A Classification of Software Vulnerabilities That Result From Incorrect Environmental Assumptions

Ivan Krsul and Eugene Spafford
Purdue University
West Lafayette, IN 47907-1398
{krsul,spaf}@cs.purdue.edu

Abstract

The consequences of a class of system failures, commonly known as software vulnerabilities, violate security policies. They can cause the loss of information and reduce the value or usefulness of the system.

An increased understanding of the nature of vulnerabilities, their manifestations, and the mechanisms that can be used to eliminate and prevent them can be achieved by the development of a unified definition of software vulnerabilities, and the development of a framework for the creation of taxonomies for vulnerabilities.

This paper provides a unifying definition of software vulnerability based on the notion that it is security policies that define what is allowable or desirable in a system. It also includes a framework for the development of classifications and taxonomies for software vulnerabilities.

This paper presents a classification of software vulnerabilities that focuses on the assumptions that programmers make regarding the environment in which their application will be executed and that frequently do not hold during the execution of the program.

1 Introduction

Software development can be complex. Added problem complexity, design complexity, or program complexity increases the difficulty that a programmer encounters in the design and coding of the software system [Conte et al. 1986]. Errors, faults, and failures are introduced in many stages of the software life-cycle [Beizer 1983; Myers 1979; DeMillo et al. 1987; Marick 1995]. The consequences of a class of system failures, commonly known as software vulnerabilities, violate security policies. They can cause the loss of information, and reduce the value or usefulness of the system [Leveson 1994; 1995; Amoroso 1994].

Since the publication of [Linde 1975], software researchers have developed various programming guidebooks for the development of secure software and analyzed in detail various vulnerabilities [Weissman 1995; Garfinkel and Spafford 1996; Gavin 1998; Bishop 1986; Smith 1994; CERT Coordination Center 1998c; Spafford 1989; Kumar et al. 1995; Bishop 1995; Schuba et al. 1997; Carlstead et al. 1975; Bibsey et al. 1975; Abbott et al. 1976]. However, vulnerabilities that are the result of the problems listed in these programming guides continue to appear [CERT Coordination Center 1998a; 1998b; Gavin 1998].

An increased understanding of the nature of vulnerabilities, their manifestations, and the mechanisms that can be used to eliminate them or prevent them can be achieved by the development of a unified definition of software vulnerabilities, the development of a framework for the creation of taxonomies for software vulnerabilities, and the classification of a representative collection of software vulnerabilities with this taxonomy.

An organizing framework can be used to generalize, abstract, and communicate findings within the research community. Taxonomies, or the theoretical study of classification, structure or organize the body of knowledge that constitutes a field. As such, they are an essential part of such a framework [Glass and Vessey 1995].

Researchers have attempted to develop such taxonomies and classifications for software vulnerabilities or related areas [Bishop 1995; Kumar and Spafford 1994; Kumar 1995; Kumar et al. 1995; Aslam 1995; Anderson 1994;
However, these classifications are ambiguous. The ambiguities are in part the result of conflicting definitions for software vulnerabilities, software faults, errors, etc.

A framework for the development of taxonomies according to generally accepted principles can be used to develop unambiguous classifications. These can result in an increased understanding of the nature of software vulnerabilities. An increased understanding of the nature of vulnerabilities can lead to improvements in the design and development of software.

2 Terminology and Notation

In this section we introduce and define some of terms that will be used through the paper. Related terms are grouped by areas.

2.1 Error, Faults, and Failures

An error is a mistake made by a developer. It might be a typographical error, a misreading of a specifications, a misunderstanding of what a subroutine does, and so on. An error might lead to one or more faults. Faults are located in the text of the program. More precisely, a fault is the difference between the incorrect program and the correct version. The execution of faulty code may lead to zero or more failures, where a failure is the [non-empty] difference between the results of the incorrect and correct program [IEEE 1990].

2.2 Computer Policy

The key concepts in existing policy definitions are value, authorization, access control, protection, and sensitivity of information [Garfinkel and Spafford 1996; Longley and Shain 1990; DoDCSEC 1983; Sterne et al. 1991; Dijker 1996; Kao and Chow 1995]. The policy definition in [Krusl et al. 1998] presents a definition of policy that takes these key concepts into account. This definition explicitly requires that the specification of the policy include a detailed account of when the system is considered to be valuable. From [Krusl et al. 1998], a Policy is the set of rules that define the acceptable value of a system as its state changes through time.

In operating systems such as UNIX and Windows NT, the security policies that can be enforced by the operating system are a subset of the policies that users and administrators expect applications and the system to enforce. Expected policies are the rules that the user expects the system and applications to enforce so as to maintain the value of the system as it changes through time.

For example, if a user runs a WWW browser he expects that it will not access and modify user files in directories other than those managed by the browser itself. Exceptions must be cleared with the user (the expected policy). The operating system, however, does not have any mechanisms for enforcing this user expectation, and the browser is free to read and modify any file that can be accessed by the user (the policy).

2.3 Software Vulnerability

Existing definitions of software vulnerability have one of three forms: Access Control, State-space, and Fuzzy [Denning 1983; Bishop and Bailey 1996; Longley and Shain 1990]. In these definitions, it is policies that define what is allowable or desirable in the system and hence the notion of computer vulnerability ultimately depends on our notion of policy.

The execution of a vulnerable software can violate the security policy, implied or explicitly specified. Software can be vulnerable because of an error in its specification, development, or configuration. A software vulnerability is an instance of an error in the specification, development, or configuration of software such that its execution can violate the security policy.

Note that the waterfall model of the software life cycle divides the development phase into design and coding [Conte et al. 1986]. Note also that an important class of errors in the development of a software system is the mismatch between the assumptions made during the development about the execution environment of the software, and the environment in which the program executes.
For example, as shown in Figure 1, and from the preceding definition, a software vulnerability can result from many errors, including errors in the specification, design, or coding of a system, or in environmental assumptions that do not hold at runtime. The following are examples of vulnerabilities for each of the categories shown:

- An example of a vulnerability that results from an error in the requirements or specification is the TCP Land vulnerability, where the TCP protocol specification has ambiguities and contradictions [Krsul et al. 1998].
- An example of a vulnerability that results from a design error is the TCP SYN Flood vulnerability, where the designer specifies that an inadequate number of buffers should be reserved for half-open connections [Schuba et al. 1997].
- An example of a vulnerability resulting from a coding error is the Java vulnerability where package membership is decided from the first component of the package name alone because a programmer delimited the package name with the first period in the full name, rather than the last period in the full name [McGraw and Felten 1997; Krsul et al. 1998].
- An example of a vulnerability that results from a mismatch between the assumptions the programmer makes about the environment in which the program will execute, and the environment in which the program actually executes, is the incorrect implementation of a system library.
- An example of a vulnerability that results from a configuration error is the vulnerability in which the NIS domain name is configured to be the same name as the DNS domain name [Krsul et al. 1998].
- An example of a vulnerability that results from an assumption made at the time of the requirement specification is a system that only allows uppercase letters in its eight byte password field. At the time of specification, it was assumed that no computer system would be capable of enumerating and trying all the possible password combinations.
2.4 Taxonomy and Classification

A taxonomy is the theoretical study of classification, including its bases, principles, procedures and rules. A classification is the separation or ordering of objects (or specimens) into classes. Classifications that are created non-empirically are called a priori classifications. Classifications that are created empirically by looking at the data are called a posteriori classifications [Simpson 1945; Grolier Incorporated 1993; Audi 1995; Simpson 1961; WEBOL 1998; EBRIT 1997].

2.5 Definitions of Other Terms

Other terms in this paper are used according to their definitions in [Spencer 1983; Longley and Shain 1990; WEBOL 1998; Longley and Shain 1990; Mockapetris 1987; Albittz and Liu 1992; Bhushan et al. 1971; Croosby et al. 1996; Postel 1981a; Borenstein 1992; Sun Microsystems Inc. 1989; Stern 1991; 1991; Comer 1984; Bach 1986; Wall and Schwartz 1990; Sun Microsystems Inc. 1988; Walsh 1994; Postel 1981b; 1980].

3 Classification Theory

Taxonomies increase our understanding of the world. A function of taxonomies is the separation or ordering of specimens so that generalizations can be made about them. Hence, we say that classifications have explanatory value. Taxonomies can also be used to predict the existence of specimens that have not been seen before by extrapolating from the known specimens. Hence, we say that taxonomies have predictive value.

The periodic table of the elements is an example of a taxonomy that has explanatory and predictive properties. It organizes the elements so that generalizations can be made in regards to groups of elements, and it predicted the existence of unknown elements before these were discovered [Bahr and Johnston 1995].

Taxonomies also establish organizing frameworks, essential for the development of a field. “Without an organizing framework, researchers and practitioners find it hard to generalize, communicate, and apply research findings. Taxonomies structure or organize the body of knowledge that constitutes a field, with all the potential advantages that brings for the advancement of the field.” [Glass and Vessey 1995].

The existence of taxonomies and classifications in computer science and related fields—for example see [Cohen 1997b; 1997a; DeMillo and Mathur 1995; Duda and Hart 1973; Kumar 1993; Olivier and Vonsolms 1994; Glass and Vessey 1995; Bier et al. 1995; Roskos et al. 1990; Young and Taylor 1991; Bishop 1995; Landwher et al. 1993; Aslam et al. 1996; Oman and Cook 1991; 1990; Kumar et al. 1993; Aslam 1995]—is an indication that computer scientists agree with the statements made in [Simpson 1945; Glass and Vessey 1995]. Many of these taxonomies or classifications, however, do not satisfy the predictive and descriptive properties desirable because they do not adhere to the fundamentals of the development of taxonomies. Hence, their contribution to our understanding of the field, as is suggested in the preceding quote, is limited.

This section presents an overview of the fundamentals of taxonomies to provide the necessary background for the development of better taxonomies for the field of computer security. In particular we focus on the classification of vulnerabilities. The concepts presented in this section, however, can be applied to other areas in computer security and computer science.

3.1 Taxonomic Characters, Object Attributes or Features

The basis for the development of successful classifications are taxonomic characters. These are the properties or characteristics of the objects that will be classified. Taxonomic characters are also commonly called features, attributes or characteristics. Taxonomic characteristics must satisfy the following properties [Simpson 1961; Glass and Vessey 1995; Krsul 1998]:

Objectivity: The features must be identified from the object known and not from the subject knowing. The attribute being measured should be clearly observable.

Determinism: There must be a clear procedure that can be followed to extract the feature.
Repeatability: Several people independently extracting the same feature for the object must agree on the value observed.

Specificity: The value for the feature must be unique and unambiguous.

If any of these characteristics is not met then the classification cannot be repeated, leads to controversy, or is misleading. We illustrate this with an example from the field of computer security:

The classification in [Howard 1997] acknowledges explicitly the need for the deterministic and specificity principles: “A taxonomy should have classification categories with the following characteristics: 1) Mutually exclusive ... 3) Unambiguous ...” In the definition of the possible values for the tool level in the classification, [Howard 1997] describes an Autonomous Agent as “... a program, or program fragment which operates independently from the user to exploit vulnerabilities”, and defines a Toolkit as “... a software package which contains scripts, programs or autonomous agents that exploit vulnerabilities.” Hence, the specificity principle cannot be fulfilled because the autonomous agents and toolkits are not mutually exclusive.

The first dimension in [Howard 1997] is “Attackers.” However, the same document observes that only 0.8% of the network attacks the CERT observed (the attacks that were to be classified) had this information available. This is the first level in the classification tree but the value is not measurable for 99.2% of the data. Hence, the feature is not observable.

Although the example presented fails to satisfy the requirements specified, it enumerates some of the taxonomic characteristics used for classification. Other taxonomies and classifications in the field of computer science fail to present even these and simply enumerate a series of elements grouped into categories and call this grouping a taxonomy or classification; examples include [Roekos et al. 1990; Cohen 1997a; 1997b].

### 3.2 Taxonomies and Classifications

Section 3 mentions that a classification is an ordering of objects into groups that have explanatory and predictive value. [Simpson 1945] defines taxonomy as “... the theoretical study of classification, including its bases, principles, procedures and rules”. Although there are variations on this definition, (e.g. [Grolier Incorporated 1993; EBRIT 1997; WEBOL 1998; OXFORD 1998]) they agree that a taxonomy includes the theory of classification, including the procedures that must be followed to create classifications, and the procedures that must be followed to assign objects to classes.

However, [Krsul 1998] shows that computer security practitioners often confuse the terms classification and taxonomy. For example, the alleged classification presented in [Cohen 1997a] is neither a classification nor a classification scheme because it does not provide the procedures that must be used to assign individual attacks to the numerous classes enumerated in the paper. Other examples are given in [Krsul 1998].

### 3.3 Types of Classifications

A classification is the separation or ordering of objects (or specimens) into classes. Classifications can be generated a priori (i.e. non-empirically from an abstract model) or a posteriori (empirically by looking at the data).

With a set of taxonomic characters that satisfy the criteria mentioned in Section 3.1, classification schemes that can be built include arbitrary selections, decision trees, natural classifications, evolutionary classifications, and natural clusterings.

Arbitrary selections are groupings of individuals on a single characteristic. These are the simplest classification schemes and require that individuals be grouped according to a simple selection criteria. For example, grouping programs by their programming language, by their use or non-use of cryptography, etc.

Classification by a decision tree is the process of answering a series of questions to walk down a decision tree until the individual is classified by reaching a leaf in the tree. Decisions trees can be generated a priori or a posteriori.

Classifications that use decision trees avoid the issue of ambiguity because by answering the questions presented the individual always reaches a node. In practice, however, it may be possible to create decision trees that are ambiguous, especially if the selection criteria in each of the internal nodes of the tree have more than one fundamentum divisionis [Simpson 1945].
A fundamentum divisionis is a term from Scholastic Logic and Ontology that means “grounds for a distinction” [Audi 1995]. Ambiguities arise when the selection criteria for an internal node of the tree has more than one fundamentum divisionis. For example, a single node in the classification of a vulnerability may ask the question “is the vulnerability a race condition, or a configuration problem?” Vulnerabilities may appear that are both a race condition and a configuration error.

A natural classification groups together individuals that seem to be fundamentally related [EBRIT 1997; Simpson 1945]. This type of classification is useful when the taxonomic characteristics themselves reveal natural groups. Features of computer programs are well suited for natural classifications, as specialized measurements can only be made once a required characteristic has been identified.

Natural clusterings group individuals together because they share the largest number of characteristics possible without regards to the reasons these individuals share the characteristics. There are many algorithms that can be used to find natural clusters given a data set (see [Jain and Dubes 1988] for an introduction). Figure 2 illustrates the process of clustering using volumes in three dimension, linear separations in two dimensions or grouping by number of shared characteristics. Note that clustering algorithms such as these can result in ambiguous classifications but are well suited for discovering relationships that may not be readily apparent.

![Ambiguous Classification Diagram](image)

Figure 2. Natural clusterings group individuals together because they have similar characteristics. Note that classification can be ambiguous if an individual has the same number of similar characteristics with more than one group.
Several projects have dealt with the issue of identifying and classifying software faults, including the Protection Analysis (PA) Project which conducted research on protection errors in operating systems [Carlstead et al. 1975; Bilsey et al. 1975]; the RISOS project that was aimed at understanding security problems in existing operating systems and to suggest ways to enhance their security [Abbott et al. 1976; Landwher et al. 1993] lists a collection of security flaws in different operating systems and classifies 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; and [Marick 1990] presents a survey of software fault studies from the software engineering literature (the studies reported faults that were discovered in production quality software).

Some of these projects developed classifications in relation to software faults and vulnerabilities. However, they do not adequately meet the requirements that were specified in Section 3.1. Hence these classifications are ambiguous and of limited predictive and explanatory value. Note that the limitations of some of these classifications are mostly the result of conflicting definitions of software vulnerability or software fault.

### 4.1 Aslam Classification

[Aslam 1995] develops a classification scheme that can aid in the understanding of software faults that can subvert security mechanisms. This classification scheme divides software faults into two broad categories: Coding Faults that result from errors in programming logic, missing requirements, or design errors; and Emergent Faults resulting from improper installation or administration of software so that faults are present even if the software is functioning according to specifications. [Bishop and Bailey 1996] shows that this classification does not satisfy the specificity requirement as it is possible to classify a fault in more than one classification categories.

### 4.2 Knuth Classification

Donald Knuth, author of the TeX typesetting system kept a detailed log of all the bugs and faults fixed in TeX for a period of over ten years and developed a detailed classification of types of faults found in his system [Knuth 1989]. [Krsul 1998] shows that this classification is subjective and ambiguous, and it is clear that it is difficult to apply the Knuth classification scheme to errors of programs where the person performing the classification is not the original programmer.

### 4.3 Grammar-based Classification

[DeMillo and Mathur 1995] presents a grammar-based fault classification scheme that takes into account that syntax is the carrier of semantics. Any error of a program manifests itself as a syntactic aberration in the code. The classification is based on the operations that need to be performed to correct the fault. In this classification scheme, the classification of faults is based on the modifications that must be performed to fix the fault using syntactic transformer functions.

Because there are many ways to fix a fault, there can be many different classifications depending on which fix is used to eliminate the fault. Hence, the fault classification is not unique until a definitive and unique fix to a fault is selected. When a vulnerability is found the fix is not unique and frequently the details of the fix, when such a fix exists, are not released to the public. Hence, this classification fails the specificity principle until a definitive fix has been chosen.

### 4.4 Endres Classification

This classification was developed in [Endres 1975] as an analysis of errors in system programs. The fault classification scheme of Endres is application and machine dependent and hence does not apply to the classification of vulnerabilities in general systems [DeMillo and Mathur 1995; Endres 1975].
4.5 Ostrand and Weyuker Classification


4.6 Basili and Perricone Classification

This classification was defined by Basili and Perricone in [Basili and Perricone 1984]. [DeMillo and Mathur 1995] shows that it is ambiguous.

4.7 Origin and Causes Classification

This classification was originally defined in [Longstaff 1997] to identify the origins of vulnerabilities. As shown in [Krsul 1998], this classification is difficult to use without detailed information about the state of mind of the programmer during the development process.

4.8 Access Required Classification

This classification was originally defined in [Longstaff 1997] and defines the access that is required to exploit the vulnerability. As shown in [Krsul 1998], there is no clear definition for each of the categories for the classification.

4.9 Category Classification

This classification identifies the system component to which a vulnerability belongs and is common in vulnerability databases as described in [Krsul 1998]. The notion of an application, system utility, etc., varies among operating system types. Micro-kernels, object oriented operating systems, and distributed systems have different views of what constitutes a system utility, or user-level application [Dasgupta et al.; Tanenbaum 1987].

4.10 Ease of Exploit Classification

This classification was originally defined in [Longstaff 1997] and identifies the difficulty of exploiting a vulnerability. The taxonomist can not know if exploit scripts or toolkits are available, and these may appear after the taxonomist has chosen the value for the classification. The value of this classification is time dependent and the classification should take this into account explicitly.

4.11 Impact Classification

This classification identifies the impact of the vulnerability. It is used to define both direct and indirect impacts. Direct impacts are those that are felt immediately after the vulnerability is exploited and indirect impact are those that ultimately result from the exploitation of the vulnerability. This classification is common in vulnerability databases as described in [Krsul 1998]. This classification is a decision tree of depth one that, as shown in [Krsul 1998], has more than one fundamentum divisionis.

4.12 Threat Classification

This classification of the threat that vulnerabilities create was extracted from [Power 1996]. It is attributed to Donn Parker of SRI International as a classification of hostile actions that your adversary could take against you. As shown in [Krsul 1998], classification trees should use decision nodes that use one fundamentum divisionis and, the threat classification does not follow this principle.

Also, this classification does not specify an explicit maximum level of indirection that can be used for the determination of the threat of a vulnerability. Hence, a vulnerability that causes the encrypted password of a user to be displayed is also a threat to integrity if the password is decrypted, the user account is compromised, an
encrypted administrator password can be obtained using this account, this last password can be decrypted, and a root shell can be obtained. Because root shells allow any operation to proceed, integrity can be violated. A similar reasoning can be applied to most of the Unix vulnerabilities.

4.13 Complexity of Exploit Classification

This classification identifies the complexity of the exploitation of a vulnerability, regardless of whether a script or toolkit exists for the exploitation of the vulnerability. This classification is common in vulnerability databases as described in [Krsul 1998]. This classification is subjective as there is no accepted definition of “simple sequence of commands,” “complex set or large number of commands,” etc.

4.14 Cohen’s Attack Classification

This classification is a list of one hundred attacks possible on a system listed in [Cohen 1997a; 1995].
[Cohen 1997a; 1995] notes that this classification is descriptive, non-orthogonal, incomplete, and of limited applicability. And indeed, many of the classes are ambiguous, and dependent on attributes that are not measurable.
This classification mixes floods and volcanoes, trojan horses and viruses, dumpster diving, bribes and extortion, as well as invalid values on system calls and race conditions. Some of these attacks are environmental conditions that can result in damage, some are techniques to manipulate the human component of a system, and some are code faults that may or may not result in a vulnerability.

4.15 Perry and Wallich Attack Classification

This is a matrix-based classification scheme in two dimensions: Potential perpetrators and potential effects [Perry and Wallich 1984]. As shown in [Krsul 1998], this classification fails the specificity principle.

4.16 Howard Process-based Taxonomy of Network Attacks

[Howard 1997] proposes a classification of computer and network attacks that identifies the process that “links” attackers to their ultimate objectives. The classification can be represented as a classification tree that has multiple levels, and for which at each level a choice must be made between a series of values. As shown in [Krsul 1998], the specificity principle cannot be fulfilled and one of the features for the classification is not observable.

5 A Taxonomy for Software Vulnerabilities

Modern computer systems are built from interrelated subsystems, and each of these subsystems can have considerable complexity. A modern operating system, such as Linux, is composed of hundreds of subsystems and each can have thousands or tens of thousands of lines of code. The security of the system depends on the interaction between these complex subsystems as well as on the behavior of the components themselves.

The developer of each of these subsystems makes a series of assumptions about the behavior of the other subsystems, implicitly or explicitly. If the security of the system depends on these assumptions, then their violation can result in critical failures, and sometimes these failures belong to the category of failures we call vulnerabilities because they violate the security policy for that system. [Brooks 1995] makes a note of this “The most pernicious and subtle bugs are system bugs arising from mismatched assumptions made by the authors of various components.”

Programmers and designers must make assumptions about the environment in which their programs will execute. For example, the Bell and LaPadula model for information flow makes the assumption that the security level of an active object cannot change [Bell and LaPadula 1973; Denning 1983]. This assumption is called the tranquility assumption and without it, it is not possible to enforce the model proposed. In systems that are not fault-tolerant, such as Unix or Windows NT, programmers must at least assume that the hardware of the system that will execute the code is correct and that the execution of the instructions is deterministic and well defined.

Programmers in high-level languages such as C, C++, or Java, and in operating systems such as Unix or Windows NT, make implicit assumptions about their environment. Software testing strategies and compiler support
tools, such as Lint, Purify, and Insure++, attempt to perform checks that users often do not perform [Myers 1979; DcMilo et al. 1987; Beizer 1983; Kolawa and Hickin 1997].

There are several classes of assumptions that programmers can make about the environment in which their programs will execute. A class of these assumptions regard the correctness of the implementation of the primitives offered by the programming language or the operating system. These assumptions are axiomatic and their violation does not entail a vulnerability in the program but in the operating system or the compiler itself.

A second class of assumptions that programmers can make—about the environment in which their programs will execute—cannot be represented by a decidable function that can be evaluated at the time the program is executed. These assumptions may be undecidable because the language that describes the assumption is not recursive (i.e. there is no algorithm that takes as an input an instance of the environment to determine if the assumption holds or not), or because the assumption is subjective.

A third class of assumptions can be expressed by a decidable algorithm where the objects of the environment are data types for the language used in this specification. These assumptions are encoded so that the algorithms can be evaluated to determine if the assumption holds at any given time, and the algorithm must result in a yes/no answer deterministically.

We can further divide these assumptions into two categories: the first where the programmer cannot verify the assumption within the program because the primitives provided within the language are not sufficiently expressive (but where the compiler or interpreter could). The second where the programmer could verify the assumption if he added the necessary code to the program. If programmers add checks to programs to verify that these assumptions hold at runtime, these checks usually add complexity and size to programs, making it harder to develop or maintain existing programs. Hence, in practice programmers often do not perform them. Also, programmers often are not aware that the assumptions they are making may not hold at runtime.

The classification presented in this section is oriented towards the identification of the assumptions that programmers make about their environment, and whose violation results in software vulnerabilities.

This classification initially divides vulnerabilities into four hierarchical classes, only one of which will be expanded further. The classes are Design Flaws, Environmental Flaws, Coding Flaws, and Configuration Flaws. As shown in Figure 3, there exists a decision tree that is used to eliminate any ambiguities from this first step in the classification. The four questions for this decision tree are as follows:

**Q1:** Is the vulnerability the result of a flaw in the design of the software? Did the designer misunderstand the requirements? Did the designer of the software assume that the environment in which the program was going to run had different characteristics than those of the actual environment?

**Q2:** [The designer made correct assumptions about the environment and requirements of the program] Is the vulnerability the result of the implementer making a simplifying assumption about the environment in which the program was going to be run, and if this assumption were to be true the vulnerability would not exist?

**Q3:** [Both the designer and the programmer made correct assumptions about the environment, and the designer understood the requirements of the program] Is the vulnerability a result of software faults or programming errors?

**Q4:** [Both the designer and the programmer made correct assumptions about the environment, and the designer understood the requirements of the program, and the program is (or appears to be) correctly implemented] Is the vulnerability in that the program was installed with improper configuration parameters, and correcting these would remove the vulnerability?

It is possible that the taxonomist might not be able to answer one of these four questions because of insufficient information. As shown in Figure 3, vulnerabilities that cannot be classified because of this reason are tagged as unknown.

The classes shown in Figure 3 correspond to groups of vulnerabilities as defined in Section 2.3. As shown in Figure 4, Class 1 vulnerabilities correspond to those in the area marked with the symbol ▽. Class 3 vulnerabilities correspond to those in the area marked with the symbol ★. Class 4 vulnerabilities correspond to those in the area marked with the symbol ♦. The classification described in this sections corresponds to the vulnerabilities in the area marked with the symbol ▲.
Figure 3. A classification for the identification of environmental assumptions made by programmers—Part 1.

There are an infinite number of assumptions that a user can make about the environment in which a program executes, and exhaustively listing these assumptions is not possible. However, as was shown in [Krsul 1998], there are a small number of assumptions commonly made by programmers that are responsible for a significant fraction of the known vulnerabilities. Hence, a significant number of vulnerabilities we know of could be prevented if either compiler support was provided to enforce these assumptions by default—and programmers were able to specify the assumptions they are making as they develop their programs—or if the operating system could provide a virtual execution environment that enforces these assumptions at runtime.

Assumptions made by programmers can be described by an algorithm and a list of object attributes (that the algorithm operates on) as an $n$ tuple $< o_1, o_2, \ldots, o_{n-1}, \text{algorithm}>$. In this section, the algorithm is referred to as an attribute constraint.

As shown in Figure 5, the environmental assumptions class can be subdivided by branching three or more times. The fundamentum divisionis used by the branches are Environment Object, Object Attributes, and Attribute Constraint.

In a computer system, an environment object is an entity that contains or receives information, that has a unique name, and that has a set of operations that can be carried out on it [Longley and Shain 1990; Tanenbaum 1987]. An attribute of an object is a data component of an object. A derived attribute of another attribute is a data component of the later attribute. A property of an attribute is a characteristic of the attribute that can be derived from the attribute by the application of a function to the attribute.

An attribute refinement is a finite refinement of attributes within attributes, and results in the identification of the attributes about which assumptions are made. The attribute refinement cannot contain a property of an attribute.

The Attribute Constraint identifies the property or set of properties that are being assumed about that particular attribute.

Attributes can be derived from other attributes. Attribute refinement on sets do not require that all attributes in the set be refined. An attribute refinement in a set of attributes is possible if at least one of the attributes in
the set can be refined to specify the attributes that are sufficient to specify the constraints in the assumption.

Sets of attributes are used when a single assumption made about the environment requires multiple attributes, possibly from multiple objects. They should not be used to specify more than one assumption.

To summarize, a class in this classification requires the specification of an object, or set of objects, an attribute expansion (that results in an attribute or set of attributes), and a constraint specification that defines the assumption made about the environment.

Figures 7 and 8 show part of an instantiation of the classification tree. In these figures the final classes (or the leaves of the tree) are indicated by underlined text, and classes in italics represent those classes for which we have no evidence that a vulnerability exists (i.e. these represent predicted classes).\(^1\)

Without loss of generality, the instantiation presented in this section is specific to the vulnerabilities in the database described in [Krusl 1998], and is specific to UNIX, Windows NT, and Java. Objects, or sets of objects, specific to other operating systems can be added to the class as vulnerabilities for those systems are collected.

The constraint specifications for some of the environmental assumptions made by programmers can be complex and cannot be shown in the figures.

The classification presented in this section requires detailed information about the system and the design and development stages of the software systems. Otherwise there are cases in which it is difficult to answer the questions in the first part of the classification (Q1, Q2, Q3, and Q4).

The taxonomy described in this section was applied to the vulnerabilities in the vulnerability database described in [Krusl 1998]. 32% of the vulnerabilities classified did not have detailed enough information in the database to determine the appropriate class in the classification. Hence, these were excluded from our analysis. Figure 6 shows the distribution of vulnerabilities classified with the taxonomy presented in this section.

The classification presented in this section has both predictive and descriptive properties. Each class in the classification describes an environmental assumption made by programmers that results in a vulnerability, and

\(^1\)The complete classification tree is shown in [Krusl 1998]
Figure 5. A classification for the identification of environmental assumptions made by programmers—Part 2.

The vulnerability could be prevented or eliminated by enforcing the environmental assumption with a specialized compiler or by running the program in a special virtual environment that can enforce the assumptions. Hence, the classification has descriptive value.

The classification tree was built a priori and confirmed with the classification of 90 vulnerabilities of the 210 in the database. Some classes of vulnerabilities were predicted from these samples by extrapolating from existing classes, and the remaining vulnerabilities were classified. The classification has predictive value because it allows the prediction of vulnerabilities that we have not seen before and that are the result of programmers making assumptions that can be extrapolated from existing classes.

5.0.1 Scope of the Taxonomy

The taxonomy presented in this section can be applied to any system where environmental assumptions can be specified as constraints on attributes of objects (or other attributes). Such systems include modern operating systems such as UNIX, Windows NT, and Macintosh MacOS, as well as object oriented operating systems, distributed operating systems, and micro kernels. The classification is extensible and assumptions about specialized objects are possible by creating a new second level node (object) that corresponds to the specialized object.

The classification is suitable for representing complex environmental assumptions that take into account multiple objects and multiple attributes by using sets of objects and attributes. Furthermore, the classification was designed so that a one-to-one mapping exists between the environmental assumption and a formal policy specified with the model presented in [Krsul et al. 1998].
Figure 6. Distribution of vulnerabilities classified with the taxonomy presented in this section.

5.0.2 Application of the Taxonomy of Software Vulnerabilities

The taxonomy of software vulnerabilities presented in Section 5 identifies the environmental assumptions that programmers make about the environment in which their applications will run, and whose violation can result in vulnerabilities.

In this section we argue that the vulnerabilities resulting from the violation of these assumptions can be prevented by the design of a domain-specific compiler, or by the design of a virtual environment that can enforce these assumptions at runtime.

Assume that programming languages, and in particular the C programming language, can be extended by adding compiler pragmas or processor directives as described in [Kernighan and Ritchie 1988], incorporating application semantics as described in [Engler 1998], or by extending the language by adding a special aspect to the language as defined in [Kiczales et al. 1997].

For each of the classes in Section 5, the compiler can be extended to enforce the constraint on the attribute of the object specified for the class.

Note that this section suggests a domain-specific tool that could be theoretically used to enforce the environmental constraints. At this point there is no experimental evidence that these modifications can be implemented as described in this section, and that programmers would remember or desire to add the directives, pragmas, or aspects.

We also argue that a special execution environment can be provided if the constraints for the attributes and objects are formally specified using the policy specification language described in [Krsul et al. 1998], and the policy violation tool described in that paper can be implemented.
Figure 7. Taxonomy of Software Vulnerabilities Top Level
Figure 8. Taxonomy of Software Vulnerabilities, Levels 2-1 and 2-2
Virtually every field where failure can be catastrophic has recognized that accumulation of information about failures is critical to the stepwise refinement of technology, particularly when the systems that fail are highly complex:

When an aircraft crashes, it is front page news. Teams of investigators rush to the scene, and the subsequent enquiries are conducted by experts from organisations with a wide range of interests—the carrier, the insurer, the manufacturer, the airline pilots' union, and the local aviation authority. Their findings are examined by journalists and politicians, discussed in pilots' messes, and passed on by flying instructors. In short, the flying community has a very strong and institutionalised learning mechanism. This is the main reason why, despite the inherent hazards of flying in large aircraft, which are maintained and piloted by fallible human beings, at hundreds of miles an hour through congested airspace, in bad weather and at night, the risk of being killed on an air journey is only about one in a million. [Anderson 1994]

Other sources, including [Schlager 1994] and [Perrow 1984] make it clear that prompt and complete information dissemination is critical if we want to learn from past mistakes. More often than not, it is not the designers of the systems that find and debug complex systems but observers who find patterns that lead to the cause of failures.

Scientists and engineers who are responsible for the development of critical systems are used to the idea of learning from past mistakes. [Levy and Salvadori 1992] describes in great detail some of the more spectacular structural failures in history and provides evidence that structural failures are likely to become less common because of the application of the knowledge gathered in the examination of past failures to modern designs. Similar arguments can be made in the design of any complex system that is difficult to design and implement [Petrosky 1985; Brooks 1995; Schlager 1994; Perrow 1984; Dorner 1996; Anderson 1994].

Collected past knowledge, however, must be part of a framework that can be used to generalize, abstract, and communicate findings within the research community. Taxonomies and classifications structure or organize the body of knowledge that constitutes a field [Glass and Vessey 1995].

This paper provides a scientific framework for the development of such taxonomies and classifications, and an extensible environment that can be used to identify the nature of software vulnerabilities. Based on this framework we collected a representative sample of software vulnerabilities with detailed information that contributes to our understanding of software vulnerabilities. The need for such increase in understanding of the nature of vulnerabilities is argued in [Leveson 1994] as follows:

[In Software Engineering] Our greatest need now, in terms of future progress rather than short-term coping with current software engineering projects, is not for new languages or tools to implement our inventions but more in-depth understanding of whether our inventions are effective and why or why not.

The collection of vulnerabilities is a detailed record of their sources, causes, and effects. This record contributes to the development of the field because other scientists can learn from past mistakes, and provides an environment that can be used to develop a more in-depth understanding of vulnerabilities. Other researchers, for example, are using this environment and the data collected to develop new software testing techniques, new comprehensive definitions of network vulnerabilities, etc. [Daniels et al. 1998a; 1998b; Daniels 1998; Du and Mathur 1998].

The application of the framework for the development of classifications—presented in Section 3—to the data collected for the vulnerability database resulted in the classification for software vulnerabilities presented in Section 5.

This classification provided insights as to the nature of software vulnerabilities that were not evident a priori. In particular, we have shown that 63% of the vulnerabilities from the database space are not the result of traditional software faults, but rather the result of incorrect assumptions made by programmers regarding the environment in which the systems will run.

In Section 5.0.2, we argue that these mistaken assumptions can be enforced or guaranteed if we develop domain-specific tools, and these tools result from an increased understanding of the nature of vulnerabilities. The application of the framework we presented in this paper provides the desired increase in our understanding of vulnerabilities.
6.1 Future Work

We believe that the data in the database is a representative sample of the vulnerabilities known. However, there is a continuous stream of new vulnerabilities being reported in mailing lists, and these must be incorporated into the database if the framework and environment provided is to be useful on a continuous basis.

Many computer systems—for example Windows NT, CISCO routers, HP-UX UNIX, and IBM AIX UNIX, etc.—are closed and source code was unavailable during the collection of data for the database. The number of fields that can be filled for a record in the vulnerability database is proportional to the information available for the system. Source code for such systems would contribute substantially to the quality of the sample collected, and hence could improve on the analysis and conclusions derived from it.

The classification presented in 5 focuses on the assumptions that programmers make regarding the environment in which their application will execute, and that frequently do not hold in the execution of the program. This classification can be extended to consider in more detail those vulnerabilities that result from configuration errors and those resulting from design errors. The application of the framework presented in this paper to those areas could increase our understanding of the nature of these vulnerabilities, the fundamental reasons for their prevalence, and result in better design or deployment strategies that can eliminate these problems.

References


CERT Coordination Center. 1998a. CERT Summary CS-98.03. ftp://ftp.cert.org/pub/cert_summaries/CS-98.03.


19


Dorner, D. 1996. The logic of failure: why things go wrong and what we can do to make them right. Metropolitan Books.


Endres, A. 1975. An Analysis of Errors and Their Causes in System Programs. IEEE Transactions on Software Engineering SE-1, 2 (June), 140-149.


Krsul, I., Daniels, T., and Wilson, A. 1998. COAST Vulnerability Database. Available by request from the COAST Laboratory.


