gathered from advisories issued by the Computer Emergency Response Team (CERT), information reported on an electronic security mailing list, and a survey of the literature. At present the database contains information on fourty nine known vulnerabilities. These vulnerabilities have been categorized based on our taxonomy, and include descriptions of how each was exploited, and the operating systems that it was found in.

4. SECURITY FAULT DETECTION TECHNIQUES

In Chapter 2 we discussed a classification scheme for security faults in Unix. The motivation behind the classification was to classify faults unambiguously into different categories and use them for data organization. The fault categories were used as the basis for data organization in the design of a vulnerability database. The design and implementation of the database was discussed in Chapter 3. As another application of our fault classification scheme, we have also identified software testing techniques that may be used to detect faults in each category.

In our taxonomy of faults, errors that share a common characteristic are classified in the same category. For example, synchronization errors share a common method of exploitation: the relative ordering of an operation sequence can be used to breach security. Faults in the condition validation category result from missing or incomplete condition checks. Configuration errors are operational faults that occur when software is adapted to a new environment after it has been developed. Environmental errors are operational dependent faults that manifest themselves when software is executed in a particular execution environment.

In the following sections we present a brief description of different software testing techniques. In Sections 4.6 through 4.9, we present a synthesis of these testing techniques and discuss their application to security faults.

4.1 Static Analysis

Static analysis techniques do not require execution of the software. These techniques rely on inspection of requirements and design documents, code walk-throughs, and formal methods of verification to detect coding errors in a program [DMMP87].

Static analysis may be performed manually or the process can be automated by using an analysis tool.

A major limitation of static analysis involves array references and pointer variables as the values referenced cannot be distinguished [DM91]. Experimental evaluation of code inspections and walk-throughs have shown these methods to be effective in finding 30% to 70% of logic design and coding errors in typical programs [Mye79]. Various static analysis techniques such as code walk-throughs, inspection of design and specification documents, and formal methods of verification were also used in the design of secure operating systems [MD79, PKKe79]. These systems provided provable security and were formally demonstrated to conform to the security requirements.

4.2 Symbolic Testing

In symbolic testing of programs, input data and output values are assigned symbolic values, that may be elementary symbolic values or expressions [DMMP87]. The possible executions of a program are characterized by an execution tree. The execution of a program during symbolic testing is performed by a symbolic evaluator that consists of an expression simplifier and a symbolic interpreter. Symbolic testing has been used in several symbolic evaluation and program proof systems. Howden [How78] studied the effectiveness of symbolic testing and reported that fifteen out of twenty two errors reported in a program may possibly be detected by symbolic testing.

Symbolic testing can be used to prove the correctness of a program. If a program under test is viewed as a finite set of assertion-to-assertion paths, the program is shown to be correct if each path is correct [DMMP87]. Symbolic testing has problems if a program contains loops or array variables. The limitations of symbolic testing because of loops and array variables does not present itself as an attractive choice to test complex operating system source code.

4.3 Path Analysis and Testing

Path analysis testing involves the selection of test data cases to execute chosen paths [DMMP87]. A test case is constructed by choosing one test point from each execution path of the program. This approach exercises all the possible paths through a program at least once. But practical and economical limitations restrict path testing to choosing only a subset of all possible paths. Path testing is meant to detect computation, path selection, and missing path errors [How76]. A computation error occurs when a computation statement along a path is computed incorrectly. A path selection error occurs when the predicates in a conditional construct are incorrect. Missing path errors result from missing predicates in the condition construct. Path analysis testing can be used in conjunction with domain analysis to design test cases. Domain testing is intended to detect path selection errors on or near the boundary of a path domain [DM91].

Control structure testing is a path coverage testing technique that exercises all the possible outcomes of a condition construct. Control structure testing is divided into condition testing, data flow testing, and loop testing. Condition testing focuses on the condition expressions in a program. Data flow testing selects paths of the program according to the use and definition of variables. Loop testing focuses exclusively on the validity of loop constructs in a program. We will explore condition testing in more detail and present it as a possible detection technique for condition validation errors. Condition testing is further refined into the following types of testing.

Branch Testing: Branch testing is the simplest control testing technique and involves evaluating the true and false branch of every conditional branch at least once [Mye79].

Branch and Relational Operator Testing (BRO): Branch and relational operator testing guarantees the detection of branch and relational operator errors in a condition provided that all boolean variables and relational operators in the condition occur only once and have no common variables [Tai89]. BRO uses condition constraints on a simple condition. A condition C has constraints $(D_1, D_2, \ldots, D_n)$ where D_i is a symbol specifying a constraint on the outcome of the ith simple condition in C. A condition constraint D for condition C is covered by an execution of C if during this execution of C the outcome of each simple condition in C satisfies the corresponding constraint in D.

For example, consider a condition of the form:

C: $B_1 \& (E_2 == E_3)$

where B_1 is a boolean expression and E_2 and E_3 are arithmetic expressions. Assuming short circuit evaluation of boolean operands, the constraint set for this condition is $\{$(true,=), (false,=),(true,<),(true,>)$\}$. Otherwise the constraint set would also include $\{$(false,<), (false,>)$\}$. Coverage of the constraint set guarantees that all boolean and relational operator errors in C will be detected.

It should be noted that condition testing can only reveal problems with evaluating conditions and the corresponding results. It cannot detect missing condition constructs. In addition to test cases that are used in control structure testing, test cases should also be designed to detect missing condition checks.

4.4 Functional Testing

In functional program testing, the design of a program is viewed as an abstract description of its design and requirements specifications [DMMP87]. The objective is to find out when the input-output behavior of the program does not agree with its specifications. Functional testing enables us to derive test cases that fully exercise all functional requirements for a program. Functional testing attempts to find the following errors in a program [Pre92].

- incorrect or missing functions
- interface errors

- errors in data structures
- performance errors
- initialization and termination errors

It is reported that functional testing exposed thirty eight of the forty two known errors in the release of a major software package [DMMP87]. In the following sections, we present a functional testing technique and a test case design technique that can be used in conjunction with functional program testing. These fault detection techniques can be used to detect condition validation errors.

Boundary value analysis (BVA) is a technique for designing test cases to uncover boundary value errors [Mye79]. It provides guidelines for complete boundary testing and derives tests from both input and output domains. Boundary value analysis can be used to derive test cases to detect missing or incorrectly specified limit checks for system resources, and static-sized data structures. The guidelines provided for boundary value analysis are as follows:

1. If an input condition specifies a *range* bounded by values a and b, test cases should be designed with values a and b, just above and just below a and b respectively. In addition, extreme values at both the high and low end should also be considered to ensure that those conditions are handled properly.

2. If an input condition specifies a number of values, test cases should be developed that exercise the minimum and maximum numbers. Values just above and just below are also tested.

3. Apply guidelines 1 and 2 to output conditions.

4. If internal program data structures have prescribed boundaries, test cases should be designed to exercise those boundaries.

4.4.1 Syntax Testing

Syntax testing is a functional testing technique, in which the object under test is treated as a black box that can accept certain inputs, reject others and process some. Syntax checking can be performed as part of the system functional test or as part of the acceptance test. The primary objective of this technique is not to establish the correctness of the system under test but to establish that the system is not vulnerable to ill-formatted input. Beizer [Bei83] has summarized the objective of syntax testing as follows:

- The element does not fail when subjected to bad inputs.
- The element does not cause another element to fail, even though it does not fail itself when subjected to bad inputs.
- The element does not corrupt the database when subjected to bad inputs, even though it does not fail itself.
- The element rejects all bad inputs and accepts all good inputs.
- It performs the correct processing on good inputs.

Some of the errors that can result from improper input validation are outlined below. Test cases should be designed to check for these errors in the implementation.

- High level syntax errors: Violations in the syntax specification through inappropriate combination of fields.
- Field-syntax errors: Syntax errors associated with a field.
- Delimiter errors: Violation of the rules governing the placement and the type of characters that must appear as separators between fields. These include: missing delimiter, wrong delimiter, not a delimiter, and too many delimiters.
- Field-value errors: Errors that occur because of the contents of a field. These include boundary values and excluded values.

- Syntax-context errors: There is a possibility of an error when the contents of a field dictate the content of subsequent fields.
- Field-value correlation errors: The contents of two or more fields are correlated by a functional relation between them and there may not be full freedom in picking their values.
- State-dependency errors: The permissible syntax and/or field value is conditional on the state of the system or the routine.

Syntax checking should be performed on at least the following modules as they handle a major portion of the input that enters the system.

- All user-interface programs including terminal device input routines.
- All communication protocols programs that accept user input.
- Internal interface routines that call supervisor routines.
- Dedicated data validation routines and modules.

4.5 Mutation Testing

Strictly speaking, mutation testing is not a testing technique but a technique for measuring the adequacy of test data [DMMP87]. A brief overview of mutation testing from [DM91] is presented here. A more detailed discussion can be found in [Bud80, DMMP87].

Assume P is the program under test, P_c is the correct version of the program and is the same as P if P is correct. T is the set of test cases on which P is being tested and D is the domain of P_c such that $P \subseteq D$. The correctness of P is assumed relative to some set of specifications. Mutation analysis relies on a set F of faults. Given a program P being tested, each of the faults in F is introduced into P one by one. By introducing a fault into P, a slightly different program known as the *mutant* of P is produced. Mutating P using all the elements of F results in a set $\mathcal{M}$ of mutants. A mutant M is *killed* if its behavior is different from P when $M \in \mathcal{M}$ is executed against a test case $t \in T$. If M cannot be killed by any $t \in T$ it could be because:

- M is equivalent to P, and will always behave identical to P on all $t \in D$

  or

- there exists a test case $t' \notin T$ but $t' \in D$, for which M behaves different than P.

Mutation testing requires knowledge of the implementation language to design the language dependent mutants.

In the following sections, we present an analysis of the different fault detection methods outlined earlier and discuss how each method can be used to detect the different fault types in our classification scheme.

4.6 Condition Validation Errors

In our classification scheme, we identified five types of faults that can result when a condition is not being evaluated properly. The four sub-categories of condition validation faults are: boundary condition errors, input validation errors, access right validation errors, origin validation errors, and failure to handle exceptional conditions. In the following sections we present software testing techniques that can be applied to detect faults in each category.

4.6.1 Boundary Condition Errors

Boundary condition errors can be detected by using boundary value analysis (BVA) to design test cases for functional testing of modules. BVA ensures that the test cases exercise the boundary conditions that can expose boundary condition errors in the module under test [Mye79].

In addition to functional testing, mutation testing can also be used to detect boundary conditions by designing appropriate language dependent mutants. For instance Agrawal et al. [ADHe89] describe a C language mutant for checking boundary conditions in scalar variables. For a statement such as:

p = a + b

The mutants that are generated consist of:

p = a + b + 1

and

p = a + b − 1

These mutants are killed if they cause an overflow or underflow in the program.

Path analysis testing in conjunction with domain analysis can be applied to detect boundary condition errors. Domain analysis has been studied with two variables and examined with three variables [JK80, ADJ]. The main disadvantage of domain testing is that it can only be applied to a small number of variables as the difficulty of selecting test cases becomes increasingly complex. In an experiment by Howden, path analysis revealed the existence of one out of three path selection errors [How76].

4.6.2 Input Validation Errors

Input validation errors result when a functional module fails to properly validate the input it accepts from another module or another process. Failure to validate the input may cause the module accepting input to fail or it may indirectly cause another interacting module to fail.

Syntax testing can be used to verify that functional modules that accept input from other processes or modules do not fail when presented with ill-formatted input. Syntax testing was described in Section 4.4.1 including test case scenario that should be tested.

Path analysis and testing can be applied to detect scenarios where a certain execution path may be chosen based on the input. In an experiment conducted by Howden, path testing revealed the existence of nine out of twelve computation errors.

Symbolic testing and static analysis are not viable choices for detecting input validation errors. Most of the input validation errors result from input received during run-time. This observation does not favor using symbolic or static testing as an attractive choice to detect input validation errors.

4.6.3 Access Right Validation Errors

Access validation errors result when a subject is allowed to invoke an operation on an object outside its access domain. In most programming languages these conditions are coded as conditional constructs and an execution path is chosen based on the outcome of the evaluated condition. An improperly specified condition can lead to an incorrect execution path that can invalidate the access right check.

Path analysis can be used to detect errors that result from incorrectly specified condition constructs. Branch and Relational Operator testing (BRO) is a test case design techniques that can aid in the design of test cases that can expose access validation errors.

In programming languages where access validation checks are coded as conditional constructs, mutation testing can also be used to detect these errors. Mutants can be designed to provide condition analysis for conditional constructs. In the C programming language, two mutants can be created for an `if` statement as follows [ADHe89]. For the statements `if (e) S` the two mutants are:

1. `v=e; if (trap_on_true(v)) S`

2. `v=e; if (trap_on_false(v)) S`

When `trap_on_true(false)` is executed, the mutant is killed if the function argument value is true(false). If the value if not true(false), then the mutants continue execution.

The above mentioned mutants only provide partial coverage for the `if` statement. To provide complete coverage in the case of statements of the form: `if (c)` S_1 `else` S_2 the mutants needed to provide complete coverage are:

1. `if (c) trap_on_statement() else` S_2

2. `if (c)` S_1 `else trap_on_statement`

These two mutants are equivalent to providing branch coverage and encourage test cases to be designed such that each branch is executed at least once.

4.6.4 Origin Validation Errors

If a system does not properly authenticate a user or process it may allow unauthorized subjects to access the system. Similarly, if a program does not properly authenticate the shared data or libraries it may be fooled into using modified data that can lead to a security breach.

It is a difficult task to verify that a system is not vulnerable to identity compromise attacks. We are not aware of any fault detection technique that can be employed to expose origin validation errors. To ensure that a system properly authenticates its users, we can apply formal methods of verification to ensure that the implementation adheres to the security requirements and also ensure that the underlying protocol does not contain any weaknesses that may be exploited.

4.6.5 Failure to Handle Exceptional Conditions

A security breach can be caused if a system fails to handle an exceptional condition. This can include unanticipated return codes, and failure events.

Static analysis techniques such as inspection of design documents, code walk-throughs, and formal verification of critical sections can be used to ensure that a system can gracefully handle any unanticipated event.

Path analysis testing can also be performed on small critical sections of code to ensure that all possible execution paths are examined. This can reveal problems that may not have been anticipated by the designers or overlooked because of complexity.

4.7 Environment Errors

Enviromental errors are dependent on the operational environment, which makes them difficult to detect [Spa90]. It is possible that these vulnerabilities manifest themselves only when the software is run on a particular machine, under a particular operating system, or a particular configuration.

As environment errors are dependent on the run-time conditions, a static analysis of the code cannot detect these faults. The underlying modules and algorithms may even be formally demonstrated to be correct yet the program fails to function correctly on a specific set of input. [Spa90].

Spafford [Spa90] used mutation testing to uncover problems with integer overflow and underflow. Mutation testing can be used to design test cases that exercise a specific set of inputs unique to the run-time environment. Path analysis and testing can also be applied to sections of the code to ensure that all possible inputs are examined.

4.8 Synchronization Faults

In our fault classification scheme, synchronization faults are introduced because of the existence of a timing window between two operations or faults that result from improper or inadequate serialization of operations. One possible sequence of actions that may lead to a synchronization fault can be characterized as [KS94]:

1. A process acquires access to an object to perform some operation.

2. The process's notion of the object changes indirectly.

3. The process performs the operation on the object.

Mutation testing can be used to detect synchronization faults in a program. To detect faults that are introduced by a timing window between two operations, a `trap_on_execution` mutant can be placed between these two operations. The mutant terminates execution of the program if certain specified conditions are not satisfied.

For instance, a timing window between the access permission checks and the actual logging in `xterm` could be exploited to compromise security [CA-93a]. A mutant for this vulnerability could be designed that terminated execution thus killing the mutant, if the access checks had been completed. This mutant could be placed between the access checks and the logging to detect the race condition.

Mutants can also be designed to detect improper serialization operations. Consider a set of n statement that must be executed sequentially to ensure correct operation. We assume that the statements do not contain any instructions that break the sequential lock-step execution. We can design $(n! - 1)$ mutants that rearrange the order of the n execution statements. These mutants are killed when the mutated program produces a different result than the original program.

Other software testing techniques we discussed earlier cannot be applied to detect synchronization errors. The emphasis of path analysis is to provide coverage that ensures that all possible execution paths through a program are executed at least once. This would not detect synchronization errors that depend on relative timing and serialization of operations. Timing errors and improper serialization are detected by introducing faults in the code as in mutation testing.

Static analysis is not a viable option because synchronization errors depend on the runtime conditions.

4.9 Configuration Errors

Configuration errors may result when software is adapted to new environments or from a failure to adhere to the security policy. Configuration errors comprise of faults introduced after software has been developed and are faults introduced during the maintenance phase of the software life-cycle.

A static audit analysis of a system can reveal a majority of configuration errors. Among the various software testing techniques discussed, static analysis is the most effective in detecting configuration errors. The other testing techniques we discussed focus on detecting coding faults and would not be effective in exposing configuration errors. The static audit of a system can be automated by using static audit tools such as COPS [FS91] and Tiger [SSH93] that search a system for known avenues of penetration.

Using COPS as an example, we illustrate the types of configuration errors that can be detected. COPS contains the following checks to detect security faults that led to a compromise of a system [FS91].

dir.check, file.chk: Check a list of specified directories and files to ensure that they are not world-writable. Typically, this list includes `/etc/passwd, /.profile, /etc/rc, /, /bin, /usr/adm, /etc`. These checks ensure that critical files and directories do not have incorrect permissions that violate a security policy.

pass.chk: This checks for poor password choices. This includes user name, common words and login identifiers. This is to prevent brute force attempts to crack passwords.

group.chk, passwd.chk: These check `/etc/passwd, yypasswd, /etc/groups`, and `ypgroup` for the validity of contents and a variety of known problems including blank spaces, null passwords, and non-standard fields.

cron.chk, rc.chk: These checks ensure that an intruder cannot run a program with root privileges at system startup by checking that files referenced in `/etc/rc*` are not world writable.

dev.chk: This checks that `/dev/kmem, /dev/mem, /etc/fstab` are not world readable or writable. If the permissions are not set correctly, it allows an intruder to read or write directly from a device.

home.chk, user.chk: These checks ensure that each user's home directory and initialization files are not world-writable. These checks are intended to prevent an individual user's account being sabotaged by an intruder.

root.chk: This checks root startup files for incorrect `umask` settings and search paths containing the current directory. This ensure that files owned by the superuser are created with the correct permissions.

suid.chk: This program checks for changes in SUID file status on a system. This is to prevent a system from several known penetration attacks that result from information flow problems.

kuang: This is an expert system written by Robert W. Baldwin of MIT. This checks if a given user's account can be compromised based on a set of rules.

The checks we described above can detect many of the known configuration errors. These checks should be updated as new configuration errors are discovered. The vulnerability database described in chapter 3 can be used to keep the set of checks current. As new configuration errors are added to the database, information about fault characteristics can be extracted and used to write new checks for the audit tools.

4.10 Summary

Table 4.1 shows a summary of different software testing methods that can be used to detect different types of faults. We use "+" to denote that the technique should be effective, and has been demonstrated to detect faults with similar characteristics. A "$-$" denotes that the technique cannot effectively be used. A "?" denotes that the testing method can be effective and a quantitative study may be performed as part of the future work.

Errors	Functional Testing	Path Analysis	Static Analysis	Mutation Testing
Boundary condition	+	?	?	+
Input validation	+	+	−	−
Access right	−	+	?	?
Origin validation	−	−	+	−
Failure to handle exceptions	?	+	+	?
Environment	?	+	−	+
Race condition	−	−	−	+
Serialization	−	−	−	+
Configuration	−	−	+	−

Table 4.1 Comparison of Different Software Testing Methods

5. CONCLUSIONS AND FUTURE WORK

The main contribution of this thesis is the security fault classification scheme for the Unix operating system. Our classification scheme allows us to categorize distinctly each security fault according to the specified criteria. This distinguishes our taxonomy from other classification schemes that do not explicitly specify the membership criteria for the faults. To demonstrate the applicability of our classification scheme we used it as a basis for the design of a vulnerability database. The database was used to store security faults found in different versions of Unix. This database can be used in conjunction with an intrusion detection system, or a static audit tool to search for known avenues of penetration. Furthermore, we used the fault categories to identify software testing methods. These methods can be applied during the test phase of the software life-cycle or can be used during a penetration analysis.

Traditionally, testing focused on proving the functional correctness of software. Little or no importance was given to security testing. Personnel who maintained a system were also responsible for its security. Some sites used penetration analysis to uncover existing security vulnerabilities. But a majority of computer sites could not afford a comprehensive penetration analysis and vulnerabilities went unnoticed until they had led to a security compromise.

Coding faults can be detected by adequate testing. Thus, security compromises that resulted by exploiting such faults could have been prevented if the faults were detected and fixed during the development cycle. To develop more reliable systems, more emphasis should be placed on security testing and system testing methods should be designed to detect security faults.

As part of the future work, we may extend our classification scheme to other modern operating system. Many modern systems are based on a software architecture that is different from Unix's structure. These include micro-kernels, object-oriented, and distributed operating systems. The criteria used in our taxonomy do not rely on implementation details and are designed to encompass general characteristics of a fault. Also, our existing categories can be extended to include faults that cannot be classified into existing categories. We believe that these factors would enable us to extend our classification scheme to include faults in various operating systems.

Another area for future work may be to evaluate the software testing methods on different operating systems. This would provide quantitative data on the effectiveness of each method. These methods may also be used to develop a test-suite that can be used to measure the secure-worthiness of different systems. Using this suite, it may be possible to develop a set of metrics that can be used for security certification of different systems.

BIBLIOGRAPHY

BIBLIOGRAPHY

[A+76] R.P. Abbott et al. Security Analysis and Enhancements of Computer Operating Systems. Technical Report NBSIR 76-1041, Institute for Computer Science and Technology, National Bureau of Standards, 1976.

[ADHe89] H. Agrawal, R. DeMillo, R. Hathaway, and et al. Design of Mutant Operators for the C Programming Language. Technical Report SERC-TR-41-P, Software Engineering Research Center, Purdue University, 1989.

[ADJ] DeMillo R. A, Hocking E. D, and Meritt M. J. A Comparison of Some Reliable Test Data Generation Procedures. Technical report, Georgia Institue of Technology.

[AMP76] C.R. Attansio, P. W. Markstein, and R. J. Phillips. Penetrating an operating system: a study of VM/370 integrity. *IBM Systems Journal*, pages 102–116, 1976.

[And72] J P Anderson. Computer security technology planning study. Technical Report ESD-TR-73-51, USAF Electronic Systems Div., 1972.

[Bei83] Boris Beizer. *Software Testing Techniques*. Electrical Engineering/Computer Science and Engineering Series. Van Nostrand Reinhold, 1983.

[BH78] R. Bibsey and D. Hollingworth. Protection analysis project final report. Technical Report ISI-RR-78-13, Institue of Information Sciences, University of Southern California, 1978.

[Bis86] Matt Bishop. Analyzing the Security of an Existing Computer System. IEEE Fall Joint Computer Conference, November 1986.

[BP84] V. Basili and Barry T. Perricone. Software Errors and Complexity: An Empirical Investigation. *Communications of the ACM*, 27(1):42–51, January 1984.

[BPC75] Richard Bibsey, Gerald Popek, and Jim Carlstead. Inconsistency of a single data value over time. Technical report, Information Sciences Institute,University of Southern California, December 1975.

[BR87] Victor R. Baisli and H. Dieter Rombach. Tailoring the Software Process to Project Goals and Environments. In *Proceedings of the 9th International Conference on Software Engineering*, pages 345–357. IEEE Press, 1987.

[BS88] Lubomir Bic and Alan Shaw. *The Logical Design of Operating Systems.* Prentice Hall, 1988.

[Bud80] T.A. Budd. *Mutation Analysis of Program Test Data.* PhD thesis, Yale University, May 1980.

[CA-89] CERT advisory CA-89:06. Computer Emergency Response Team Advisory, 1989.

[CA-90a] CERT advisory CA-90:11. Computer Emergency Response Team Advisory, 1990.

[CA-90b] CERT advisory CA-90:12. Computer Emergency Response Team Advisory, 1990.

[CA-91a] CERT advisory CA-91:18. Computer Emergency Response Team Advisory, 1991.

[CA-91b] CERT advisory CA-91:19. Computer Emergency Response Team Advisory, 1991.

[CA-93a] CERT advisory CA-93:17. Computer Emergency Response Team Advisory, 1993.

[CA-93b] CERT advisory CA-93:19. Computer Emergency Response Team Advisory, 1993.

[CA-94] CERT advisory CA-94:02. Computer Emergency Response Team Advisory, 1994.

[Car78] Jim Carlstead. Serialization: Protection errors in operating systems. Technical report, Information Sciences Institute, University of Southern California, April 1978.

[CBP75] Jim Carlstead, Richard Bibsey II, and Gerald Popek. Pattern-directed protection evaluation. Technical report, Information Sciences Institue,University of Southern California, June 1975.

[Com88] Douglas E. Comer. *Internetworking with TCP/IP.* Prentice Hall, 1988.

[CPRZ89] L. A. Clarke, A. Podgruski, D. J. Richardson, and S. Zeil. A Formal Evaluation of Data Flow Path Selection Criteria. *IEEE Transactions on Software Engineering*, 15(11):1318–1332, November 1989.

[Den83] Dorothy Denning. *Cryptography and Data Security.* Addison-Wessley Publishing Company, 1983.

[DGM78] A. Dniestrowski, J. M. Guillaume, and R. Mortier. Software Engineering in Avionics Applications. In *Proceedings of the 3rd International Conference on Software Engineering*, pages 124–131. IEEE Press, 1978.

[Dis85] A. V. Discolo. 4.2 BSD Unix Security. Technical report, University of California - Santa Barbara, 1985.

[DM91] Richard A. DeMillo and Aditya P. Mathur. On the Use of Software Artifacts to Evaluate the Effectiveness of Mutation Analysis for Detecting Errors in Production Software. Technical report, Software Engineering Research Center, Purdue University , SERC-TR-92-P, March 1991.

[DMMP87] Richard A. DeMillo, W. Michael McCracken, R. J. Martin, and John F. Passafiume. *Software Testing and Evaluation.* The Benjamin/Cummings Publishing Company Inc., 1987.

[EN89] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems.* The Benjamin/Cummings Publishing Comany Inc., 1989.

[End75] A. Endres. An analysis of errors and their causes in system programs. *IEEE Transactions on Software Engineering*, SE-1:140–149, 1975.

[FS91] Daniel Farmer and Eugene H. Spafford. The COPS Security Checker System. Technical Report CSD-TR-993, Software Engineering Research Center, Purdue University, September 1991.

[GJM91] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering.* Prentice Hall, 1991.

[Gla81] Robert L. Glass. Persistent software errors. *Transactions on Software Engineering*, SE-7(2):162–168, March 1981.

[GS91] Simson Garfinkel and Eugene Spafford. *Practical Unix Security.* O'Reily and Associates, 1991.

[H+80] B. Hebbard et al. A penetration analysis of the Michigan terminal system. *ACM SIGOPS Operating System Review*, 14(1):7–20, 1980.

[Hal90] Anthony Hall. Seven myths of formal methods. *IEEE Software*, pages 11–20, September 1990.

[HB76] Dennis Hollingworth and Richard Bibsey II. Allocation/deallocation residuals. Technical report, Information Sciences Institute, University of Southern California, June 1976.

[How76] W. E. Howden. Reliability of the Path Analysis Testing Strategy. *IEEE Transactions on Software Engineering*, SE-2(3):208–214, 1976.

[How78] W. E. Howden. An Evaluation of the Effectiveness of Symbolic Testing. *Software Practice and Principle*, 8(4):381–397, July-August 1978.

[How81a] William E. Howden. A survey of dynamic analysis methods. *Tutorial: Software Testing & Validation*, pages 209–231, 1981.

[How81b] William E. Howden. A survey of static analysis methods. *Tutorial: Software Testing & Validation*, pages 101–115, 1981.

[HSP] David K. Hess, David R. Safford, and Udo W. Pooch. A Unix Network Protocol Security Study: Network Information Service. Technical report, Texas A & M University.

[IEE90] IEEE. ANSI/IEEE Standard Glossary of Software Engineering Terminology. IEEE Press, 1990.

[ISV95] Dave Icove, Karl Seger, and William VonStorch. *Computer Crime: A Crimefighter's Handbook*. O'Reilly & Associates, Inc., 1995.

[JK80] White L. J and Cohen E. K. A Domain Strategy for Computer Program Testing. *IEEE Transactions on Software Engineering*, 6(3):247–257, May 1980.

[Knu89] D.E. Knuth. The Errors of TeX. *Software Practice and Experience*, 19(7):607–685, 1989.

[KS94] Sandeep Kumar and Eugene Spafford. A Pattern Matching Model for Misuse Intrusion Detection. In *17th National Computer Security Conference*, 1994.

[L+93] Carl Landwher et al. A taxonomy of computer program security flaws. Technical report, Naval Research Laboratory, November 1993.

[Lin75] Richard Linde. Operating system penetration. In *National Computer Conference*, pages 361–368, 1975.

[Lip79] M. Lipow. Prediction of Software Failures. *Journal of Systems and Software*, 1(1):71–75, 1979.

[Mar90] Brian Marick. A survey of software fault surveys. Technical Report UIUCDCS-R-90-1651, University of Illinois at Urbana-Champaign, December 1990.

[MB77] R. W. Motley and W. D. Brooks. Statistical Prediction of Programming Errors. Technical Report RADC-TR-77-175, 1977.

[MD79] E.J. McCauley and P.J. Drongowski. KSOS - The design of a secure operating system. National Computer Conference, 1979.

[Mye76] Glenford J. Myers. *Software Reliability*. Wiley-Interscience, 1976.

[Mye79] G. Myers. *The Art of Software Testing*. Wiley, 1979.

[Ous94] John K. Ousterhout. *Tcl and Tk Toolkit*. Addison Wesley Publishing Company, 1994.

[OW84] Thomas J. Ostrand and Elaine J. Weyuker. Collecting and Categorizing Software Error Data in an Industrial Environment. *Journal of Systems and Software*, 4:289–300, 1984.

[PAFB82] D. Potier, J.L. Albin, R. Ferrol, and A. Bilodeau. Experiments with Computer Software Complexity and Reliability. In *Proceedings of the 6th International Conference on Software Engineering*, pages 94–103. IEEE Press, 1982.

[PKKe79] G. J. Popek, M. Kampe, C. S Kline, and et al. UCLA Secure Unix. In *Proceedings of the National Computer Conference*, pages 335–364, 1979.

[Pre92] Roger Pressman. *Software Engineering: A Practitioner's Approach*. McGraw Hill, Inc., Third edition, 1992.

[RDB75] R. J. Rubey, J. A. Dana, and P. W. Biche. Quantitative aspects of software validation. *IEEE Transactions on Software Engineering*, SE-1:150–155, 1975.

[Rub75] Raymond J. Rubey. Quantitative Aspects of Software Validation. *SIGPLAN Notices*, SE-5(3):276–286, May 1975.

[SB75] M.L. Schooman and M.I. Bolsky. Types, Distribution, and Test and Correction Times for Programming Errors. In *Proceedings of the 1975 International Conference on Reliable Software*, 1975.

[Sch75] W. L. Schiller. The Design and Sepcifications of a Security Kernel for the PDP 11/45. Technical Report ESD-TR75-69, THE MITRE Corporation, 1975.

[Sch79a] R. R. Schell. Computer Security: the Achilles Heel of the Electronic Air Force? *Air University Review*, 30(2):16–33, 1979.

[Sch79b] N. F. Schneidewind. Software metrics for aiding program development debugging. In *National Computer Conference*, 1979.

[Sch93] Christoph Schuba. Addressing Weaknesses in the Domain Name System Protocol. Master's thesis, Purdue University, 1993.

[SCS77] M. D. Schroeder, D. D. Clark, and J. H. Saltzer. The MUTLICS Kernel Design Project. *ACM Operating Systems Review*, 11(5):43–56, 1977.

[Spa89] Eugene H. Spafford. Crisis and Aftermath. *Communications of the ACM*, 32(6):678–687, June 1989.

[Spa90] Eugene H. Spafford. Extending Mutation Testing to Find Environmental Bugs. *Software Practice and Principle*, 20(2):181–189, Feb 1990.

[SSH93] David R. Safford, Douglas Lee Schales, and David K. Hess. The TAMU security package. In Edward Dehart, editor, *Proceedings of the Security IV Conference*, pages 91–118, 1993.

[Tai89] K. C. Tai. What to do beyond branch testing. *ACM Software Engineering Notes*, 14(2):58–61, April 1989.

[WB85] David M. Weiss and Victor R. Basili. Evaluating Software development by Analysis of Changes: Some Data from the Software Engineering Laboratory. *IEEE Transactions on Software Engineering*, SE-11(2):3–11, February 1985.

[web88] Webster's New World Dictionary of Computer Terms. Webster's New World, New York, 1988.

[Wil81] A. L. Wilkinson et al. A penetration analysis of a Burroughs large system. *ACM SIGOPS Operating System Review*, 15(1):14–25, 1981.

[Wit90] Carol Withrow. Error Density and Size in Ada Software. *IEEE Software*, 7(1):26–30, January 1990.

[Y+85] Shen Yu et al. Identifying Error-Prone Software - An Empirical Study. *Transactions on Software Engineering*, SE-11(4):317–323, April 1985.

[YT91] Michal Young and Richard N. Taylor. Rethinking the Taxonomy of Fault Detection Techniques. Technical report, Software Engineering Research Center, Purdue University, September 1991.

[Zwa92] Vladmir Zwass. *Management Information Systems*. Dubuque, Wm. C. Brown, 1992.

APPENDICES

Appendix A:
Description of Software Fault Studies

[BR87] Basili and Rombach report on software faults in projects characterized by high code reuse, established processes, and high turnover among programmers.

[Gla81] Glass surveyed 100 faults each from two software systems for military aircraft. The first was coded in 500,000 instructions and was built by 150 programmers. The second system had 100,000 instructions and was built by 30 programmers. All the faults were reported after delivery of the software.

[DGM78] Dniestrowski et al. describe a digital flight control/avionics real time system. The system contained 10,000 lines of code written in LTR (a special high level language) and assembly.

[OW84] Ostrand and Weyuker report on 173 faults discovered during the development and system testing of an interactive special-purpose editor coded in 10,000 lines of a high-level language and 1000 lines of assembly.

[Rub75] Rubey reports on faults found from over a dozen validation efforts. The system included in the study were small commercial real-time control programs.

[BP84] Basili and Perricone discuss faults in a 90,000 lines Fortran project. The system was general purpose program for satellite planning studies. It was characterized by rapidly changing requirements and by code and design reuse.

[Y^{+}85] Shen et al. describe faults discovered after release of software in three different programs: a metric calculation tool written in Pascal, a compiler in PL/S, and a database system written primarily in assembly.

Appendix B: A Taxonomy of Unix Vulnerabilities

1. Synopsis: `lpr(1)` can be used to delete or create any file on the system. `lpr -s` allows users to create symbolic links to lpd's spool directory. After 1000 invocations, `lpr` will reuse files in the spool directory, and follow the previously installed link.

   Source: 8lgm advisory

   Operating System: SunOS 4.1.1 or earlier, BSD 4.3, BSD NET/2 derived systems, A/UX 2.01

   Classification: Condition validation error: check for origin.

2. Synopsis: If an access list of hosts in `/etc/exports` is longer than 256 characters, or if the cached list of netgroups exceeds the cache capacity then the filesystem can be mounted by any user.

   Source: CERT advisory CA-94:02

   Operating System: SunOS on Sun3 and Sun4

   Classification: Condition validation error: check for limit.

3. Synopsis: In systems that used `delivermail(8)` the mail message could be appended to the file by sending mail to the filename instead of a user. This could be used to create an arbitrary entry in `/etc/passwd`.

   Source: See [Bis86].

   Operating System: BSD systems that used `delivermail`.

   Classification: Condition validation error: check for origin (destination).

4. Synopsis: It is possible to lock the password file and prevent any user from changing his/her password.

   Source: security distribution list

   Operating System: 4.2 BSD and derived systems

   Classification: Condition validation error: check for limit.

5. Synopsis: `rcp` can be exploited by any trusted host in `/etc/hosts.equiv` or `./rhosts`.

   Source: CERT advisory CA-89:07

   Operating System: SunOS4.x

   Classification: Condition validation error: check for origin.

6. Synopsis: Any user can gain root access to a machine running HP's NIS `ypbind`.

Source: CERT advisory CA-93:01

Operating System: Any system running HP's NIS ypbind.

Classification: Condition validation error: check for origin.

7. Synopsis: A system running NIS/RPC/UDP can be penetrated by generating a fake `/etc/password` entry in response to a NIS client request.

   Source: See [HSP].

   Operating System: Systems running NIS/RPC/UDP.

   Classification: Condition validation error: check for origin.

8. Synopsis: A vulnerability in Majordomo software allows users access to the system without authentication.

   Source: CERT advisory CA-94:11

   Operating System: Any system running Majordomo up to and including version 1.91.

   Classification: Condition validation error: check for origin.

9. Synopsis: `/usr/etc/modload` and `$OPENWINDOWS/bin/loadmodule` can be exploited to execute a user's program using the effective id of root.

   Source: CERT advisory CA-93:18

   Operating System: SunOS 4.1.1, 4.1.2, 4.1.3(c)

   Classification: Condition validation error: check for access rights.

10. Synopsis: TIOCCONS can be used to re-direct console input/output away from console regardless of the permissions on the terminal device.

    Source: CERT advisory CA-90:12

    Operating System: SunOS

    Classification: Condition validation error: check for access rights.

11. Synopsis: Implementation of `/usr/sys/sys/kern_exec.c` takes the text and data sizes in the header of the file being exec'd to be true. This can cause a security breach when a program with a large data-size causes a coredump.

    Source: Security distribution list.

    Operating System: BSD 4.2

    Classification: Condition validation error: check for limits.

12. Synopsis: The password file reading routines use `fgets()` to read a string of input. These routines do not properly check if the last line was completely read. This can be used to create a bogus entry in the password file.

    Source: Security distribution list.

Operating System: BSD 4.3

Classification: Condition validation error: check for input syntax.

13. Synopsis: Setting the stack size on a Sun 386 causes a coredump and crashes the system. The problem occurs because the input is parsed incorrectly.

    Source: Security distribution list.

    Operating System: Sun 386 systems.

    Classification: Condition validation error: check for input.

14. Synopsis: The `xterm` emulator defines some escape sequences that can be sent to a user's terminal via a mail message. This can be used to invoke arbitrary commands on another user's terminal.

    Source: Security distribution list.

    Operating System: Systems running X windows version 11 release 3 and 4.

    Classification: Environment error: interaction between functionally correct modules

15. Synopsis: `rdist` uses `popen(3)` to execute `sendmail(3)` as root. It can be made to execute arbitrary programs as root.

    Source: 8lgm advisory

    Operating System: SunOS 4.1.2 or earlier, A/UX 2.0,1, BSD NET/2Derived systems

    Classification: Condition validation: check for origin of subject(program).

16. Synopsis: In version 6 Unix, if `su` could not open the password file, it would create a shell with the effective and real gid of root and grant root privileges to the user.

    Source: *Unreferenced source.*

    Operating System: Unix version 6.

    Classification: Condition validation: check for limits.

17. Synopsis: A vulnerability in NCSA HTTP daemon allows remote users access to the account under which `httpd` is running.

    Source: CERT advisory CA-95:04

    Operating System: Systems running NCSA HTTPD version up to and including version 1.3.

    Classification: Condition validation: check for limit.

18. Synopsis: In the code for `binmail` in the function `sendrmt()` there is a loop that copies the address up to the first "!" or `NULL`. If the input string is long enough to overwrite the stack, it will be executed when the procedure returns using the address stored on the stack.

Source: *Unreferenced source.*

Operating System: *Information not available.*

Classification: Condition validation: check for limits.

19. Synopsis: The `ioctl` system call can be combined with `at` to take over a user's terminal and breach security.

    Source: Security distribution list.

    Operating System: SunOS 4.0

    Classification: Synchronization error: race condition.

20. Synopsis: If multiple users change their passwords simultaneously, the yellow pages password map file is updated concurrently with no file locking. This corrupts the yellow page map file.

    Source: Security distribution list.

    Operating System: SunOS 4.0.3

    Classification: Synchronization error: improper serialization of operations.

21. Synopsis: A race condition in `mkdir` exists that can be exploited to gain root access.

    Source: Security distribution list.

    Operating System: Most BSD derived systems.

    Classification: Synchronization error: race condition.

22. Synopsis: A vulnerability exists in the SITE EXEC feature of `ftpd` that allows any local or remote user to gain root access.

    Source: CERT advisory CA-94:08

    Operating System: Systems running wuarchive version 2.0-2.3

    Classification: Configuration error: utility installed with incorrect setup.

23. Synopsis: A vulnerability in the logging function of `xterm` allows local users to create new files or modify any existing files.

    Source: CERT advisory CA-93:15

    Operating System: Systems running X window system version 4, and version 5.

    Classification: Synchronization error: race condition

24. Synopsis: In SunOS 4.0 `/etc/utmp` is writable by default. This can be exploited by using programs such as `rwall` to overwrite the password file.

    Source: Security distribution list.

Operating System: SunOS 4.0

Classification: Condition validation error: input validation error

25. Synopsis: `yypasswd` file is world writable. This allows any user to edit and create new entries in the file.

    Source: Security distribution list.

    Operating System: SunOS 4.0.3

    Classification: Configuration error: secondary storage object (system database file) installed with incorrect permissions.

26. Synopsis: `tftpd` is installed on some machines that allows intruders access to the system.

    Source: CERT advisory CA-89:05

    Operating System: DEC/Ultrix 3.0

    Classification: Condition validation error: check for origin

27. Synopsis: Any user can create a setuid program by using a `-P` option in `login(1)` and breach security.

    Source: Security distribution list.

    Operating System: Ultrix 3.0

    Classification: Origin validation error: check for origin of subject

28. Synopsis: If `uucp` is mode 777, it can be exploited to gain root privileges.

    Source: Security distribution list.

    Operating System: *Information not available.*

    Classification: Configuration error: program installed with incorrect permissions.

29. Synopsis: If `rexd` (RPC Remote program execution) daemon is enabled, anyone on the Internet can get access to the machine running `rexd`.

    Source: CERT advisory CA-92:05

    Operating System: Aix 3.1, Aix 3.2

    Classification: Condition validation error: check for origin

30. Synopsis: If `sendmail` is installed with `-d` it can be exploited by unauthorized users to gain access to a system.

    Source: CERT advisory CA-94:12. Also see [Spa89].

    Operating System: Systems running sendmail version 8.6.7.

    Classification: Configuration error: program installed with incorrect setup parameters.

31. Synopsis: When installing DECnet Internet software, it is necessary to create a guest account on the host. By default this account has `/bin/csh` as its default shell. As the guest account has a valid shell, it can be exploited to gain root privileges.

    Source: CERT advisory CA-91:17

    Operating System: Ultrix 4.0,1,2

    Classification: Configuration error: utility installed with incorrect setup.

32. Synopsis: The queuing system on AIX includes a batch queue `bsh` that is turned on by default in `/etc/qconfig`. This can be exploited by remote and local users to gain root privileges.

    Source: CERT advisory CA-94:10

    Operating System: AIX version 3 and earlier

    Classification: Configuration error: utility installed with incorrect setup parameters.

33. Synopsis: A vulnerability in the performance tools in AIX allows local users to gain root privileges. The solution suggested is to remove the setuid bit.

    Source: CERT advisory CA-94:03

    Operating System: AIX 3.2.5

    Classification: Configuration error: secondary object (utility) installed with incorrect permissions.

34. Synopsis: If `fsck(8)` fails during system bootup, a privileged shell is run on the console. This can allow users with physical access to the system to gain unrestricted privileges.

    Source: CERT advisory CA-93:19

    Operating System: Solaris 2.x

    Classification: Condition validation: failure to handle exceptions

35. Synopsis: A user can create an interactive shell with the userid of a setuid shell script by creating a link to it with the name "-i".

    Source: *Unreferenced source.*

    Operating System: SunOS 3.2 and earlier

    Classification: Environment error: interaction between functionally correct modules

36. Synopsis: Any user could bypass authentication process by supplying a `-n` as an argument to the login program.

    Source: *Unreferenced source.*

Operating System: Sun 386i based systems.

Classification: Condition validation error: check for origin

37. Synopsis: `sendmail` could be exploited to read any file on the system regardless of the permissions by specifying it as the configuration file to `sendmail`.

    Source: See [Dis85].

    Operating System: BSD 4.2.

    Classification: Condition validation error: check for origin

38. Synopsis: A vulnerability exists on IRIX systems that can be exploited by users to gain root privileges. The vulnerability is exploited using `/usr/sbin/fmt`. The suggested solution is to change permission on `fmt` 755 and ownership to root.

    Source: CERT advisory CA-91:14

    Operating System: SGI IRIX systems.

    Classification: Configuration error: utility installed with incorrect permissions.

39. Synopsis: A writable setuid file exists in Masscomp RTU that can be exploited to gain root privileges.

    Source: Security distribution list

    Operating System: Masscomp RTU — a system V derived kernel.

    Classification: Configuration error: secondary storage object installed with incorrect permissions.

40. Synopsis: Mail programs on Unix systems change the uid of the mail file to that of the recipient. If the mail program does not clear the setuid bits of the file and the directory is writable security can be breached.

    Source: See [Bis86].

    Operating System: *Information not available.*

    Classification: Configuration error: secondary storage (mail directory) object installed with incorrect permissions.

41. Synopsis: The `restore` command was setuid which could be exploited to breach security.

    Source: CERT advisory CA-89:02.

    Operating System: SunOS 4.0.x

    Classification: Configuration error: utility installed with incorrect permissions.

42. Synopsis: Some systems were shipped with accounts that did not have passwords.

Source: CERT advisory CA-90:03

Operating System: Unisys U5000

Classification: Configuration error: utility set with incorrect setup parameters.

43. Synopsis: A vulnerability in `in.telnetd` allowed the possibility of capturing passwords by reading the terminal output.

    Source: CERT advisory CA-91:02a

    Operating System: SunOS 4.1 and SunOS 4.1.1

    Classification: Condition validation: check for access rights.

44. Synopsis: `/usr/bin/chroot` was improperly installed and could be exploited to gain root privileges.

    Source: CERT advisory CA-91:05

    Operating System: Ultrix 4.0,1

    Classification: Configuration error: utility installed with incorrect permissions.

45. Synopsis: Local users could gain unauthorized privileges if a setuid program changed its real and effective user ids to be the same and subsequently caused a dynamically-linked program to be exec'd.

    Source: CERT advisory CA-92:11

    Operating System: SunOS 4.0 and higher.

    Classification: Condition validation: check for origin.

46. Synopsis: Default permission on a number of files and directories were set incorrectly. For instance, files owned by root were owned by bin.

    Source: CERT advisory CA-93:03

    Operating System: SunOS 4.1, 4.1.{1,2,3}.

    Classification: Configuration error: secondary storage object has incorrect permissions.

47. Synopsis: As part of the installation, the system creates `/usr/release/bin` and installs two setuid root files: `makeinstall` and `winstall`. These could be exploited to gain root access.

    Source: CERT advisory CA-91:07

    Operating System: SunOS 4.1, SunOS 4.0.3, SunOS 4.1.1

    Classification: Configuration error: utility installed with incorrect permissions.

48. Synopsis: `/bin/login` is improperly installed and may be exploited to gain root privileges.

    Source: CERT advisory CA-91:08

Operating System: System V release 4.

Classification: Configuration error: utility installed with incorrect setup parameters.

49. Synopsis: In some SunOS version the root directory was writable. This could result in a user being able to overwrite or destroy sensitive files.

    Source: Security distribution list.

    Operating System SunOS 4.0

    Classification: Configuration error: secondary storage object installed with incorrect permissions.