SOFTWARE VULNERABILITY ANALYSIS

A Thesis

Submitted to the Faculty

of

Purdue University

by

Ivan Victor Krsul

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

May 1998

To Jacqueline M. Martinez sensei, for giving me the Dojo Kun, and to my family for giving me the life needed to understand it.

ACKNOWLEDGMENTS

There are many people that contributed significantly to my work and I would like to acknowledge their contributions.

Thanks to the members of my examination committee, Eugene Spafford, Aditya Mathur, Matthew Bishop, Carla Brodley, and Antony Hosking for their valuable suggestions and support. Carla Brodley's guidance in the fields of machine learning and data analysis, and her support with the tools and equipment needed for part of the analysis, is gratefully acknowledged.

I would like to thank Diego Zamboni, Tom Daniels, David Isacoff, Chapman Flack, and Eugene Spafford for their help with the reviews of this dissertation. Thanks to Diego Zamboni, Tom Daniels, Kevin Du, Mahesh Tripunitara, and Tugkan Tuglular for all their help and valuable suggestions during the vulnerability database weekly meetings. A significant portion of the ideas for this dissertation originated during our discussions there.

Tom Daniels and Adam Wilson helped enter data into the vulnerability database. I am grateful for their contributions. Thanks to Tugkan Tuglular for helping me develop most of the work relating to policies and the definitions of software vulnerability. Thanks for Edward Felten of Princeton University, and Michael Dilger for their contributions on software vulnerabilities. Thanks to Jeff Bradford for his help with the mineset classification tools.

Thanks to David Isacoff and Mahesh Tripunitara for their contributions on the nature of vulnerabilities and for clarifying many doubts with regards to the taxonomy of software vulnerabilities developed in section 6.1.

Portions of this work were supported by contract MDA904-97-6-0176 from the Maryland Procurement Office, and by the various sponsors of the COAST Laboratory — support that is gratefully acknowledged.

Diego Zamboni, Tugkan Tuglular, Tom Daniels and Mahesh Tripunitara deserve special recognition for tolerating my personality changes during the last few weeks of writing.

Last, but not least, my friends, Susana Soriano, Maria Khan, Elli Liassidou, Engin Uyan, Sidem Yavrucu, Pelin Aksit, Denis Lekic, Carlos Gonzales, Claudia Fajardo, Danielle Bolduc, Colin Brammer, Robert Ferguson, Carlos Ortiz, Charles Meyer, Juli Phillips, Pat Randolph, Lisa Anderson, and many others that I wish I could name, contributed significantly to my mental and spiritual well-being during the writing of this dissertation. Words are not sufficient to *thank* them.

TABLE OF CONTENTS

Page

LIST OF TABLES x

LIST OF FIGURES xi

ABSTRACT xiv

1 INTRODUCTION 1

1.1 Problem Statement 1

1.2 Thesis Statement 1

1.3 Contributions 2

1.4 Organization of the Dissertation 3

2 NOTATION AND TERMINOLOGY 5

2.1 Terminology 5

2.1.1 Error, Faults, and Failures 5

2.1.2 Computer Policy 5

2.1.3 Software Vulnerability 6

2.1.4 Taxonomy and Classification 12

2.1.5 System Objects, Attributes and Constraints 12

2.1.6 Definitions of Other Terms 13

2.2 Notation 13

3 RELATED WORK 16

3.1 Classification Theory 16

3.1.1 Historical Background 17

3.1.2 Taxonomic Characters, Object Attributes or Features 18

Page

3.1.3 Taxonomies and Classifications 21
3.1.4 Types of Classifications 22
3.2 Prior Software Vulnerability Classifications 25
3.2.1 Aslam Classification 27
3.2.2 Knuth Classification 27
3.2.3 Grammar-based Classification 29
3.2.4 Endres Classification 30
3.2.5 Ostrand and Weyuker Classification 30
3.2.6 Basili and Perricone Classification 30
3.2.7 Origin and Causes Classification 31
3.2.8 Access Required Classification 31
3.2.9 Category Classification 32
3.2.10 Ease of Exploit Classification 32
3.2.11 Impact Classification 32
3.2.12 Threat Classification 33
3.2.13 Complexity of Exploit Classification 34
3.2.14 Cohen's Attack Classification 34
3.2.15 Perry and Wallich Attack Classification 35
3.2.16 Howard Process-based Taxonomy of Network Attacks 36
3.2.17 Dodson's Classification Scheme 36
3.3 Vulnerability Databases 36

4 DEVELOPMENT OF NEW TAXONOMIC CHARACTERS 39
4.1 Threat Features 39
4.2 Environmental Assumption Features 40
4.3 Features on the Nature of Vulnerabilities 42
4.3.1 Objects Affected 42
4.3.2 Effect on Objects 43
4.3.3 Method or Mechanism Used 44
4.3.4 Input Type 45

Page

4.4 Chapter Summary 45

5 EXPERIMENTAL ANALYSIS OF SOFTWARE VULNERABILITIES 47

5.1 Experiment Hypothesis 48

5.2 Experimental Setup 48

5.2.1 Sources for Data Collection 49

5.2.2 Database Structure 50

5.2.3 Data Characteristics 55

5.2.4 Data Distribution 55

5.3 Experiments 66

5.3.1 Co-word Analysis 66

5.3.2 Induction of Decision Trees 86

5.3.3 Data Visualization Tools 89

5.4 Chapter Summary 90

6 *A PRIORI* CLASSIFICATIONS OF SOFTWARE VULNERABILITIES 93

6.1 A Taxonomy for Software Vulnerabilities 93

6.1.1 Scope of the Taxonomy 112

6.1.2 Application of the Taxonomy of Software Vulnerabilities 112

6.1.3 Formalization of the Taxonomy of Vulnerabilities 113

6.2 Evolutionary Classification 119

6.3 A Classification for Software Testing 120

6.4 Chapter Summary 120

7 SUMMARY, CONCLUSIONS, AND FUTURE DIRECTIONS 122

7.1 Conclusions 122

7.2 Summary of Main Contributions 124

7.3 Future Work 125

BIBLIOGRAPHY 126

APPENDICES 136

Page

A SCHEMAS FOR PRIOR VULNERABILITY DATABASES 137
A.1 Vulnerabilty Databse at ISS 137
A.2 Vulnerabilty Databse at INFILSEC 137
A.3 Vulnerabilty Databse of Michael Dilger 138
A.4 Eric Miller's Database 141
A.5 The CMET Database at the AFIW 141
A.6 Mike Neuman's Database 146

B VULNERABILITY CLASSIFICATIONS - DETAILED LIST 148
B.1 Aslam Classification 148
B.2 Knuth Classification 150
B.3 Grammar-based Classification 150
B.4 Endres Classification 151
B.5 Ostrand and Weyuker's Classification 152
B.6 Basili and Perricone Classification 152
B.7 Origin and causes 153
B.8 Access required 153
B.9 Category 153
B.10 Ease of Exploit 154
B.11 Impact 154
B.12 Threat 156
B.13 Complexity of Exploit 156
B.14 Cohen's Attacks 157
B.15 Cohen's Attack Categories 158
B.16 Perry and Wallich Attack Classification 158
B.17 Howard's Process-Based Taxonomy of Network Attacks 159
B.18 Dodson's Classification Scheme 159

C IMPROVEMENTS ON PRIOR CLASSIFICATIONS 162
C.1 Indirect Impact 163
C.2 Direct Impact 163

Page

C.3 Access Required 163
C.4 Complexity of Exploit 163
C.5 Category 164
C.6 OS Type 164

VITA 171

LIST OF TABLES

Table Page

5.1 Keywords used in the co-word analysis run. 70

B.1 Values allowed for each level of the Howard's Process Based Taxonomy of Network Vulnerabilities 159

LIST OF FIGURES

Figure Page

2.1 Visualization of the definition of vulnerability 11

3.1 Natural classifications take advantage of taxonomic characters that are hierarchical in nature. 24

3.2 Natural clusterings group individuals together because they have similar characteristics. 26

3.3 There is often more than one way to correct a software fault and hence grammar-based classifications are not unique until a unique fix has been issued. 29

3.4 The Threat classification is ambiguous because it uses nodes that have more than one *fundamentum divisionis* 33

3.5 Decision tree for the classification of the direct impact of vulnerabilities. 38

5.1 Distribution of filled fields in the database 56

5.2 Scatter plots for some classifications. 57

5.3 Scatter plots for some classifications. 58

5.4 Distribution Plot the Nature of Threat Features 59

5.5 Distribution Plot for Environmental Assumption Features 60

5.6 Distribution plot for the Nature of Vulnerability feature Object Affected 61

5.7 Distribution plot for the Nature of Vulnerability feature Effect on Object . . . 62

5.8 Distribution plot for the Nature of Vulnerability Method feature 63

5.9 Distribution plot for the Nature of Vulnerability Method Input feature 64

5.10 Distribution plot for the System Features 65

5.11 Plot of Centrality vs. Density for the results of co-word analysis for the vulnerability database 72

5.12 Principal network number 1 for co-word analysis. 76

Figure Page

5.13 Isolated network number 2 for co-word analysis. 77
5.14 Isolated network number 3 for co-word analysis. 78
5.15 Isolated network number 4 for co-word analysis. 79
5.16 Principal network number 5 for co-word analysis. 80
5.17 Isolated network number 6 for co-word analysis. 81
5.18 Network number 7 for co-word analysis. 82
5.19 Network number 8 for co-word analysis. 83
5.20 Isolated network number 9 for co-word analysis. 84
5.21 Isolated network number 10 for co-word analysis. 85
5.22 A decision tree generated by MLC++ for predicting the direct impact of a vulnerability. 88
5.23 Visualization techniques can derive knowledge from vulnerability data. Example 1 91
5.24 Visualization techniques can derive knowledge from vulnerability data. Example 2 92
6.1 A classification for the identification of environmental assumptions made by programmers—Part 1. 97
6.2 Mapping the Classification to the Vulnerability Definition 98
6.3 A classification for the identification of environmental assumptions made by programmers—Part 2. 99
6.4 Distribution of vulnerabilities classified with the taxonomy presented in this section. 104
6.5 Taxonomy of Software Vulnerabilities Top Level 106
6.6 Taxonomy of Software Vulnerabilities, Levels 2-1 and 2-2 107
6.7 Taxonomy of Software Vulnerabilities, Levels 2-3 and 2-4 108
6.8 Taxonomy of Software Vulnerabilities, Levels 2-5 and 2-6 108
6.9 Taxonomy of Software Vulnerabilities, Level 2-7 109
6.10 Taxonomy of Software Vulnerabilities, Levels 2-8 and 2-9 110
6.11 Taxonomy of Software Vulnerabilities, Level 2-10 110
6.12 Taxonomy of Software Vulnerabilities, Levels 2-11 and 2-12 111
6.13 Programs normally execute code from well defined regions in memory, even if the memory is fragmented or the program contains dynamic executable code. . 116

Figure Page

6.14 A possible subtree of an evolutionary classification of software vulnerabilities. . 119
6.15 An example of a goal-oriented classification for software testing using environmental perturbations. 121
B.1 Aslam Classification decision tree (part 1 of 2) for the `classification` feature. 148
B.2 Aslam Classification decision tree (part 2 of 2) for the `classification` feature. 149
C.1 Selection decision tree for the `indirect_impact` classification. 165
C.2 Selection decision tree for the `direct_impact` classification. 166
C.3 Selection decision tree for the `access_required` classification. 167
C.4 Selection decision tree for the `complexity_of_exploit` classification. 168
C.5 Selection decision tree for the `category` classification. 169
C.6 Selection decision tree for the `os_type` classification. 170

ABSTRACT

Krsul, Ivan Victor. Ph.D., Purdue University, May 1998. Software Vulnerability Analysis. Major Professor: Eugene H. Spafford.

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, the development of a framework for the creation of taxonomies for vulnerabilities, and the application of learning, visualization, and statistical tools on a representative collection of software vulnerabilities.

This dissertation 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 dissertation 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.

This dissertation concludes by showing that the unifying definition of software vulnerability, the framework for the development of classifications, and the application of learning and visualization tools can be used to improve security.

1 INTRODUCTION

1.1 Problem Statement

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].

There is no single accepted definition of the term *software vulnerability*, and hence it is difficult to objectively measure the features of vulnerabilities, or make generalizations on this class of failures. 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; 1997a; 1997b; 1997c; Gavin 1998].

1.2 Thesis Statement

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 application of learning, visualization, and statistical tools on a representative collection of software vulnerabilities.

A unifying definition of software vulnerabilities can identify characteristics of vulnerabilities and allows researchers to agree on the object of study. A unifying definition can also be used to identify areas of focus for the development of taxonomies of software vulnerabilities.

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; Landwher et al. 1993; Cohen 1997a; 1997b]. However, as shown in Section 3.2, these classifications are ambiguous. The ambiguities are in part the result of conflictive 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.

The taxonomic characters developed for the classifications in taxonomies of software vulnerabilities can be used, in conjunction with the classifications themselves, to apply data mining and data visualization tools. These tools can reveal characteristics and properties of vulnerabilities that may not be apparent from the raw data.

1.3 Contributions

As shown in Section 2.1.3, the existing definitions of *software vulnerability* have one of the following forms: Access Control definitions, State Space definitions, and Fuzzy definitions. This dissertation provides a unifying definition based on the notion that it is security policies that define what is allowable or desirable in the system, and hence, the notion of software

vulnerability ultimately depends on our notion of policy. This dissertation also shows that existing classifications and taxonomies for software vulnerabilities, or related fields, do not satisfy all the desirable properties for classifications and taxonomies.

Section 3.1 defines the properties of measurements or observations necessary for the development of classifications; and provides a framework for the development of taxonomies for software vulnerabilities and related fields. This framework can be used as a basis for measuring features of vulnerabilities that can be used for the generation of classifications. These can be used to generalize, abstract, and communicate findings within the security research community, and contribute to our understanding of the nature of software vulnerabilities.

This dissertation presents an extension and revision of the classification of vulnerabilities presented by Aslam in [Aslam 1995]. Unlike its predecessor, this classification 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. Those vulnerabilities identified with this classification are not the result of software faults identified by common testing methods, because when tested in an environment that conforms to the assumptions made by programmers, the programs execute correctly.

This dissertation shows that machine learning and statistical analysis tools can reveal patterns and regularities that either reinforce our understanding of vulnerabilities, or provide new insights into the nature of vulnerabilities. Machine learning and statistical analysis tools can also influence the development of *a priori* classifications.

Finally, this dissertation describes how the development of taxonomies for software vulnerabilities can be used to build special domain-specific tools for the development of security-sensitive software.

1.4 Organization of the Dissertation

This dissertation is organized as follows: Chapter 2 introduces the terminology and the algorithmic conventions to be used throughout the dissertation. Chapter 3 presents the related work. Chapter 4 presents the development of new taxonomic characters for classifications and analysis. Chapter 5 shows how these taxonomic characters, measured for a collection of vulnerabilities, can be used with data mining and visualization tools.

Chapter 6 presents examples of *a priori* classifications and a taxonomy that focuses on vulnerabilities that result from mistaken environmental assumptions. Finally, Chapter 7 presents the conclusions, summarizes our findings, and discusses future directions.

2 NOTATION AND TERMINOLOGY

2.1 Terminology

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

2.1.1 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 [IEEE 1990]. 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 [IEEE 1990].

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].

2.1.2 Computer Policy

There are multiple definitions possible for computer policies, and the definitions presented in this dissertation have one of the following forms:

1. Policy helps to define what is considered valuable, and specifies what steps should be taken to safeguard those assets [Garfinkel and Spafford 1996].
2. Policy is defined as the set of laws, rules, practices, norms, and fashions that regulate how an organization manages, protects, and distributes sensitive information, and that regulates how an organization protects system services. [Longley and Shain 1990; DoDCSEC 1985; Sterne et al. 1991; Dijker 1996]

3. Access to a system may be granted only if the appropriate clearances are presented. Policy defines the clearance levels that are needed by system subjects to access objects [DoDISPR 1982; Sterne et al. 1991].
4. In an access control model, policy specifies the access rules for an access control framework [Kao and Chow 1995].

The key concepts in these definitions are value, authorization, access control, protection, and sensitivity of information. [Krsul 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 [Krsul 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.

Example 2.1: This example illustrates the difference between *policy* and *expected policy*. 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*). □

2.1.3 Software Vulnerability

Existing definitions of *software vulnerability* have one of three forms: Access Control, State-space, and Fuzzy.

[Denning 1983] states that an access control policy specifies the authorized accesses of a system and gives the following definitions of system states and policies:

The state of a system is defined by a triple (S, O, A), where:

1. S is a set of subjects, which are the active entities of the model. Subjects are also considered to be objects; thus $S \subseteq O$.
2. O is a set of objects, which are the protected entities of the system. Each object is uniquely identified with a name.
3. A is an access matrix, with rows corresponding to subjects and columns to objects. An entry $A[s, o]$ lists the access rights (or privileges) of subject s over object o.

Changes to the state of a system are modeled by a set of commands, specified by a sequence of primitive operations that changes the access matrix.

A configuration of the access matrix describes what subjects can do—not necessarily what they are authorized to do. A protection policy (or security policy) partitions the set of all possible states into authorized versus unauthorized states.

The exploitation of an **access control vulnerability** is whatever causes the operating system to perform operations that are in conflict with the security policy as defined by the access control matrix.

This definition requires the specification of a clear access control matrix that specifies what operations are allowed on system objects for every subject and object in the system. Such clear and precise access control matrices are not specified in operating systems such as UNIX, Macintosh OS, VMS, or Windows NT. Also, there are systems, as shown in the following example, where there are clear conflicts between the access control specifications in the operating system and the user's expectations.

Example 2.2: In UNIX and Windows NT systems, applications that are run by users inherit all the privileges that the access control mechanisms of the operating system provides to the user. The Java virtual machine is one of these applications and it defines, and is responsible for the enforcement of, its own access control matrix inside a *sandbox*. The sandbox is comprised of a number of cooperating system components that ensure that an untrusted—and possibly malicious—application cannot gain access to system resources [SECJAVA 97].

If the access control mechanism of the virtual machine fails, a hostile applet can be given access beyond the sandbox (as described in [McGraw and Felten 1997]). In such a case, the operating system will allow the hostile applet full access to the users files because

to the operating system there is no difference between the virtual machine and the applet. In such cases there will be a clear violation of expected cumulative access control rules. □

[Bishop and Bailey 1996] proposes a state-space definition of vulnerability:

> A computer system is composed of *states* describing the current configuration of the entities that make up the computer system. The system computes through the application of *state transitions* that change the state of the system. All states reachable from a given initial state using a set of state transitions fall into the class of *authorized* or *unauthorized*, as defined by a security policy... the definitions of these classes and transitions is considered axiomatic...
>
> A *vulnerable* state is an authorized state from which an unauthorized state can be reached using authorized state transitions. A *compromised state* is the state so reached. An *attack* is a sequence of authorized state transitions which end in a compromised state. By definition, an attack begins in a vulnerable state...

"A **state-space vulnerability** is a characterization of a vulnerable state which distinguishes it from all non-vulnerable states. If generic, the vulnerability may characterize many vulnerable states; if specific, it may characterize only one..." [Bishop and Bailey 1996]

The definition of a software vulnerability in [Bishop and Bailey 1996] is difficult to apply to systems such as UNIX or Windows NT because the identification of states is dependent on the intended functionality of the system. Even if we could enumerate all the possible safe and unsafe states of the system, say by partitioning the state-space, every program can have a unique state labeling that is a reflection of its intended behavior. Each safe and unsafe state labeled must take into account the environment of the system, including the user.

The Data & Computer Security Dictionary of Standards, Concepts, and Terms [Longley and Shain 1990] defines computer vulnerability as:

> 1) In computer security, a weakness in automated systems security procedures, administrative controls, internal controls, etc., that could be exploited by

> a threat to gain unauthorized access to information or to disrupt critical processing. 2) In computer security, a weakness in the physical layout, organization, procedures, personnel, management, administration, hardware or software that may be exploited to cause harm to the ADP system or activity. The presence of a vulnerability does not itself cause harm. A vulnerability is merely a condition or set of conditions that may allow the ADP system or activity to be harmed by an attack. 3) In computer security, any weakness or flaw existing in a system. The attack or harmful event, or the opportunity available to a threat agent to mount that attack.

Unlike the previous definitions, this one identifies that vulnerabilities are a function of perceived expectations at many levels. [Amoroso 1994] defines a *vulnerability* as an unfortunate characteristic that allows a threat to potentially occur. A *threat* is any potential occurrence, malicious or otherwise, that can have an undesirable effect on the assets and resources associated with a computer system.

According to these definitions, a **fuzzy vulnerability** is a violation of the expectations of users, administrators, and designers. Particularly when the violation of these expectations is triggered by an external object.

In the three previous definitions of computer vulnerability—access control, state-space, and fuzzy—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.

We give two arguments for the development of a unifying and practical definition for software vulnerabilities: improvement of systems, and identification of protection domains.

Improvement of Software Systems: Software vulnerabilities have undesirable consequences that go beyond the annoyance of common software system failures. The exploitation of vulnerabilities can affect the lives and livelihood of people and can have potentially disastrous effects [Leveson 1995]. Hence, an understanding of the nature of vulnerabilities would improve system design to reduce the risk of running critical or sensitive systems.

Identification of Protection Domains: Systems such as firewalls and systems that check for vulnerabilities attempt to reduce the risk of using computer systems by preventing the

exploitation of existing vulnerabilities [Farmer and Spafford 1991; Polk 1992]. In circumstances where it is not possible or feasible to eliminate the risk, intrusion detection systems attempt to detect the exploitation of vulnerabilities [Denning 1987; Kumar 1995]. A clear definition of software vulnerabilities identifies what these systems need to protect.

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.

Example 2.3: As shown in Figure 2.1, 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].

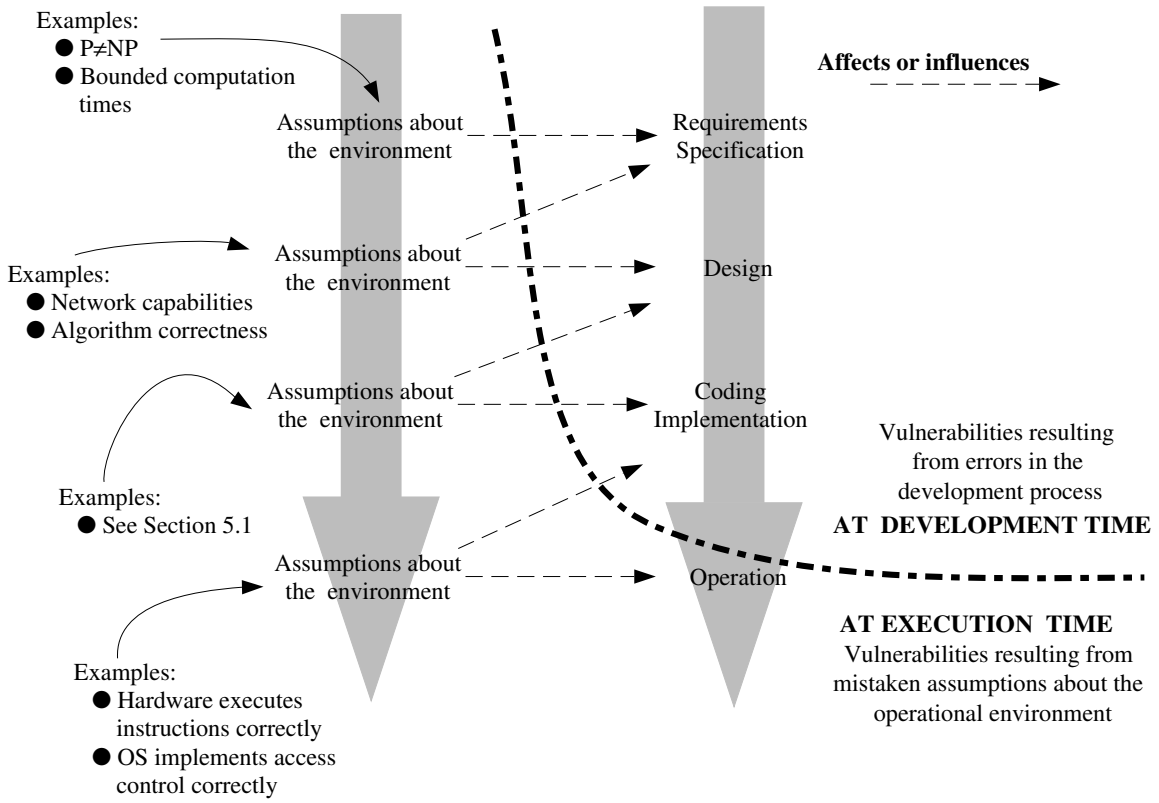


Figure 2.1: Vulnerabilities can result from errors in the specification, design, or coding of a system, or in environmental assumptions that do not hold at the time of execution of a program.

- 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.

□

2.1.4 Taxonomy and Classification

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

2.1.5 System Objects, Attributes and Constraints

The definition of *software vulnerability* presented in Section 2.1.3 includes mismatches between the assumptions about the environment made during the development and operation of the program, and the environment in which the program executes. The definitions in this section refer to these assumptions.

In a computer system, a **system 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.

Example 2.4: The following example illustrates the attribute refinement process: The object is the *running program*. An attribute of the running program is its *environment*. The *PATH environment variable* is an attribute of the program environment. The *first path*

in the PATH environment variable is a refinement of the PATH environment variable. Another refinement is possible if we look at the *first character* of the first path of the PATH environment variable.

Note that the length of the first path in the PATH environment variable is not an attribute because the length of the path is computed by the application of a function that counts the number of characters in the path. Hence, the length is a property. □

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

2.1.6 Definitions of Other Terms

Other terms in this dissertation are used according to their definitions in [Spencer 1983; Longley and Shain 1990; WEBOL 1998; Longley and Shain 1990; Mockapetris 1987; Albitz and Liu 1992; Bhushan et al. 1971; Crosbie 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].

2.2 Notation

In this section we present the algorithmic conventions that will be used throughout the dissertation. The notation is an adaptation of that of [Cohen 1990; Sethi 1989].

Functions are specified by indicating the types of parameters, the return type, and the function body. If the function returns a value, the function name is used as a variable to assign the return value. The format for a function is shown in equation 2.1.

Conditions and loops are indicated by the **if** and **do** keywords. The format of conditions and loops is shown in equation 2.2. One line conditional operations can be specified if the operations are followed by a condition. For example, a conditional assignment would have the following form:

$$x :=< \text{value} > \quad \textbf{if} \quad < \text{condition} >;$$

Statements and conditions are mathematical expressions where the following special operators are defined:

$\Rightarrow$ ***Some text*** $\Leftarrow$ The text inside the arrows is a comment.

$\wedge$ The short circuit AND logical operator. Used in logical expressions such as "**if** *object* is a file $\wedge$ *file name* is empty **then**." Short circuit operators are those where the expression is evaluated from left to right until a value can be determined for the operation but not more. For example, $x > 0 \wedge y/x > 5$ would not result in a division by zero error if $x = 0$ because when the $\wedge$ operator is reached the left hand side of the operator is *false* and hence the entire expression cannot be *true*. Hence, the expression $y/x > 5$ does not need to be evaluated.

$\vee$ The short circuit OR logical operator. Used in logical expressions such as "**if** *object* is a file $\vee$ *object* is a pipe **then**." Short circuit operators are those where the expression is evaluated from left to right until a value can be determined for the operation but not more. For example, in the expression $x > 0 \vee x < -100$, the right hand side of the expression, $x < -100$, would not get evaluated if $x > 0$ because the left hand side of the expression is enough to yield the entire expression *true*.

$\forall$ The *for all* operator that iterates over all the elements of a set. For example, a loop construct such as "$\forall x \in S$ **do**" would iterate x over all the elements of set S.

$\in$ The *in* operator that tests for set membership.

$:=$ Assignment operator.

$::=$ the *definition* operator. Used to define functions and terms.

$|$ The *such that* operator. It can be used as a qualifier with the $\forall$ operator. For example, the expression "$\forall x \in S \mid x \in E$" would iterate over all the elements in set S that are also in E.

$\triangleright$ A string comparison and substring operator. The expression $x \triangleright y$ evaluates to `true` if the string y begins with string x (x is a substring of y starting with the first character of y). The expression $\triangleright x\,(n, m)$ returns the substring of x starting at the character in

position n and ending in the character in position m. The first character of a string x is $\triangleright x\,(0,0)$.

Other operators, such as $\leq, \neq, =, *, ()$, etc., have their generally accepted meaning.

Function Name : **parameter type** $\times \cdots \times$ **parameter type** $\rightarrow$ **return type**
fun *Function Name* (*parameter name*, $\cdots$, *parameter name*) ::=
Function Body Line 1;
$\vdots$
Function Body Line n;
nuf (2.1)

if *Condition* **then**	*Loop Condition* **do**
Condition Body Line 1;	*Loop Body Line 1*;
$\vdots$	$\vdots$
Condition Body Line n;	*Loop Body Line n*;
fi	**od**

(2.2)

3 RELATED WORK

This chapter presents an overview of related work in classification theory,

3.1 Classification Theory

Taxonomies increase our understanding of the world. [Simpson 1945] summarizes the value of [animal] taxonomies:

> Taxonomy is at the same time the most elementary and the most inclusive part of zoology, most elementary because animals cannot be discussed or treated in a scientific way until some systematization has been achieved, and most inclusive because taxonomy in its various guises and branches eventually gathers together, utilizes, summarizes, and implements everything that is known about animals, whether morphological, physiological, or ecological.

A function of these 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 1995; 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. 1995; 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 (as shown in Section 3.2). 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.

3.1.1 Historical Background

Making sense of apparent chaos by finding regularities is an essential characteristic of human beings, as argued by the Austrian-born British philosopher of natural and social science Karl Popper: "we are born with expectations... one of the most important of these expectations is that of finding a regularity. It is connected with an inborn propensity to lookout for regularities, or with a *need* to *find* regularities..." [Popper 1969]

The first attempt at constructing a systematic classification was developed by the Greek philosopher Aristotle (322-284 B.C.), who concentrated mainly on animals. His pupil Theophrastus (371-287 B.C.) concentrated mainly on the ordering of plants. There were no significant advances in taxonomy for the next 2,000 years until the beginning of the 16th century [Grolier Incorporated 1993; Durant 1961; Simpson 1961].

Originally, taxonomies were the exclusive domain of the biological sciences. The Encyclopedia Americana still defines Taxonomy as "the theory and practice of classifying

organisms... two branches of biology—systematics and taxonomy—cover this area. Systematics is concerned with the entire diversity of organisms and all its aspects includes taxonomy" [Grolier Incorporated 1993].

With Darwin, who published in 1859 "*The Origin of Species*," starts the period of modern evolutionary taxonomies. Since then, we have seen the development of taxonomies in fields as diverse as communication media [Bretz 1971], soil science [Bailey 1987], computer graphics [Bier et al. 1995], computer databases [Olivier and Vonsolms 1994], and computer security [Roskos et al. 1990; Young and Taylor 1991; Bishop 1995; Landwher et al. 1993; Aslam et al. 1996; Oman and Cook 1991; Kumar et al. 1995].

3.1.2 Taxonomic Characters, Object Attributes or Features

The basis for the development of successful classifications are taxonomic characters [Simpson 1961; Glass and Vessey 1995]. These are the properties or characteristics of the objects that will be classified. Taxonomic characters are also commonly called features, attributes or characteristics. [Simpson 1961] argues that such properties should be readily and objectively observable from the objects in question.

[OXFORD 1998] defines Objectivity as "the quality or character of being objective; external reality; objectiveness," and objectiveness as "the character of dealing with or representing outward things rather than inward feelings." [WEBOL 1998] defines objectivity as "expressing or dealing with facts or conditions as perceived without distortion by personal feelings, prejudices, or interpretations." Objectivity implies that the property must be identified from the object known and not from the subject knowing.

If the property is being deduced, rather than observed, then its value will hold the observer's bias, and other taxonomists cannot necessarily repeat the deduction without knowing the role of the bias, and the measurement cannot be validated. Objective and observable properties simplify the work of the taxonomist and provide a basis for the repeatability of the classification. "The good [taxonomist] develops what the medieval philosophers called a *habitus*, which is more than a habit and is better designated by its other name of *secunda natura*. Perhaps, [as with] a tennis player or a musician, he works best when he does not get too introspective about what he is doing." [Thompson 1852].

Taxonomic characteristics must satisfy the following properties:

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 a few examples from the field of computer security.

Example 3.1: Consider the taxonomy of UNIX system and network vulnerabilities proposed in [Bishop 1995]:

> To summarize, the taxonomy we use has six axes, and every vulnerability is classified on each axis. The first axis is the *nature* of the flaw, and we use the Protection Analysis categories; the second axis is the *time of introduction*, and we use the (modified) classes of Landwehr. Third is the *exploitation domain* of the vulnerability and fourth is the *effect domain*; for these, we use the classes outlined above. The fifth axis is the *minimum number* of components needed to exploit the vulnerability. The sixth axis is the *source* of the identification of the vulnerability.

The *nature*, *exploitation domain*, and *effect domain* taxonomic characters are themselves classifications that are ambiguous (see Section 3.2 for a detailed analysis). The *time of introduction* character is ambiguous if the vulnerabilities can reappear. The procedure for determining the value of this character is given as:

> For our purposes, we adopt the following definitions. Suppose we have some software *plugh*, at version *xyzzy*. If a security vulnerability exists in all versions of *plugh* up to version *xyzzy*, it is in the class "during development." (If we can

> further identify the flaw as being introduced in design or in implementation, we shall do so; but this is not always obvious.) If there is a version *glorkz* before which no version of *plugh* had the vulnerability, but the vulnerability exists in versions of *plugh* from *glorkz* to *xyzzy*, it will be in the class "during maintenance." If the vulnerability depends only upon the operation of the entity, then we shall put it in the class "during operation."

Assume that a software had a vulnerability in versions 1.1 and 3.1 and in no other version of the software. Then *xyzzy* corresponds to 3.1. The vulnerability does not exist in all versions up to 3.1 so the time of introduction cannot be "during development." There exists a version 1.1 (*glorkz*) before which no version had the vulnerability but the vulnerability does not exist in versions from 1.1 to 3.1 so the time of introduction cannot be "during maintenance." The vulnerability does not depend on the operation of the entity so the time of introduction cannot be "during operation." At this point there are no more possibilities and the value of the characteristic is ambiguous.

It can be argued that the value of the characteristic is "during maintenance" because the question in the definition can be interpreted as "... but the vulnerability exists in *some* versions of *plugh* from *glorkz* to *xyzzy*..." However, the definition can also be interpreted as "... but the vulnerability exists in *all* versions of *plugh* from *glorkz* to *xyzzy*..." The value of the characteristic is then left to the interpretation of the taxonomist and the characteristic fails the specificity principle.

The definition of the *minimum number* characteristic fails to mention what can be considered a component. The specifications for this characteristic reads "... the question of number of components required to exploit the vulnerability is a good metric, because it indicates the number of programs' audit records [that] must be analyzed to discover the exploitation..." In the examples of classification we see the following values for this characteristic: "... the *mkdir* process and another process to delete the directory and link the password file to the name," "... the *sendmail process*," and "... the back-door in the *login* program." Component is more than a process and includes a back-door. The characteristic is not objective (it is not clear how a back-door is reflected in the audit records), not deterministic (there is no procedure), and not repeatable (it is not clear how a component is defined). □

Example 3.2: A taxonomy where the taxonomic characteristics do not satisfy the criteria outlined in this section is presented in [Landwher et al. 1993]. One of their taxonomic characteristics is *genesis*. In [Landwher et al. 1993] we read "Characterizing intention is tricky... Although some malicious flaws could be disguised as inadvertent flaws, this distinction should be easy to make in practice—inadvertently created Trojan horse programs are hardly likely!" In practice, however, there does not exist a procedure that can be followed to determine the value of *genesis* and hence the characteristic is not objective and not specific. □

Example 3.3: 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 (see Appendix B), [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 examples presented fail to satisfy the requirements specified, they enumerate 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 [Roskos et al. 1990; Cohen 1997a; 1997b].

3.1.3 Taxonomies and Classifications

Section 3.1 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.

As the following examples show, computer security practitioners confuse the terms *classification* and *taxonomy*:

Example 3.4: The 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. □

Example 3.5: The taxonomy of security faults developed in [Aslam 1995], and later refined in [Aslam et al. 1996], is not a taxonomy but a classification scheme because it does not present generalizations about the classification and does not discus the explanatory and predictive properties of the classification scheme. □

Example 3.6: The taxonomy of computer program security flaws developed in [Landwher et al. 1993] fails to provide principles and generalizations about the classification scheme provided. □

Example 3.7: The taxonomy of UNIX system and network vulnerabilities developed in [Bishop 1995] provides explanatory value by providing generalizations about the classes. Unfortunately, it does this for only one of the six taxonomic characteristics. Other problems with this taxonomy were discussed in Section 3.1.2. □

3.1.4 Types of Classifications

As stated in Section 2.1.4, 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 3.1.2, classification schemes that can be built include arbitrary selections, decision trees, natural classifications, evolutionary classifications, and natural clusterings.

Arbitrary Selections

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.

Decision Trees

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*. Figure 3.5 is an example of an *a priori* decision tree for classifying the direct impact of a vulnerability.

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.

Consider an example of a decision that has multiple *fundamentum divisionis* from the biological sciences: In a decision tree that is used to classify members of the canine family (i.e dogs and wolves) we cannot have a single node that asks the question "*does the animal have legs or hair?*" A single specimen can have both characteristics and hence the classification would be ambiguous.

Note that it is possible to use more than one characteristic to build the selection criteria in a node of a decision tree so long as the *fundamentum divisionis* remains the same.

Natural Classifications

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. Figure 3.1 illustrates how taxonomic characteristics that are hierarchical in nature favor natural classifications.

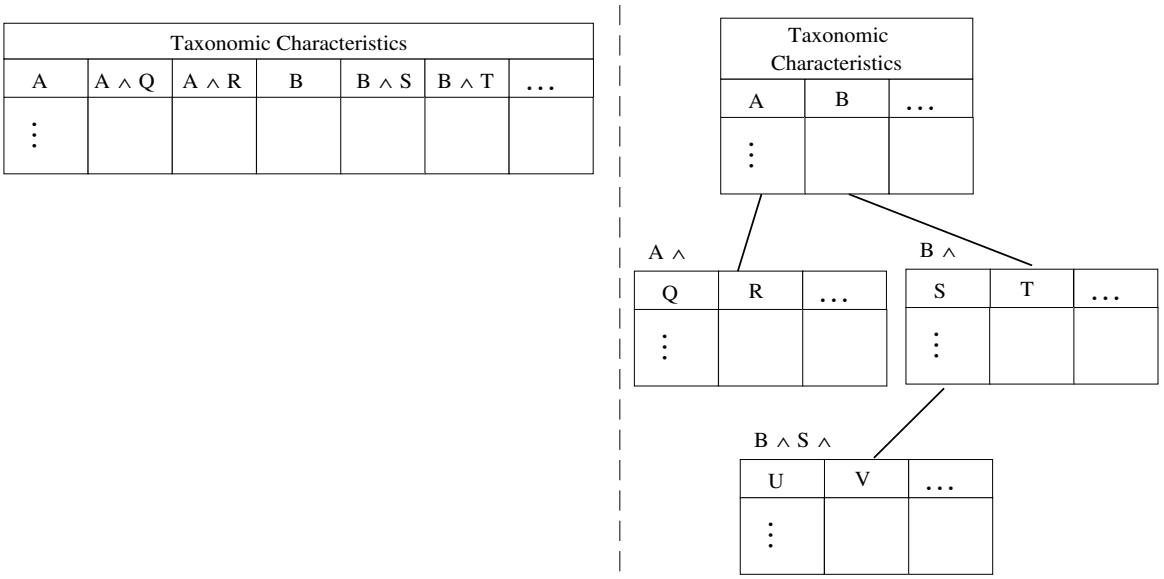


Figure 3.1: Natural classifications take advantage of taxonomic characters that are hierarchical in nature.

Features of computer programs are well suited for natural classifications, as specialized measurements can only be made once a required characteristic has been identified.

Example 3.8: The taxonomic character "program uses inheritance" requires that the programming language is object oriented. Similarly, the taxonomic characteristic "program uses native methods" requires that the program run in a virtual machine. □

Evolutionary Classifications

Modern biological and botanical taxonomies are evolutionary [Simpson 1945] and its basis involves phylogeny, or the history of development of an organism [WEBOL 1998; OXFORD 1998]. Evolutionary classifications look at propinquity of descent (nearness in

time and place [WEBOL 1998; OXFORD 1998]) to judge similarities between individuals. Because evolutionary classifications look at the development or evolution of an individual by looking at propinquity of descent, it is necessary to include an explicit time axis.

In evolutionary taxonomies individuals do not belong to the same class because they are similar, but rather are similar because they belong to the same class. Similarities are indicators of propinquity of descent but it is possible to have similar individuals that are not evolutionary related. In the field of biological taxonomy the terms *homology*, *homoplasy*, *parallelism*, *analogy*, *mimicry*, and *chance similarity* are commonly used to describe different types of similarities in individuals that are being classified using evolutionary classifications [Simpson 1945].

Natural Clusterings

Unlike evolutionary classifications, 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 3.2 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.

3.2 Prior Software Vulnerability Classifications

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; Bibsey 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

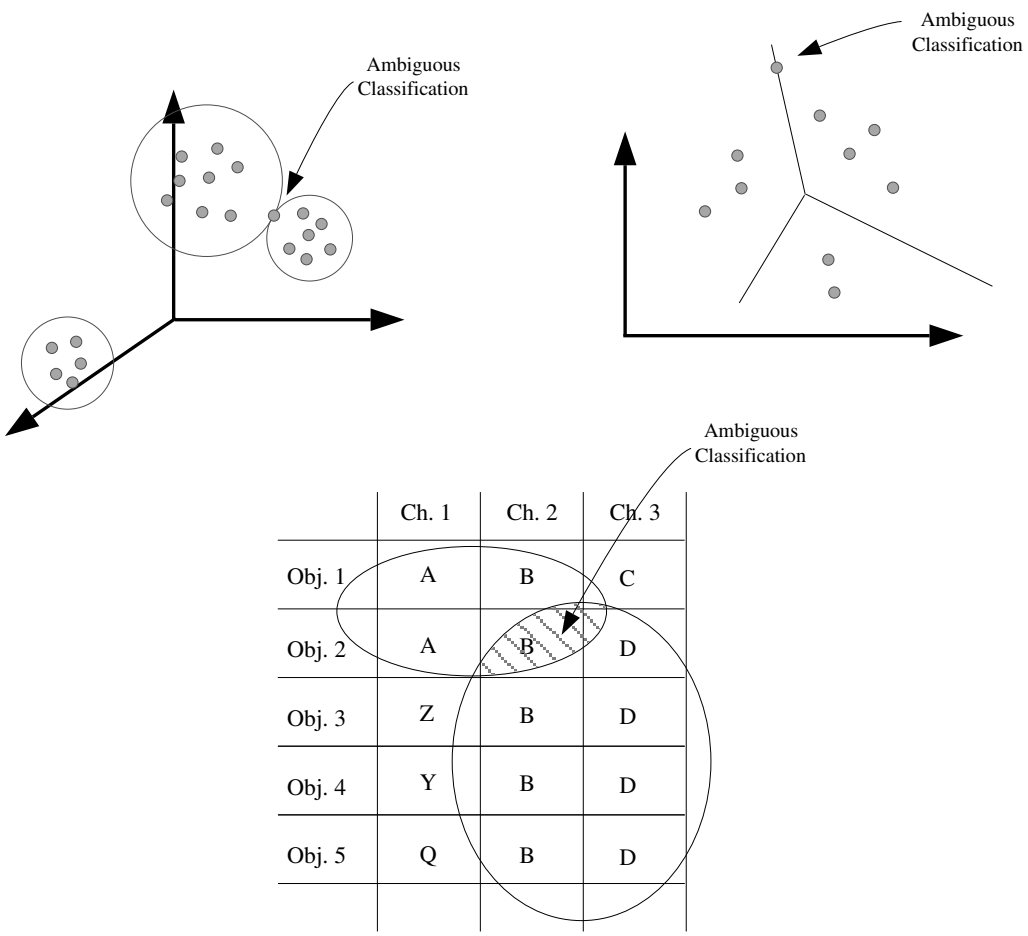


Figure 3.2: Natural clusterings group individuals together because they have similar characteristics. Note that classification can be ambiguous if an individual has the same number of similar characteristics with more than one group.

from the software engineering literature (the studies reported faults that were discovered in production quality software)

In this section we list the classifications that were developed in relation to software faults and vulnerabilities, discuss their limitations, and show that 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 these classifications are mostly the result of conflicting definitions of software vulnerability or software fault.

3.2.1 Aslam Classification

[Aslam 1995; Aslam et al. 1996] 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. For example, while talking about the `xterm` log file flaw they argue that:

> The selection criteria for fault classification places the flaw in class 1a from the point of view of the attacking program (object installed with incorrect permissions, because the attacking program can delete the file), in class 3a4 from the point of view of the *xterm* program (access rights violation error, as *xterm* does not properly validate the file at the time of access), and class 3b1 from the point of view of the operating system (improper or inadequate serialization error, as the deletion and creation should not be interspersed between the access and the *open*). As an aside, absent the explicit decision procedure, the flaw could have been placed in class 3b2, race conditions...

3.2.2 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 [Knuth 1989] and developed a detailed classification of types of faults found in his system.

It is difficult to apply the Knuth classification scheme to errors of programs where the person performing the classification is not the original programmer. As shown in the following example, the classification is subjective and ambiguous.

Example 3.9: Consider the following program segment:

```
if(strncmp(str1,str2,strlen(str1))==0 ||
   strncmp(str2,str1,strlen(str2))==0)
      if(strncmp(str1,str2,strlen(str1))<0)
            /* str1 is a substring of str2 */
      else
            /* str2 is a substring of str1 */
```

The error in the program is in that the second `if` statement uses the `strncmp` function to test whether a string is lexically smaller than the other. There are various ways in which this error can classified with the Knuth error classification, in this particular case, depending on our choice of a solution as follows:

If we assume that the programmer mistakenly used the `strncmp` function knowing that what was really needed was the `strcmp` function, resulting in the correct program segment following, then the error falls under the category of *blunder or botch*. As stated in [Knuth 1989]: "Here I knew what I ought to do, but I wrote something else that was syntactically correct—sort of a mental typo."

```
if(strncmp(str1,str2,strlen(str1))==0 ||
   strncmp(str2,str1,strlen(str2))==0)
      if(strcmp(str1,str2)<0)
            /* str1 is a substring of str2 */
      else
            /* str2 is a substring of str1 */
```

If, on the other hand, the programmer wanted to use the `strncmp` function then a possible correct program segment following. Here the error is either *algorithm awry* because the algorithm itself is incorrect or a *language liability* because the programmer failed to understand the semantics of the `strncmp` function.

```
if(strncmp(str1,str2,strlen(str1))==0 &&
   strncmp(str2,str1,strlen(str2))<0)
       /* str1 is a substring of str2 */

if(strncmp(str2,str1,strlen(str2))==0 &&
   strncmp(str2,str1,strlen(str1))<0)
       /* str2 is a substring of str1 */
```

□

3.2.3 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. Consider, for example, the fault shown in Figure 3.3: The program segment will attempt to access unallocated memory when it tries to fill slot `arr[MAX]`. As the corrections suggested in the same figure show, there is more than one way to solve this problem. Hence, the fault classification is not unique until a definitive and unique fix to a fault is selected.

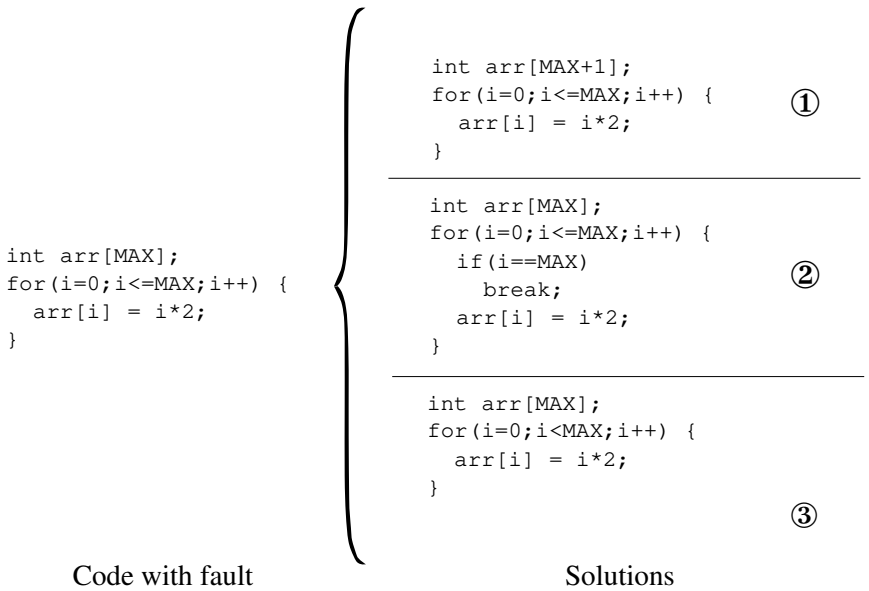


Figure 3.3: There is often more than one way to correct a software fault and hence grammar-based classifications are not unique until a unique fix has been issued.

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.

3.2.4 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].

3.2.5 Ostrand and Weyuker Classification

[Ostrand and Weyuker 1984] proposes an attribute categorization scheme for the classification of faults. [DeMillo and Mathur 1995] shows that the Ostrand and Weyuker classification scheme is ambiguous and it provides the following example as an illustration:

> ... consider the following erroneous program fragment written in C:
>
> if $(a = 1)\, p = q$;
>
> The condition $a = 1$ in the preceding fragment should be $a == 1$. The incorrect program modifies the value of data, namely the variable a, and hence corresponds to major category *data handling*. However, the fault appears in the condition part of the `if` statement. Hence, it also falls under the major category `decision` in spite that the fault has no effect on the truth value of the condition and the path followed thereafter.

3.2.6 Basili and Perricone Classification

This classification was in [Basili and Perricone 1984]. [DeMillo and Mathur 1995] shows that it is ambiguous by giving the following example:

> ... consider the following program segment containing one fault:
> $\vdots$
> $A[j] = 0; i = 1;$
> $\vdots$
> $\text{if}(A[i] < 0) \ldots$
> $\vdots$
>
> In the preceding segment, $A[j]$ should be $A[i]$... this fault belongs to the data error category because an incorrect subscript, namely j, has been used to index A. However, the effect of this fault is that an incorrect path may be taken when the following `if` is executed. Thus, this fault also belongs to the control error category.

3.2.7 Origin and Causes Classification

This classification was originally defined in [Longstaff 1997] to identify the origins of vulnerabilities. This classification is difficult to use without detailed information about the state of mind of the programmer during the development process. There is no definition of a "debugging statement." An external reviewer cannot determine if the vulnerability was caused by inconsistent specifications or lack of training of the programmers who implemented the functionality.

3.2.8 Access Required Classification

This classification was originally defined in [Longstaff 1997] and defines the access that is required to exploit the vulnerability.

There is no clear definition of each category. In UNIX, for example, there is no consensus about which of the following accounts can be considered a privileged account: `root`, `bin`, `ftp`, `http`, `nobody`, `krsul`, etc. One possible interpretation of privileged access is that any account where the user has UID or GID of 0 can be considered a privileged account. Other interpretations may consider the user `bin` as a privileged account regardless of its UID or GID.

3.2.9 Category Classification

This classification identifies the system component to which a vulnerability belongs. This classification is common in the vulnerability databases described in Section 3.3.

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].

3.2.10 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.

3.2.11 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 the vulnerability databases described in Section 3.3.

This classification is a decision tree of depth one that, as shown in the following example, has more than one fundamentum divisionis:

Example 3.10: A vulnerability in UNIX that allows an attacker to overwrite the file `/vmunix` will effectively disable the system and overwrite information at the same time. Hence, the ultimate impact would be a denial of service *and* loss of system data. □

3.2.12 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.

In Section 3.1 we state that classification trees should use decision nodes that use one *fundamentum divisionis* and, as can be seen in Figure 3.4, the threat classification does not follow this principle.

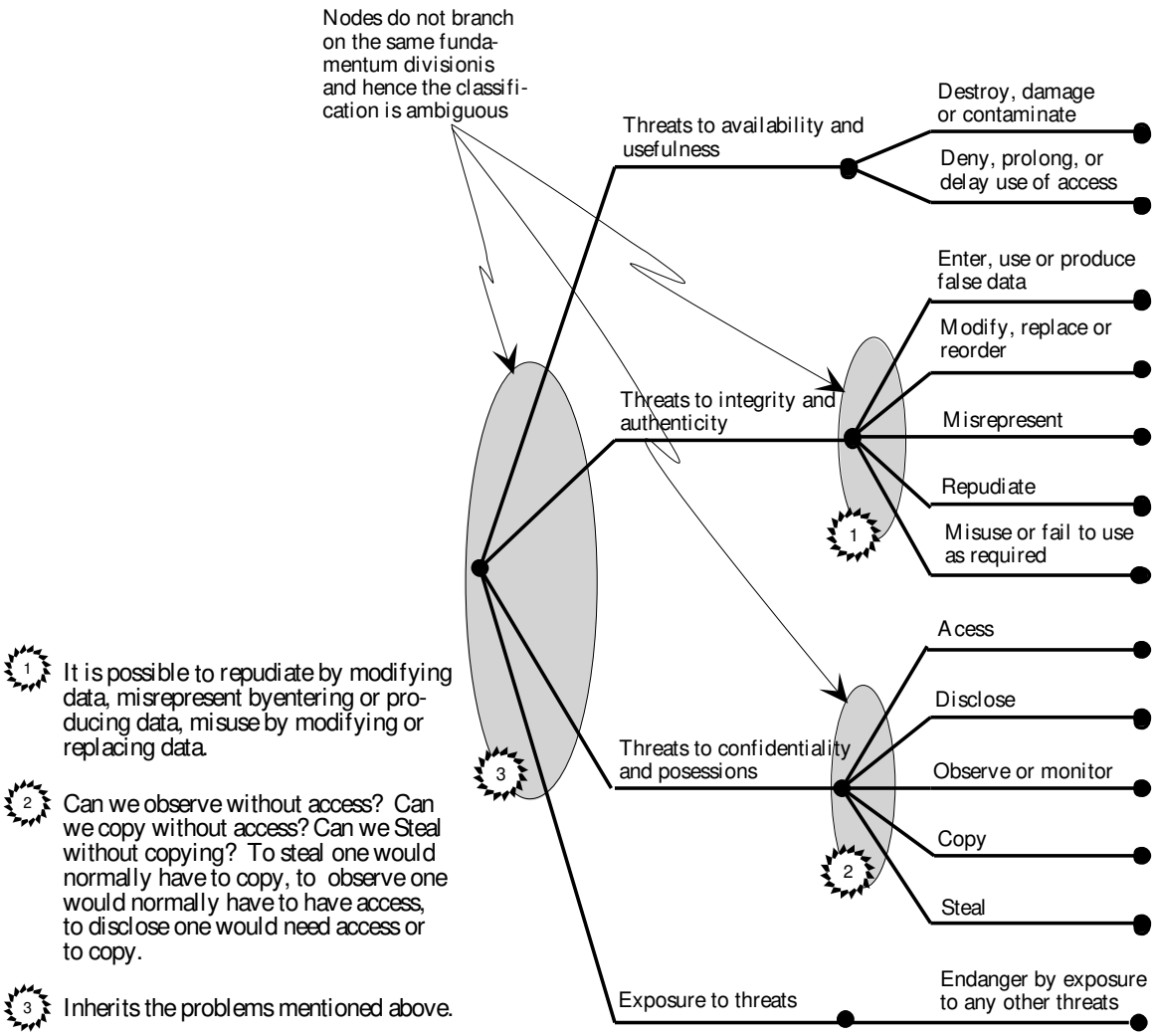


Figure 3.4: The Threat classification is ambiguous because it uses nodes that have more than one *fundamentum divisionis*

In the threat classification, a single node in the tree branches into *Access*, *Disclose*, *Observe*, *Copy*, and *Steal*. The categories *Observe* and *Access* are concrete actions while the category *Steal* is subjective and requires a value judgment. Hence, it is possible to *Access* and *Steal* simultaneously.

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.

Also, in systems where the administrator has unbounded privileges, such as UNIX or Windows NT, access to this account implies all threats at once. For example, if an attacker obtains a root account in UNIX then he could erase the hard disk (destroy, damage or contaminate), modify the password file (enter, use or produce false data), send email pretending to be another user (misrepresent), read any user's email (access), etc.

3.2.13 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 subjective as there is no accepted definition of "simple sequence of commands," "complex set or large number of commands," etc.

3.2.14 Cohen's Attack Classification

This classification is a subset of 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, and 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.

3.2.15 Perry and Wallich Attack Classification

This is a matrix-based classification scheme in two dimensions: Potential perpetrators and potential effects [Perry and Wallich 1984].

Consider the Java vulnerability where the restriction allowing an applet to only connect to the host from which it was loaded was not properly enforced by the browser. This vulnerability, combined with the subversion of the DNS system, allowed an applet to open a connection to an arbitrary host on the Internet. In the browser Java implementations, the Applet Security Manager allowed an applet to connect to any of the IP addresses associated with the name of the computer from which it came.

An attacker in control of a DNS server, say at `moria.mordor.com`, could design an applet that, when sent to a victim machine, would attempt to connect back to the server where it came from by connecting to `gollum.mordor.com`. Because `moria.mordor.com` is the DNS server for the `mordor.com` domain, it can return a set of IP addresses for `gollum.mordor.com` in a tuple of the form {`IP Address 1`, `IP Address 2`, etc.}. If `IP Address 1` corresponds to a machine outside the `mordor.com` domain, and that could include a machine behind a firewall if the client is behind the same firewall, then the applet will be allowed to connect to that IP address, regardless of where that machine really is (For a more details description of the problem see [McGraw and Felten 1997]).

For the *potential perpetrators* dimension we could choose internal because the applet that is attempting the connection is already running on the machine in an internal network, or we could choose outside because the applet originated from an external site. The results of the vulnerability could be theft of services because the applet is successfully using services that may not be available (it could be consuming pay-per-use services), theft of information because the applet may be scanning the ports on a machine and that information may be sensitive or valuable, or browsing if the applet is acting as a proxy to an external agent.

3.2.16 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 "link" between attackers and objectives is established through an operational sequence as follows: *Attackers* $\Rightarrow$ *tools* $\Rightarrow$ *access* $\Rightarrow$ *results* $\Rightarrow$ *Objectives*. Section 3.1.2 presents some of the limitations of this taxonomy.

3.2.17 Dodson's Classification Scheme

This classification was developed in [Dodson 1996] for the classification of computer vulnerabilities such that the classes identify generic flaws in software. These can be detected by using the Tester's Assistant, a tool used to automate software testing [Fink et al. 1994]. This classification is an extension of that proposed in [Aslam 1995].

3.3 Vulnerability Databases

Several groups have constructed vulnerability databases. Private databases of restricted distribution include the CMET database at the Air Force Information Warfare (AFIW) Center [Air Force Information Warfare (AFIW) Center 1996]; the database maintained by Mike Neuman [Neumann 1995]; the database at the Computer Emergency Response Team (CERT) [CERT Coordination Center 1998d]; the database of the Australian Computer Emergency Response Team (AUSCERT) [AUSCERT Coordination Center 1998]; the database maintained by Michael Dilger [Dilger 1995]; and the internal vulnerability databases at Netscape [Netscape Communications Corporation 1996], Sun [Sun Microsystems Inc. 1997], and Haystack Labs [Haystack Labs, Inc. 1996].

Publicly available databases include the vulnerability database at Internet Security Systems (ISS) [Internet Security Services 1998]; the vulnerability database at INFILSEC [INFILSEC Systems Security 1998]; the vulnerability and exploit collections of various groups such as Firosoft Consulting [Firosoft Consulting 1998], Kao's Unix Security Library [VDBKAO 1998], Rootshell [VDBFIR 1998], The Brotherhood of Darkness [VDBBOD 1998], The Legacy [VDBLEG 1998], Future Kill [VDBFKI 1998], Security Bugware [VDBBUG 1998], "Known NT Exploits" [Stout 1998], NegativeZero Exploit Page [VDBNZE 1998],

Elitehackers [VDBELI 1998], Dop's terribly geeky page of naughty hacks [VDBDOP 1998], and Exploit World [VDBEXP 1998]. Other un-named databases are [VDBNN1 1998; VDBNN2 1998; VDBNN3 1998; VDBNN4 1998; VDBNN5 1998; VDBNN6 1998].

The databases at Firosoft Consulting, NegativeZero Exploit Page, Elitehackers, "Known NT Exploits," Kao's Unix Security Library, Security Bugware, Rootshell, The Brotherhood of Darkness, The Legacy, and Future Kill are collections of exploit scripts, messages extracted from security mailing lists such as BUGTRAQ and NTBUGTRAQ, and advisories from the CERT, the Department of Energy's Computer Incident Advisory Capability (CIAC), the AUSCERT, the 8lgm security advisories, etc. These databases are freely available in the Internet.

The database at Exploit World provides, in addition, a simple characterization that includes information regarding the systems affected by the vulnerability and the potential ultimate impact that the vulnerability can have in a system. This database is freely available on the Internet.

The vulnerability databases of ISS, INFILSEC, and Michael Dilger make allowances for characterizations and classifications. However, none of these characterizations satisfy the properties required in in Section 3.1.2.

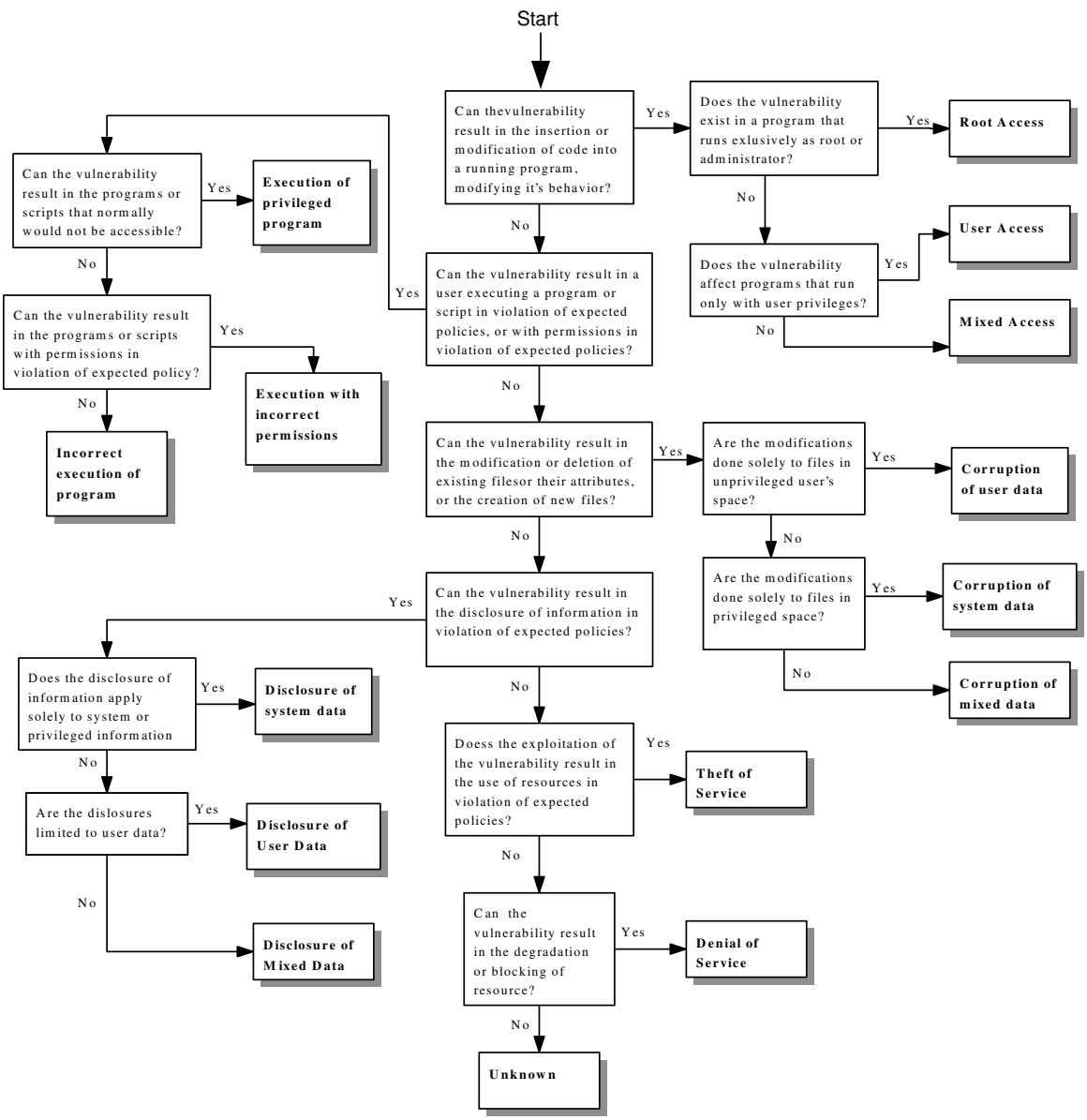


Figure 3.5: Decision tree for the classification of the direct impact of vulnerabilities.

4 DEVELOPMENT OF NEW TAXONOMIC CHARACTERS

A limitation of current classifications is the lack of taxonomic characters (or features) as detailed in Section 3.1.2. In this section we suggest new taxonomic characters that can be used as a foundation for improved classifications and taxonomies. Where possible, the taxonomic characters are observations or measurements that do not require the application of decision trees to avoid ambiguities resulting from the nodes having more than one *fundamentum divisionis*. A similar approach is used in [Dodson 1996] as an extension of the classification developed in [Aslam 1995].

4.1 Threat Features

One comprehensive threat classification—extracted from [Power 1996]—was developed to identify hostile actions that an adversary could take against your system. We split the classification into a list of binary action and consequence features, and we clearly specify that we are interested in the immediate threat that is present with the vulnerability, and not the ultimate threat. This clarification is necessary to avoid the ambiguity that comes from indirect threats..

The action features identified are (1) The exploitation of the vulnerability can result in a user observing objects, data, etc., in violation of expected policy. (2) The exploitation of the vulnerability can result in a user destroying objects, data, etc., in violation of expected policy. (3) The exploitation of the vulnerability can result in a user modifying objects, data, etc., in violation of expected policy. (4) The exploitation of the vulnerability can result in a user creating objects in violation of expected policy.

The consequence features identified are (1) The exploitation of the vulnerability can result in a change of availability of the system. (2) The exploitation of the vulnerability can result in the disclosure of information in violation of expected policy. (3) The exploitation

of the vulnerability can result in the misrepresentation of information. (4) The exploitation of the vulnerability can result in repudiation of information. (5) The exploitation of the vulnerability can result in a change of integrity of the system. (6) The exploitation of the vulnerability can result in the loss of confidentiality of information.

Each of these features can take the values "Yes," "No," "Does Not Apply," and "Unknown." Hence, each feature is a decision tree with a depth of one that has a single *fundamentum divisionis*.

4.2 Environmental Assumption Features

These features identify the environmental assumptions that were made by programmers or designers and that, if correct, would eliminate the vulnerability. Each of these features can take the values "Yes," "No," "Does Not Apply," and "Unknown." Hence, each feature is a decision tree with a depth of one that has a single *fundamentum divisionis*.

The following environmental assumption features are defined:

permsdir: A programmer assumes that a set of directories have a specific set of permissions (or minimum set of permissions)

nocore: A programmer assumes that a user cannot see the internals of the program as it is executing (i.e., no user readable core dumps in UNIX).

nameinv: A programmer assumes that a name (i.e., a path) is strongly bound to a specific system object.

objinv: A programmer assumes the invariance of an object during the execution of a program (i.e., the program assumes that no other subject can change the object while the program is running).

objne: A programmer assumes that an object does not exist at the time of execution (i.e., a program assumes that a file with a specific name does not exist).

tempdel: A program assumes that a temporary item it created cannot be deleted by any other subject while the program is running.

memavail: A programmer assumes that sufficient memory for its execution will always exist.

netdata: A programmer assumes that data from a network service will always be valid and bounded.

envdata: A programmer assumes that the data in environment variables is valid and bounded.

userdata: A programmer assumes that user-provided input is valid and bounded.

filedata: A programmer assumes that the input from a file is valid and bounded.

reassembly: A programmer assumes that the re-assembly of a data object from fragments will not affect the essential properties of the original object.

execpath: A programmer assumes a specific execution path.

objatt: A programmer assumes that certain attributes of certain objects have predefined values.

perstore: A programmer assumes that persistent store is immutable (i.e., assumes that a file it writes cannot be modified by any other subject in between program runs).

dataexec: A programmer assumes that the modification of program data (by external subjects) will not affect the semantics of the program.

nameover: A programmer assumes that, while creating a file, any existing file that has the same name can be overwritten.

falseconst: A programmer falsely assumes that a constraint or property holds in the system.

insufverif: A programmer falsely assumes that a set of operations are sufficient for the verification of the property of an object

namepurpose: A programmer assumes that there is a strong binding between the name and purpose of an object.

reservedobject: A programmer assumes that an object with a specific name will not be used by any other entity in the system by virtue of its name alone.

turstnetobj: A programmer assumes that a network object that claims an identity can be trusted.

4.3 Features on the Nature of Vulnerabilities

These features capture the effects of vulnerabilities by looking at the object that is immediately affected by the vulnerability, the effect that the vulnerability has on that object, the method or means that lead to the effect on the object, and, if appropriate, the type of input that leads to the effect on the object.

Each of these features can take the values "Yes," "No," "Does Not Apply," and "Unknown." Hence, each feature is a decision tree with a depth of one that has a single *fundamentum divisionis*.

4.3.1 Objects Affected

The features that identify the objects affected directly by the vulnerability are:

command_prompt: A command prompt presented to the user.

user_files: User files in the system.

system_files: System-related or administrative files in the system.

public_files: Publicly available files in the system.

directory: Directories in the system.

partition: A file system partition.

heap_data: Data in the heap of a running program.

heap_code: Executable code in the heap of a running program.

stack_data: Data in the stack of a running program.

static_data: Data that is statically allocated in a running program.

stack_return: Return address of a function in the stack of a running program.

stack_code: Executable code in the stack of a running program.

password: Password or access token. Can also be a pass-phrase.

shell_command: Shell command.

system_program: System program.

user_program: User installed or owned program.

system_info: Information regarding the system.

outfiles: Files outside a restricted space.

classloader: A ClassLoader or object responsible for loading dynamic classes.

library: System function or service library.

a_net_connection: Network connections to arbitrary hosts.

web_pages: WWW page.

web_session: A WWW browsing session.

names: User names, domain names, work-group names, etc.

pass_known: Well-known nonce encrypted with user password.

o_attributes: System-managed object attributes.

cpu: CPU time.

os: Operating System.

email: Electronic Mail.

netport: Network Port.

packets: Network Packets.

system_names: Internal system names (in control of the system).

device: A device in the system.

addr_mapping: Address mapping maintained by the system. i.e., an ARP cache.

4.3.2 Effect on Objects

The features that identify the effect that the vulnerability has on the objects affected are:

replaced: Contents are completely replaced.

changed: Can be written to or can be changed.

read: Can be read.

append: Information can be appended.

created: Can be created.

displayed: Can be displayed or revealed.

change_owner: Ownership can be changed.

change_permission: Permissions can be changed.

predictable: Is predictable or can be guessed.

executed: Can be executed in violation of expected policy.

loaded: Can be dynamically loaded and linked.

clear_text: Is transmitted or stored in clear text.

exhausted: Is exhausted.

crash: Crashes.

bound: Can be bound to in violation of expected policy.

exported: Can be exported for mounting.

mounted: Is mounted or attached.

locked: Can be locked.

debugged: Can be debugged or attached to with a debugger.

presented: Presented to the user in a console or terminal.

closed: Can be closed.

terminated: Can be terminated or killed.

4.3.3 Method or Mechanism Used

The features that identify the method or mechanism that is used to affect the objects are:

symlink: Program follows symbolic link or late binding link.

memcpy: Program uses `strcpy`, `sprintf` or `bcopy` to copy data to a stack buffer.

config: Configuration error.

back_ticks: Back ticks in parameter or input string.

special_chars: Special characters in input string.

dotdot: Uses ".." to climb up a directory tree past allowable bounds.

verify_fail: Code verifier allows to catch security exception when creating an object loader.

mod_name: Modifying compiled code to alter the name of objects.

mod_env: Modifying environment variables.

inherit_privs: Program inherits unnecessary privileges.

capability: System provides inappropriate capability.

hidden_mount: System provides hidden system mount point.

syscall_disclose: System call discloses sensitive information.

incorr_imp: Incorrect implementation given current environment (mistaken environmental assumption).

rel_paths: Program refers to relative paths.

incprot: System fails to implement the protection mechanisms correctly.

proxy: Program uses a trusted intermediary or proxy to bypass protection mechanisms.

coresymlink: A program dumps a core file that follows symbolic links or late binding link.

infloop: Program uses an infinite and tight loop that consumes resources.

criticalsect: Program fails to protect or isolate a critical section.

coredump: Program dumps a core-file that users can read.

4.3.4 Input Type

The features that identify the source of the input, if any, that is necessary for the object to be affected are:

env: Environment variable.

command: User command line option.

netdata: Network data.

store: Persistent store.

tempfile: Temporary file.

conffile: Configuration file.

datafile: Data file.

gecos: System User information (Name, phone number, etc.)

parameter: Parameter to a system call

libparameter: Parameter to a library call

floppy: Removable media

4.4 Chapter Summary

In Section 3.1 we argue that the basis for successful classifications are taxonomic characters that satisfy the properties of *objectivity*, *determinism*, *repeatability*, and *specificity*.

Features that will be used in analysis using tools such as data mining and data visualization, also need to satisfy the properties for taxonomic character as specified in section 3.1.2. This chapter presents examples of such taxonomic characters, and these can be used as a foundation for improved classifications and taxonomies. The taxonomic characters are observations or measurements that do not require the application of decision trees to avoid ambiguities resulting from the nodes having more than one *fundamentum divisionis*.

5 EXPERIMENTAL ANALYSIS OF SOFTWARE VULNERABILITIES

[Dorner 1996] argues that complex systems that fail often have four characteristics that make them especially prone to failure: complexity, intransparence[1], internal dynamics, and incomplete or incorrect understanding of the system. Although [Dorner 1996] analyzes highly dynamic systems such as nuclear power plants, management of entire cities, etc., many of these ideas can apply to the development of software because software systems have similar characteristics [Brooks 1995; Conte et al. 1986; Ghezzi et al. 1991]. The work presented in [Brooks 1995] makes the argument that "Since software construction is inherently a systems effort—an exercise in complex interrelationships—communication effort is great, and it quickly dominates the decrease in individual task time brought about by partitioning."

The complexity, intransparence, and volume of code in computer systems makes it difficult to find patterns and dependencies. As shown in this chapter, existing data mining techniques and feature-extracting tools can be used effectively to extract information that allows a better understanding of why and how vulnerabilities get introduced in computer systems.

Given a database of taxonomic characters for a representative population of software vulnerabilities, we can apply a variety of statistical and analysis tools to extract patterns, distributions, and regularities that may not be obvious at first sight. The analysis resultant from these tools may also result in *a posteriori* classifications that can provide insights into the nature of vulnerabilities, or can confirm *a priori* classifications by reconstructing them from the data.

[1] Intransparence is defined as the fact that an observer of the operation of a system cannot see the inner workings of a program. It contributes to the development of faulty software because developers can not *see* the execution of the program without the help of sophisticated monitoring tools, and sometimes even these tools are not useful because their presence alters the behavior of the system.

5.1 Experiment Hypothesis

As stated in Section 3.1.4, decision trees can be induced *a posteriori* from a set of data or examples [Quinlan 1986]. Other types of classifications can also be generated *a posteriori*, including neural networks [Hertz et al. 1991; Holsheimer and Siebes 1994], clusterings [Jain and Dubes 1988; Kukolich and Lippmann 1995], and pattern classification algorithms [Kukolich and Lippmann 1995]. We can also search for relationships, patterns, and regularities *a posteriori* from data or examples, applying machine learning algorithms in a process called data mining [Holsheimer and Siebes 1994; KDDSIF 1998; Hedberg 1995; Quinlan 1986].

The hypothesis for our experiments is that it is possible to find patterns and regularities in a collection of vulnerabilities by applying machine learning, statistical analysis, and visualization tools. These patterns and regularities contribute to a better understanding of why and how vulnerabilities get introduced in computer systems.

5.2 Experimental Setup

Data mining and statistical analysis techniques, applied to software vulnerabilities, require a collection of vulnerabilities with a set of characteristics (also known as features or dimensions) that satisfy the properties described in Section 3.1.2. In this dissertation, this collection is referred to as the **vulnerability database** [Krsul 1998].

The vulnerability database was built as a repository of software vulnerability information with a strong emphasis on quality and completeness of information. The information in the database is useful for a variety of purposes, including classifications of vulnerabilities, software engineering research on coding faults, and testing for the existence of vulnerabilities.

Without detailed knowledge of the distribution and characteristics of the data in question, we cannot know *a priori* the number of samples required for this database. The machine learning community, however, provides heuristics that help us estimate the number of samples required.

An **error in a classifier** is simply a misclassification [Weiss and Kulikowski 1991]. The **true error of a classifier** is statistically defined as the error rate of the classifier

on an asymptotically large number of new cases that converge in the limit of the actual population distribution [Weiss and Kulikowski 1991]. The **apparent error of a classifier** is the error rate on the sample cases that were used to design of build the classifier [Weiss and Kulikowski 1991].

[Weiss and Kulikowski 1991] argues that for classifiers and learning systems, a surprisingly small number of test cases are needed for test sample error to be essentially the true error: "at 50 test cases and a test sample error of 0%, there is a good chance that the true error is as large as 10%, while for 1000 test cases the true error rate is almost certainly below 1%". Traditionally, a small statistical sample size is about 30 samples [Weiss and Kulikowski 1991]. Many simplifying assumptions were made for this particular heuristic, including assumptions about the distribution of the test cases. These are assumed to be a good representation of the true population.

A heuristic that provides further insight as to the size of the state space is that linear-hyperplane learning systems need samples that must exceed two to three times the number of features [Duda and Hart 1973]. If we have 30 features in the state space, we must have at least 60-90 samples. Other heuristics can be found in [Weiss and Kulikowski 1991; Duda and Hart 1973; Quinlan 1986; Quinlan and Cemeron-Jones 1995; Breiman 1994; Freund and Schapire 1996].

Our initial estimate, based on these heuristics, is that we will need approximately 100–200 samples and 20–30 features. In Sections 5.2.1 and 5.2.3 we confirm that our data collection, based on this initial approximation, is a good distribution of known software vulnerabilities.

5.2.1 Sources for Data Collection

Most records in the database combine information from various sources. CERT, CIAC, and AUSCERT advisories are good indicators for the existence of a vulnerability but provide little information useful for the extraction of our taxonomic characters. The following sources were used in the collection of data for the database:

- Computer security mailing lists: BUGTRAQ, NTBUGTRAQ, IDS, Best of Security.
- Advisories: CERT, CIAC, AUSCERT, L0pht Security Advisories, Vendor Security Bulletins, Secure Networks Incorporated Security Advisories.

- Academic publications: [Dean and Wallach 1995; Dean et al. 1996; Cohen 1997b; 1997a; Garfinkel and Spafford 1996; Bishop and Dilger 1996; Kumar 1995; Aslam 1995; Kumar et al. 1995]
- Security tools: COPS [Farmer and Spafford 1991], SATAN, ISS.
- Hacker toolkits.
- Private vulnerability databases: Michael Dilger, Eric Miller, Eugene Spafford
- Private electronic mail correspondence: Edward Felten of Princeton University.
- Vulnerability Databases: [Internet Security Services 1998; INFILSEC Systems Security 1998; Firosoft Consulting 1998; VDBKAO 1998; VDBFIR 1998; VDBBOD 1998; VDBLEG 1998; VDBFKI 1998; VDBBUG 1998; Stout 1998; VDBNZE 1998; VDBELI 1998; VDBDOP 1998; VDBEXP 1998; VDBNN1 1998; VDBNN2 1998; VDBNN3 1998; VDBNN4 1998; VDBNN5 1998; VDBNN6 1998].

5.2.2 Database Structure

The database was designed to be a superset of all the databases and sources of vulnerability information enumerated in Section 5.2.1, includes the taxonomic characters developed in Chapter 4, and the classification developed in Section 6.1. Fields from previous databases that were shown to be ambiguous in Section 3.2 were modified as described in Appendix C.

The database is a flat file that has the following schema:

Identification Section	
`title`	Title of the vulnerability

Modification History	
`modifications`	Person(s) who have modified this record, the date of modification, and the modifications made.

Description and impact	
desc	Description of the vulnerability.
indirect_impact	Ultimate consequences of an attack exploiting the vulnerability by a threat agent. See Figure C.1.
direct_impact	Rather than the ultimate impact of the vulnerability, the direct or immediate impact. See Figure C.2.
impact_verbatim	Textual description of the impact of exploiting the vulnerability.

Information Regarding the Source of the Information	
source_address	Detailed information on the source of the information. The WWW address, email address, books, etc. from which the information was gathered.

System Identification	
system	System(s) vulnerable.
system_version	System Version.
vendor	System Vendor.
system_verbatim	Additional textual description of system.
os_type	Type of operating systems affected. See Figure C.6.

Application Information	
app	Application that contains the vulnerability.
app_version	Application Version.
app_verbatim	Long description of applications that contain vulnerabilities.

References	
advisory	Advisory/ies that warn/describe about the vulnerability.
reference	References to the vulnerability in literature or in the net.
related_docs	Documents that describe the vulnerability, related to the vulnerability or that are useful in the analysis of the vulnerability.

Detailed Analysis, Detection Techniques, and Fixes	
`analysis`	A detailed analysis of the vulnerability.
`core_vulner`	If the vulnerability is in a piece of code, the smallest piece of code that still has the vulnerability.
`detection`	Method of detecting that the vulnerability is being exploited.
`fix`	A fix that can be used to eliminate the vulnerability.
`test`	Method that can be used to detect whether the vulnerability is present in a system.
`workaround`	A temporary workaround for the vulnerability. Used until a patch can be applied.
`patch`	A patch or a series of patches that can be used to eliminate the vulnerability.

Detailed Information About Exploitation	
`exploit`	Reference to exploit scripts or programs.
`ease_of_exploit`	How easy is it to exploit the vulnerability.
`idiot`	IDIOT Pattern used to detect the exploitation of the vulnerability. See [Crosbie et al. 1996].
`access_required`	Access is required for the exploitation. See Figure C.3.
`complexity_of_exploit`	Complexity of the exploitation of the vulnerability. See Figure C.4.

Source code and pointers to source code for the systems that contain the vulnerabilities.	
`system_source`	Source code or a pointer to the source code for the system that contains the vulnerability.

Fault Classification	
`envass`	Environmental assumptions that were made by designers or programmers that, if they were to hold, would make the program correct. See Section 4.2.
`class`	Aslam Classification. See Figures B.1 and B.2.

Category and Component Classification	
`category`	The system or component to which the vulnerability belongs to. See Figure C.5.

Identification of Nature of the Vulnerability	
`nature_object`	The object fundamentally affected by the vulnerability. See Section 4.3.1.
`nature_effect`	The effect that the vulnerability has on the object. See Section 4.3.2.
`nature_method`	The method or means by which the object is affected. See Section 4.3.3.
`nature_method_input`	The type of input, if any, that leads to the effect. See Section 4.3.4.

Verification of Vulnerability	
`verif`	Person or entity who verified the vulnerability. Verification should imply that the vulnerability is know to exist and had been exploited or verified by the person named.

Identification of Policy Violation	
`policyvio`	Expected Policy Violated by the Vulnerability. These policies need not be formally specified and are the expectation that users feel have been violated.

Identification of Environmental Factors	
`environment`	What environmental conditions contribute to the vulnerability? What assumptions are made about the environment that don't hold? What about the environment makes this vulnerability possible?
`features`	What other characteristics and features are relevant for the understanding of the vulnerability?

Identification of the Nature of Threat (See Section 4.1).	
thac_observe	The vulnerability can result in a user observing objects, data, etc., in violation of expected policy. Value can be yes, no, ?, or NA.
thac_destroy	The vulnerability can result in a user destroying objects, data, etc., in violation of expected policy. Value can be yes, no, ?, or NA.
thac_modify	The vulnerability can result in a user modifying objects, data, etc., in violation of expected policy. Value can be yes, no, ?, or NA.
thac_create	The vulnerability can result in a user creating objects in violation of expected policy. Value can be yes, no, ?, or NA.
thac_cavail	The vulnerability can result in the change of availability of the system. Value can be yes, no, ?, or NA.
thac_disclose	The vulnerability can result in the disclosure of information in violation of expected policy. Value can be yes, no, ?, or NA.
thac_exec	The vulnerability can result in the execution of applications in violation of expected policy. Value can be yes, no, ?, or NA.
thac_misrep	The vulnerability can result in misrepresentation of information. Value can be yes, no, ?, or NA.
thac_repudiate	The vulnerability can result in repudiation of information. Value can be yes, no, ?, or NA.
thac_integrity	The vulnerability can result in change of integrity of the system. Value can be yes, no, ?, or NA.
thac_conf	The vulnerability can result in the loss of confidentiality of information. Value can be yes, no, ?, or NA.

5.2.3 Data Characteristics

As of March 18, 1998, the vulnerability database contained information relevant to 210 vulnerabilities. These entries include, but are not limited to, 57 CERT advisories[2], seven CERT vendor-initiated bulletins[3], and 21 AUSCERT advisories[4].

Not all CERT advisories describe software vulnerabilities. Some CERT advisories, for example, are descriptions of trojan horses, warnings regarding hacking techniques, descriptions of several vulnerabilities grouped together, warnings about rumors and alleged attacks, reports of security probes, warnings about false emails, reports of social engineering techniques, descriptions of email scams, virus reports, reports on Internet intruder activities, configuration guidelines, and checksums for files. Of the 175 advisories published by CERT by March of 1998, 51 do not describe vulnerabilities. We have detailed documentation for approximately 50% of the remaining advisories.

5.2.4 Data Distribution

The minimum information required for each record in the database is a record identifier and a title, although there are no records in the database that have so little information.

Let the **size of a record** be the number of fields that have information for that record. The size of a record depends on the amount of detailed information that can be found for each vulnerability. A CERT advisory, for example, typically is sufficient to fill fifteen fields in the database (`title`, `modifications`, `desc`, `impact_verbatim`, `source_address`, `system`, `system_version`, `vendor`, `system_verbatim`, `os_type`, `app`, `app_version`, `app_verbatim`, `advisory`, and `workaround`). The other fields require more detailed information regarding the vulnerability than what advisories—CERT, CIAC, AUSCERT, etc.—provide.

[2]The CERT advisories included or referenced in the database are CA-93:01, CA-93:02, CA-93:06, CA-93:09, CA-93:17, CA-94:02, CA-94:06, CA-94:09–CA-94:11, CA-95:02, CA-95:08, CA-95.10, CA-95:13–CA-95:16, CA-96.04, CA-96.05, CA-96.08, CA-96.12, CA-96.13, CA-96.15–CA-96.19, CA-96.21, CA-96.22, CA-96:24, CA-96.25, CA-96.27, CA-97.01–CA-97.06, CA-97.08–CA-97.10, CA-97.12–CA-97.17, CA-97.19, CA-97.20, CA-97.27, CA-97.28, and CA-98.01–CA-98.04

[3]The CERT vendor initiated bulletins in the database are VB-96.04, VB-96.05, VB-97.01, VB-97.02, VB-97.05, VB-97.06, and VB-97.13

[4]The AUSCERT advisories referenced in the database are AA-96.03, AA-96.04, AA-96.06, AA-96.11–AA-96.14, AA-96.16, AA-97.02, AA-97.03, AA-97.07, AA-97.12–AA-97.14, and AA-97.18–AA-97.23.

As shown in Figure 5.1, the smallest record in the database has only 6 fields and the largest 46. The median (26) and average (30) are well above the fifteen fields mentioned. Hence the records in the database have, on average, more detailed information than simple high-level descriptions of vulnerabilities.

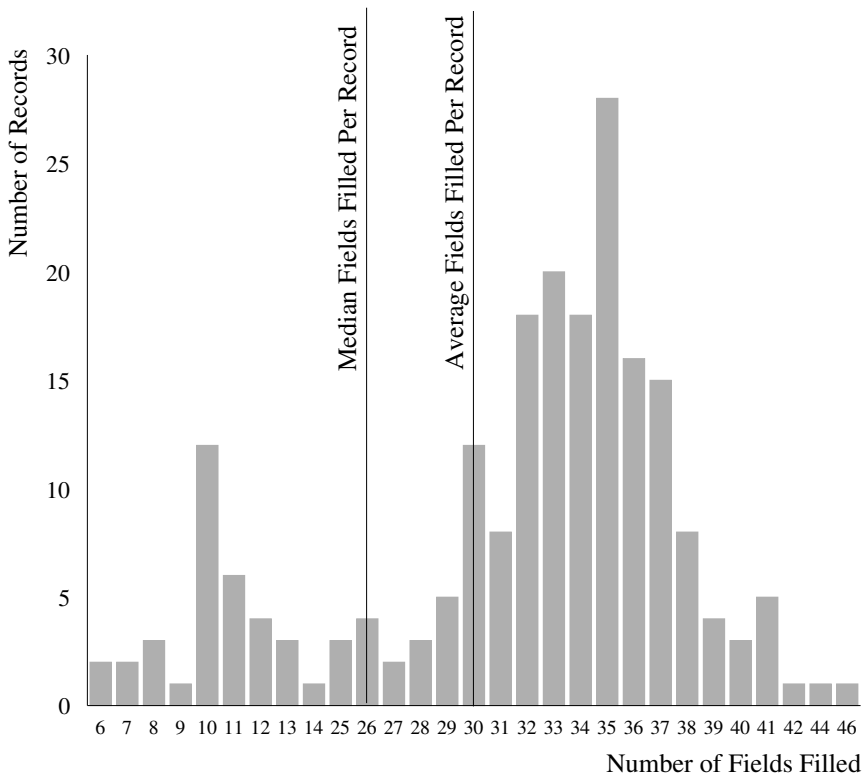


Figure 5.1: Distribution of filled fields in the database

The non-textual data in the database (i.e. lists and choices), as shown next, has variation and does not present hot-spots (where all the records have a single value).

The fields `indirect_impact`, `direct_impact`, `os_type`, `access_required`, `category`, and `complexity_of_exploit` are filled in 175 of the 210 records in the database. Figures 5.2 and 5.3 show scatter plots for these fields.

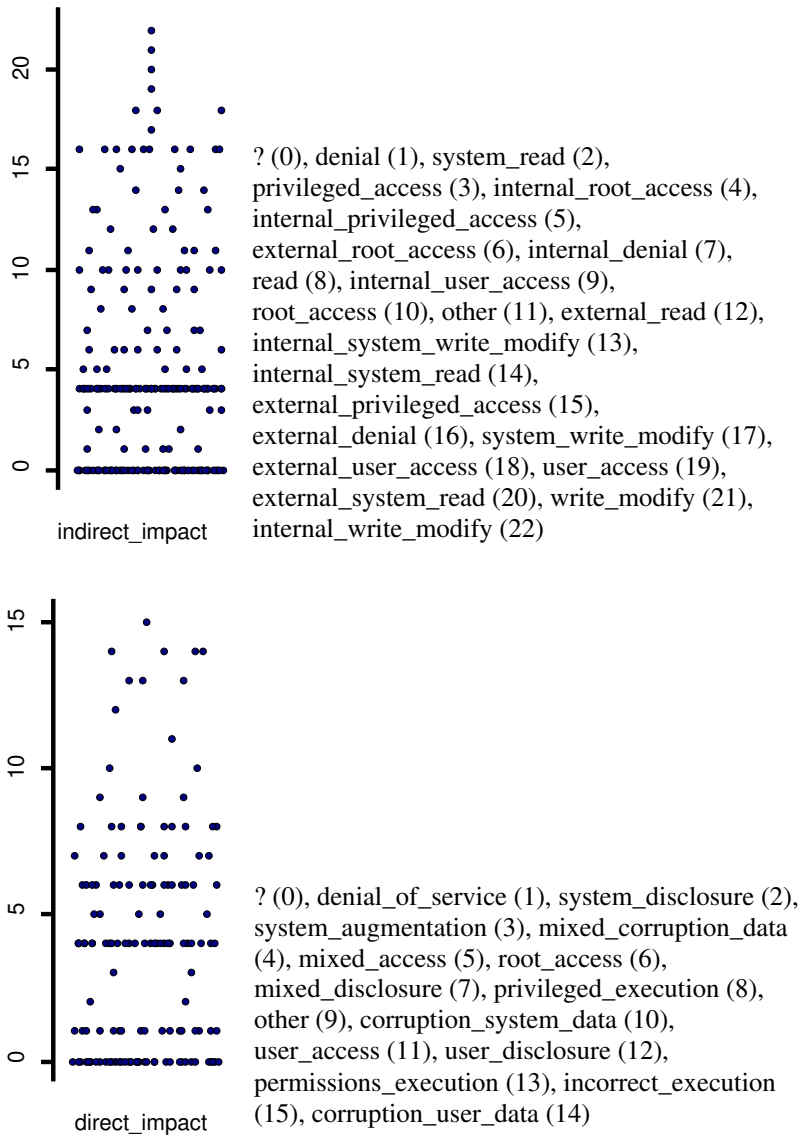


Figure 5.2: Scatter plot for fields `indirect_impact` and `direct_impact`.

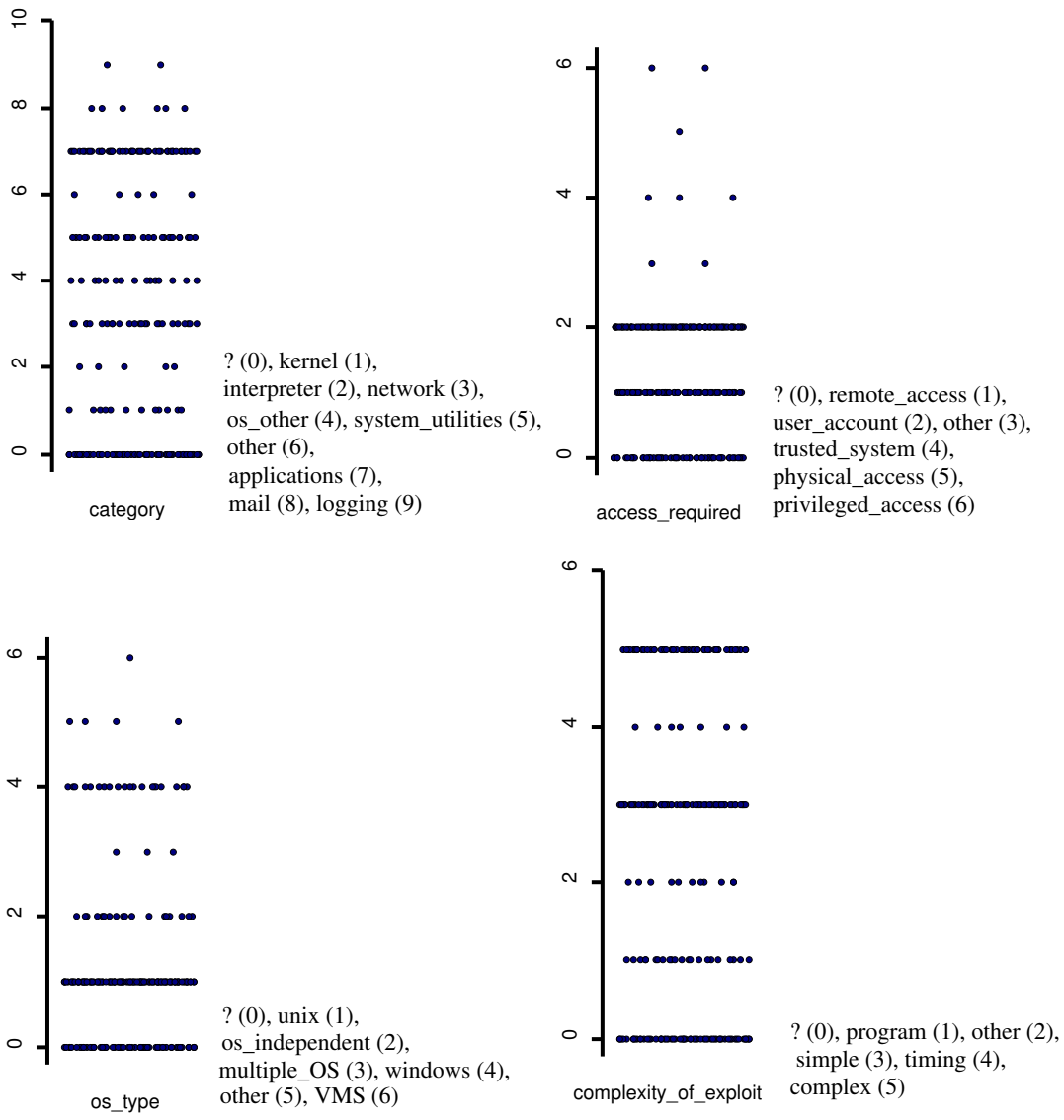


Figure 5.3: Scatter plot for fields `os_type`, `access_required`, `category`, and `complexity_of_exploit`.

Figure 5.4 show a distribution plot for the Nature of Threat identification fields. The nature of threat identification fields are filled for 175 records in the database.

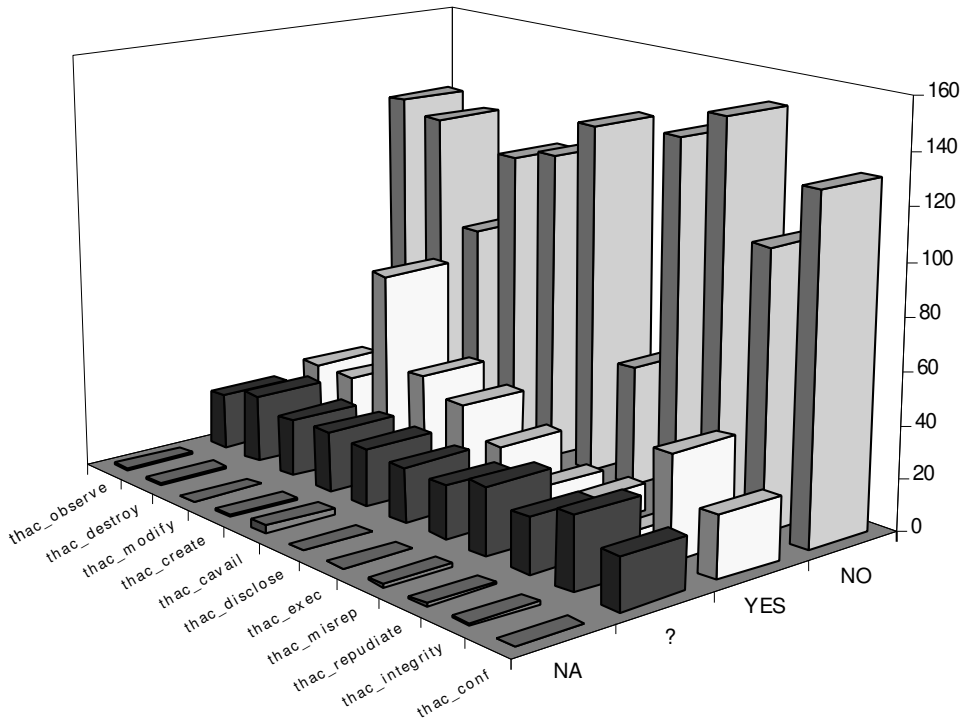


Figure 5.4: Distribution Plot the Nature of Threat Features

Figures 5.5, through 5.10 show distribution plots for the fields environmental assumption, identifications of the nature of the vulnerability, and system. 170 records in the database have information regarding the system field, 148 records have information regarding the nature of the vulnerability fields, and only 61 records in the database have information regarding the environmental assumptions field.

The `envass` field is one of the most difficult fields to fill because it requires detailed information regarding the environment in which the program runs, the source code for the vulnerable system, and information regarding the exploitation of the vulnerability.

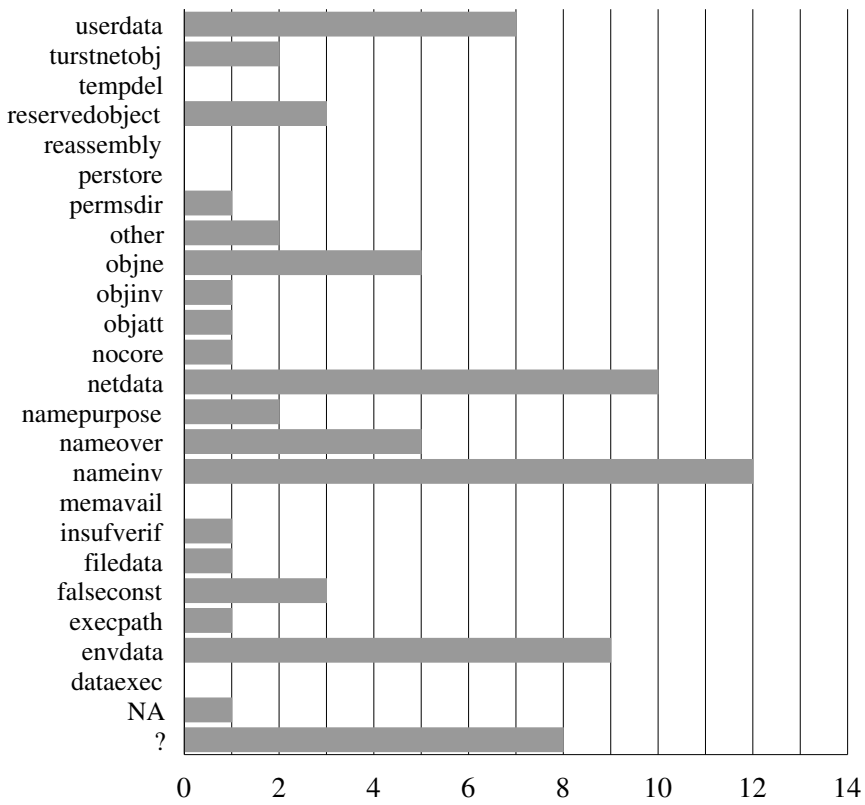


Figure 5.5: Distribution Plot for Environmental Assumption Features

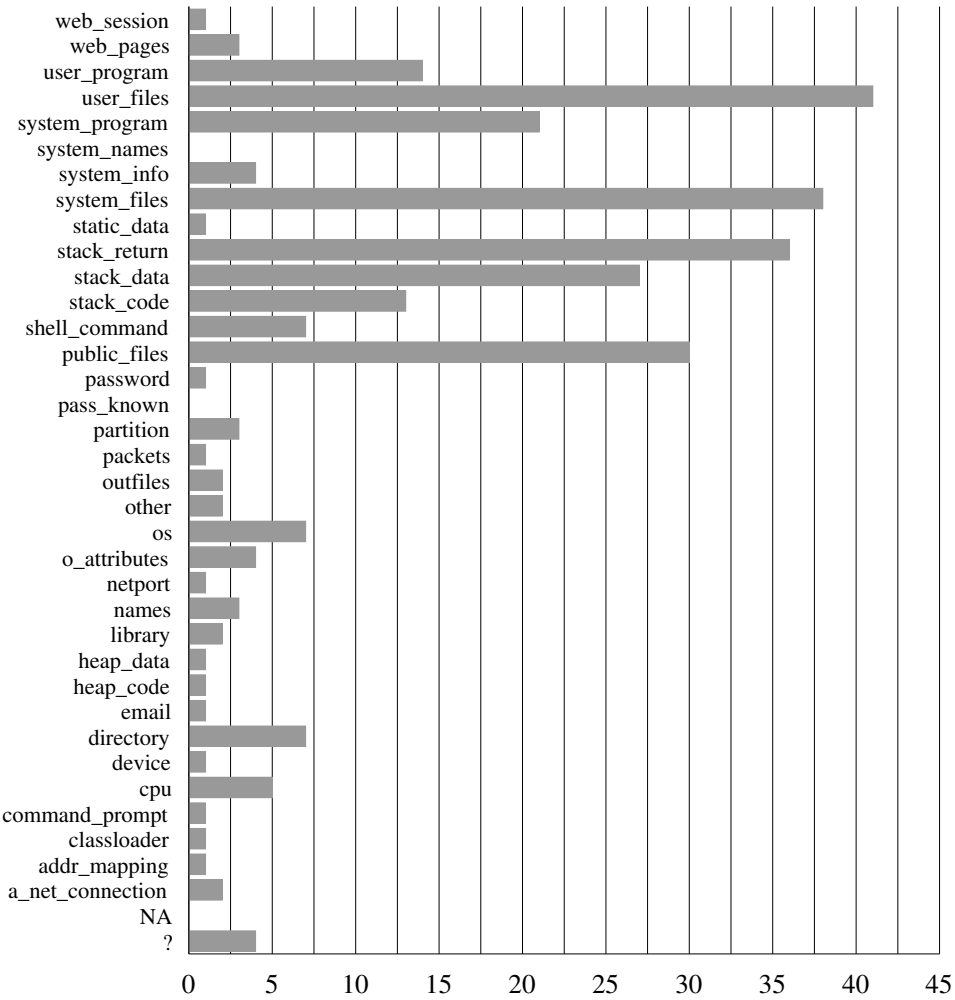


Figure 5.6: Distribution plot for the Nature of Vulnerability feature Object Affected

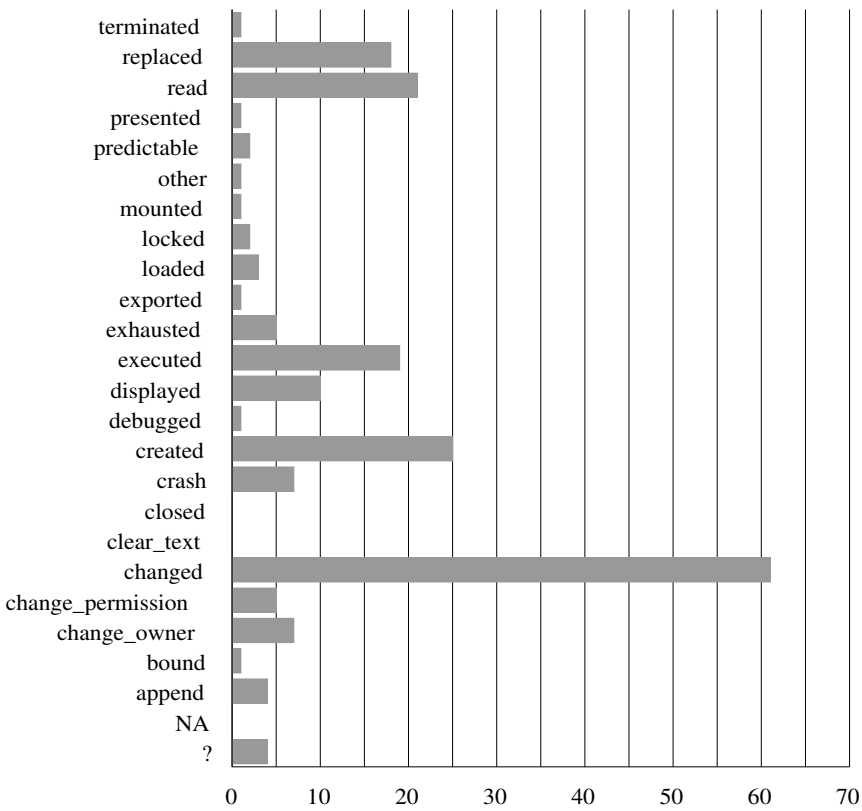


Figure 5.7: Distribution plot for the Nature of Vulnerability feature Effect on Object

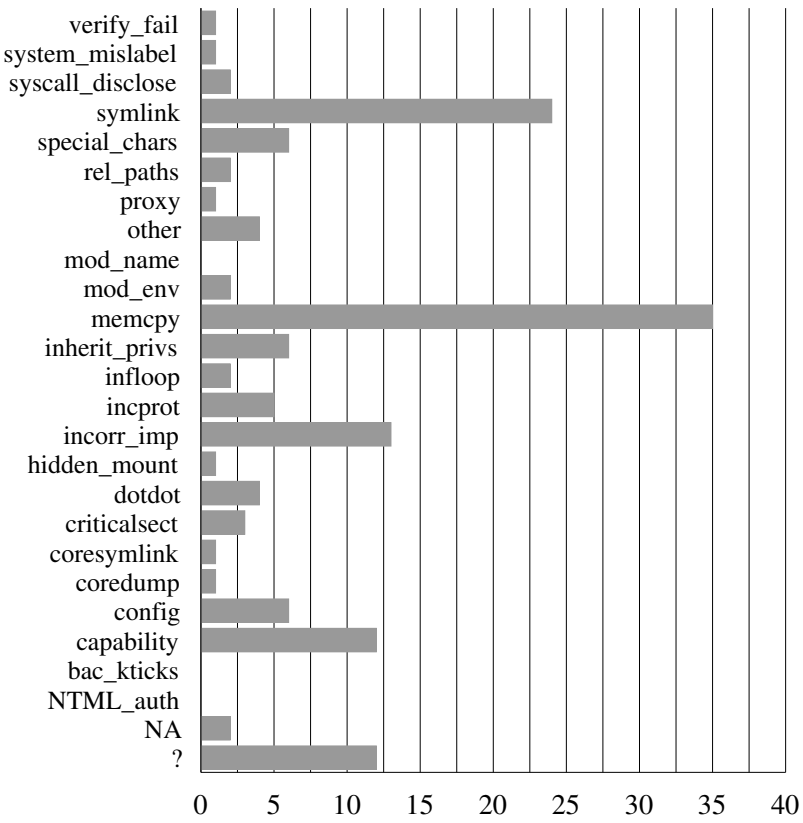


Figure 5.8: Distribution plot for the Nature of Vulnerability Method feature

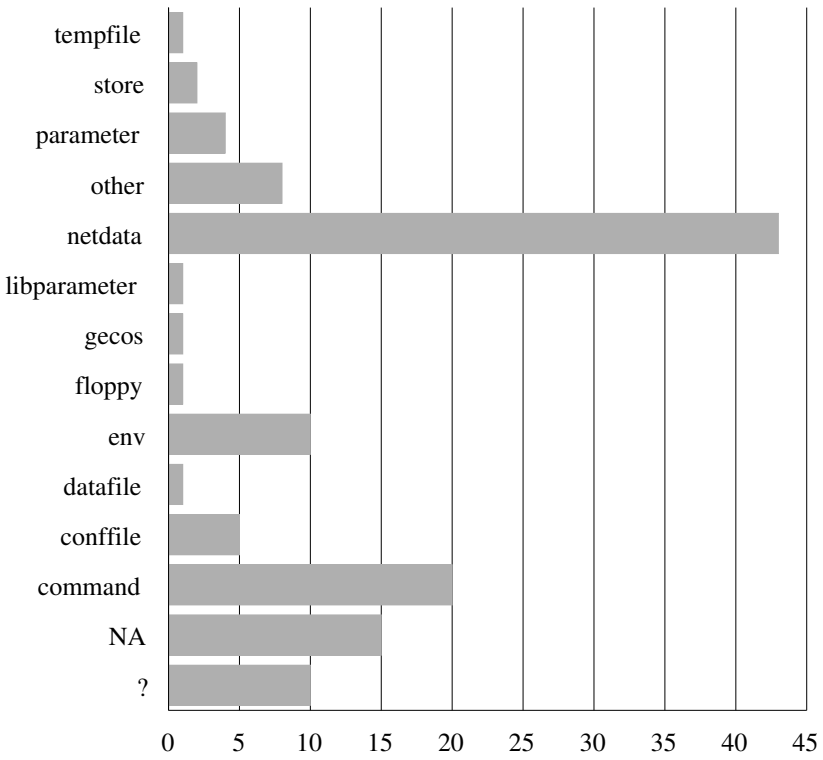


Figure 5.9: Distribution plot for the Nature of Vulnerability Method Input feature

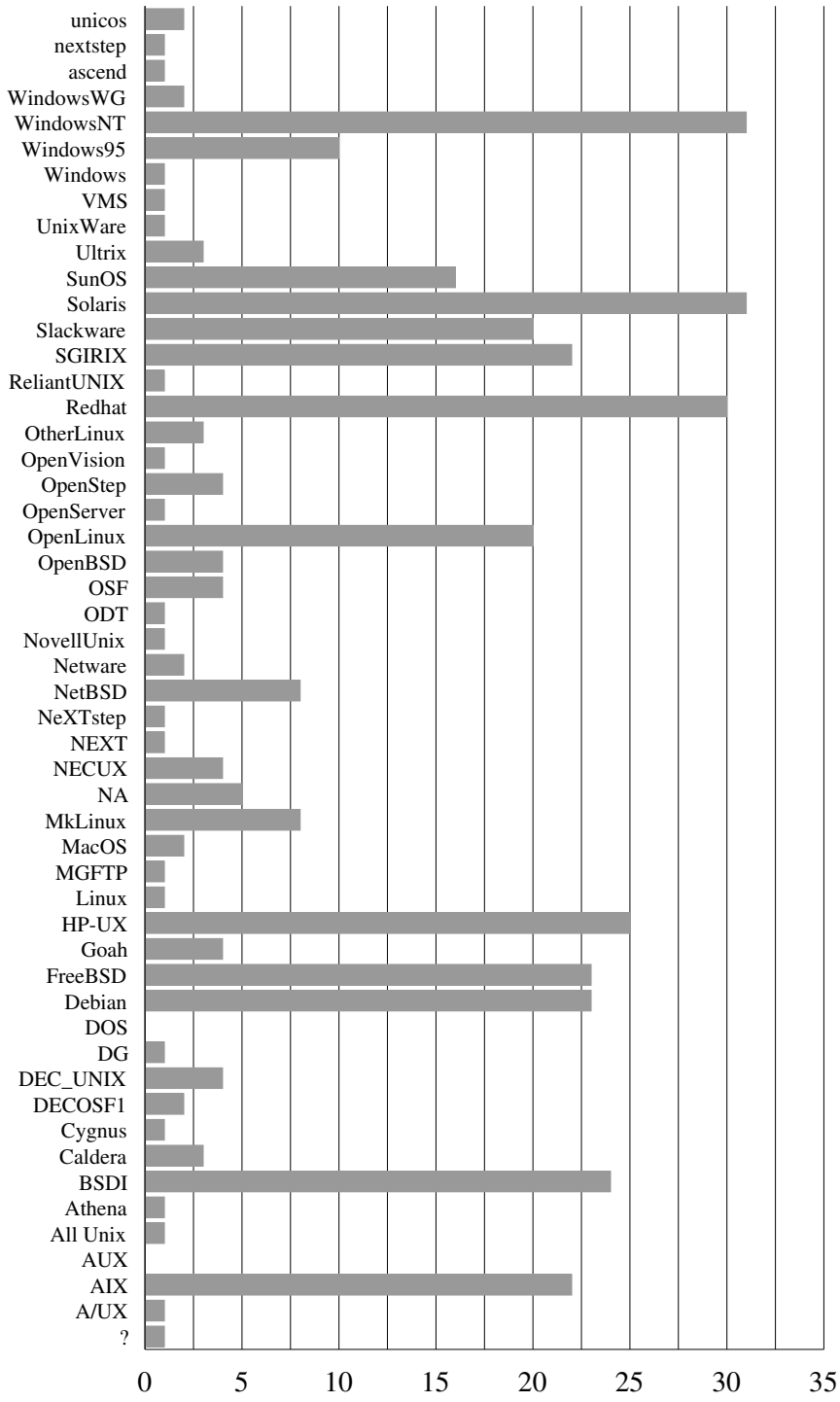


Figure 5.10: Distribution plot for the System Features

5.3 Experiments

In this section we present the application of three analysis techniques to the data described in Section 5.2. The first is an example of a statistical analysis technique similar to traditional association analysis [Agrawal and Srikant 1994] and graphical modeling [Edwards 1995]. The second is an example of a machine learning technique. The last technique demonstrated is an example of graphical visualization of multivariate data.

5.3.1 Co-word Analysis

Co-word analysis is a content analysis technique that is effective in mapping the strength of association between keywords in textual data. Co-word analysis reduces a space of descriptors (or keywords) to a set of network graphs that effectively illustrate the strongest associations between descriptors [Coulter et al. 1997; Whittaker 1989].

Co-word analysis is an example of a graphical modeling technique that applies some of the ideas of association analysis [Edwards 1995; Kaufman and Rousseeuw 1990]. Graphical modeling is a variant of statistical modeling that uses graphs to display models. "In contrast to most other types of statistical graphics, the graphs do not display *data*, but rather and interpretation of the data, in the form of a *model*... Graphs have long been used informally... to visualize relations between variables." [Edwards 1995].

We have chosen co-word analysis because the models that can be generated with this technique illustrate associations between keywords by constructing multiple networks that highlight associations between keywords, and where associations between networks are possible.

Co-Word Analysis Algorithm Description

In this section we describe the algorithms used in [Coulter et al. 1997] for constructing the networks that highlight the strongest associations between keywords, modified to fit our needs.

For our purposes, the set of descriptors corresponds to the keywords used in the vulnerability database as shown in Section 5.3.1. Two keywords co-occur if they are used in the same records in the database. Let c_k represent the number of occurrences of the keyword

k in the entire database. Let c_{ij} represent the number of co-occurrences of keywords i and j. The strength S of association between descriptors i and j is given by equation 5.1:

$$
\begin{array}{l}
S : \textbf{integer} \times \textbf{integer} \times \textbf{integer} \rightarrow \textbf{real} \\
\textbf{fun} \quad S\,(c_{ij}, c_i, c_j) ::= \\
\qquad \textbf{if} \ ((c_i \leq 0) \vee (c_j \leq 0)) \ \ \textbf{then} \\
\qquad\qquad S := \textit{undefined}; \\
\qquad \textbf{else} \\
\qquad\qquad \Rightarrow S \textit{ is in the range } 0 \leq S \leq 1 \Leftarrow \\
\qquad\qquad S := \frac{c_{ij}^2}{c_i c_j}; \\
\qquad \textbf{fi} \\
\textbf{nuf}
\end{array}
\tag{5.1}
$$

Keywords that often appear together will have strengths closer to 1, and keywords that appear together infrequently will have strengths closer to 0. In co-word analysis, strengths of larger value constitute the links between nodes in a network depicting the strongest associations in the database.

The co-word algorithm uses two passes through the data to produce the desired networks. The first pass constructs the networks depicting the strongest associations, and links added in this pass are called *internal links*. The second pass adds to these networks links of weaker strengths that form associations between networks. The links added during the second pass are called *external links*.

Note that two keywords that appear infrequently in the database but always appear together, will have larger strength values than keywords that appear many times in the database almost always together. Hence, possibly irrelevant or weak associations may dominate the network. A solution to this problem—incorporated into the algorithm described in this section—is to require that only the keyword pairs that exceed a minimum co-occurrence are considered potential links while building networks during the first pass of the algorithm.

During the first pass, the link that has the largest strength is selected first, its nodes becoming the starting nodes of the first pass-1 network. Other links and their corresponding nodes are added to the graph using a breath-first search on the strength of the links (i.e. the strongest link connecting a node that is not in any graph to the graph being constructed is added first), until there are no more links that exceed the co-occurrence threshold, or a maximum pass-1 link limit is exceeded. The next network is generated in a similar manner starting with the link with the largest strength that is not in any existing graph.

During the second pass of the algorithm we add nodes to each existing graph, choosing the links that have the largest strength that exceed the co-occurrence threshold, and that are in some pass-1 network.

As described in [Coulter et al. 1997], the algorithm for the generation of the networks is as follows:

1. Select a minimum for the number of co-occurrences, c_{ij}, for descriptors i and j, select maxima for the number of pass-1 links, and select maxima for the total (pass-1 and pass-2) links;
2. Start pass-1;
3. Generate the highest S value from all possible descriptors to begin a pass-1 network;
4. From that link, form other links in a breadth-first manner until no more links are possible due to the co-occurrence minima or to pass-1 link or node maxima. Remove all incorporated descriptors from the list of subsequent available pass-1 descriptors;
5. Repeat steps 3 and 4 until all pass-1 networks are formed; i.e., until no two remaining descriptor pairs co-occur frequently enough to begin a network;
6. Start pass-2;
7. Restore all pass-1 descriptors to the list of available descriptors;
8. Start with the first pass-1 network.
9. Generate all links to pass-1 nodes in the current network to any pass-1 nodes having at least the minimal co-occurrences in descending order of S value; stop when no remaining descriptor pairs meet the co-occurrence minima, or when the total link maxima is met. Do not remove any descriptors from the available list;
10. Select the next succeeding pass-1 network, and repeat step 9.

Networks are interconnected by pass-2 links. The **centrality** of a network measures the degree of interaction to other networks and is defined as the square root of the sum of the squares of the S values of the pass-2 links of the network. The **density** of a network measures the internal strength of the network and is defined as the mean of the S values of the pass-1 links of the network.

Isolated Networks are those that have low centrality values. **Principal Networks** are those that have high centrality and high density values.

Selection of Keywords

Every field in the database described in Section 5.2 can be used as a keyword, or can be transformed to a series of keywords for co-word analysis by applying the following rules:

1. If a field in the database can have the values yes, no, ?, and NA, then a keyword with the name of the field is generated if the value of the field is yes.
2. If a field in the database is running text then a keyword with the name of the field name followed by the string "`Defined?`" is generated if the field is not empty.
3. If the field in the database can have a single value from a list then a keyword with the name of the field followed by the value of the field is generated.
4. If the field in the database can have a list of values from a well defined set, then for every value in the field we generate a keyword with the name of the field followed by the value of the field.

Applying these rules to the database we can generate the keywords for the database as shown in Table 5.1. The keywords in group K1 are present in every record in the database and hence are not useful in our analysis. The keywords in group K2 do not satisfy any of the requirements for taxonomic characters as stated in Section 3.1.2 and are not reliable indicators. The keywords in group K9, as shown in Section 3.2.1, also fail to satisfy these properties. The keyword in group K11, as shown in Section 3.2.1, is not a reliable indicator.

Analysis can be performed with the remaining groups: K3, K4, K5, K6, K7, K8, K9, K10, K12, K13, K14, K15, and K16.

Table 5.1: Keywords used in the co-word analysis run.

Group	Keywords	Source Field(s)	Rule
K1	2	`title` and `modification`	2
K2	25	`desc`, `impact_verbatim`, `source_address`, `system_version`,`system_verbatim`, `app_version`,`app_verbatim`, `advisory`, `reference`, `related_docs`, `analysis`, `core_vulner`, `detection`, `fix`, `test`, `workaround`, `patch`, `exploit`, `ease_of_exploit`, `idiot`, `system_source`, `verif`, `policyvio`, `environment`, and `features`	2
K3	11	`indirect_impact`	3
K4	17	`direct_impact`	3
K5	11	`thac_observe`, `thac_destroy`, `thac_modify`, `thac_create`, `thac_cavail`, `thac_disclose`, `thac_exec`, `thac_misrep`, `thac_repudiate`, `thac_integrity`, and `thac_conf`	1
K6	44	`system`	4
K7	9	`os_type`	3
K8	6	`access_required`	3
K9	5	`complexity_of_exploit`	3
K10	24	`envass`	4
K11	40	`class`	4
K12	9	`category`	3
K13	35	`nature_object`	4
K14	23	`nature_effect`	4
K15	23	`nature_method`	4
K16	12	`nature_method_input`	4

Experimental Results

The results presented in this section correspond to the co-word algorithm presented in Section 5.3.1 applied to the keywords generated for groups K3, K4, K5, K6, K7, K8, K9, K10, K12, K13, K14, K15, and K16. This choice of keyword groups produces networks with sufficient variability in centrality and density to demonstrate the full range of possibilities of the technique.

There are three parameters that can be changed in the generation of networks with the co-word analysis tools. These are the number of pass-1 links, the total number of links in the network, and the minimum co-occurrence necessary for a link to be included in a network.

In the co-word analysis tool, raising the minimum co-occurrence requirement produces smaller and fewer networks, revealing the most prominent groups of related keywords. Lowering the minimum co-occurrence produces larger and more networks that show more subtle relationships among keywords.

In both cases, the most prominent networks will be similar because the first network is created in descending co-occurrence strength, and keywords that co-occur frequently will be added first in all cases. Hence, it is frequently desirable to choose a small minimum co-occurrence value.

Similarly a higher value for the number of pass-1 links and the total number of links in the network will produce larger and fewer networks. A lower value will produce smaller and more networks. Larger networks reveal more complex relationships among keywords and are frequently difficult to interpret. Small networks produce simpler networks that reveal smaller groups of related keywords.

The values chosen for the results presented here are as follows: A minimum co-occurrence of five (5), a maximum number of pass-1 links of nine (9), and a maximum number of pass-1 and pass-2 links of eighteen (18). These values were chosen after several trials because they produce networks that have both high and low values for density and centrality, so we can demonstrate the meaning of each combination.

Ten networks were generated by the co-word analysis tool and are shown in figures 5.12 through 5.21. Figure 5.11 shows the distribution of their centrality and density values. Each network has been labeled according to the interpretations that follow.

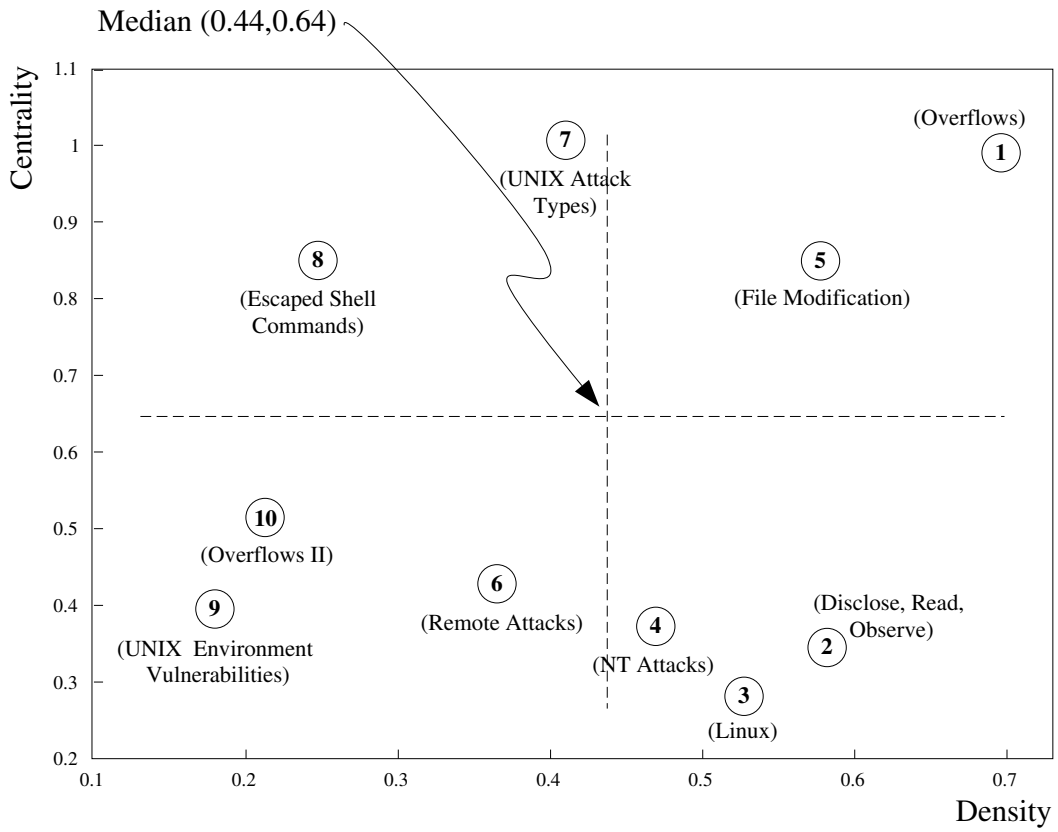


Figure 5.11: Plot of Centrality vs. Density for the results of co-word analysis for the vulnerability database

In these networks, solid lines connecting nodes represent pass-1 links and dotted lines represent pass-2 nodes. Each link is labeled with a triple <*number*, *co-occurrence*, *strength*> that indicates the order in which the links were added (1 corresponds to the first link), the co-occurrence for the key pair, and the strength of the link.

The principal networks of our database, and for the given groups, are networks 1 and 5. These are shown in Figures 5.12 and 5.16. These networks have the highest centrality and density values and hence represent the more predominant relationships among the keywords in our data.

Network 1, or the *Overflows* network, shows that one of the predominant groups of related keywords corresponds to a group that is commonly called "buffer overflows" by the computer security community. These are vulnerabilities where the object that is affected

by the vulnerability is the stack (both the return address and the content of the stack), where the effect is that the contents of the stack are changed using an operation that copies bytes from one memory location to another[5], and where the impact is root access in UNIX machines. Other links indicate that frequently the applications where this happens are system utilities.

The name "Overflows," however, does not necessarily imply that a buffer was overflowed in the program that contains the vulnerability. As shown in the following example, it is possible to create a program that would fit in this category (i.e. would have the keywords that indicate memory changes to the stack with a memory copy operation) without an array to overflow. We note, however, that we found no vulnerability that matches this scenario.

Example 5.1: The program that follows illustrates how a program that can provide an attacker an index into arbitrary stack memory can be vulnerable to the same problem without overwriting the program's local memory or altering anything other than the return address in the stack and the portion of memory that will be used to store the code to be executed. The program segment was extracted from a project for a graduate-level operating system course and its function is to allow the programmer to change the value of debugging flags without having to recompile the code.

```
main() {
  int  dbg1, dbg2, dbg3, dbg4, numiter,index, i, j;
  FILE *fp;

  /* Read from a file the values of the debugging variables
     as a series of (position, value) pairs: (1,5) would set dbg1 to 5
     and (4,0) would clear the dbg4 flag. */
  if((fp = fopen("conf","r"))!=NULL){
    /* How many flags to change? */
    fscanf(fp,"%d",&numiter);
    printf("numiter = %d\n",numiter);
    for(i=0;i<numiter;i++) {
      fscanf(fp,"%d%d",&index,&j);
      *(&dbg1-index+1) = j;
    }
  }
}
```

□

[5]Recall from Section 4.3.2 that the `memcpy` feature is used to indicate that the mechanism that is used to affect the object are the `strcpy`, `sprintf` or `bcopy` system calls.

Network 5, or the *File Modification* network, shows that UNIX files (public, user, and system) are modified or replaced taking advantage of problems with late binding links (or symbolic links), and that the effect frequently is root access obtained by a user who has an account in the system. This network identifies the group of vulnerabilities commonly referred to as "symbolic link vulnerabilities" by the computer security community.

Note that these groups were identified using mostly the taxonomic characters defined in Chapter 4. Hence, the co-word analysis tool can reconstruct classes of vulnerabilities from the taxonomic characters themselves.

Networks 7 and 8, or the UNIX *Attack Types* and *Escaped Shell Command* networks, have low density and high centrality values, indicating that relationships in the network are weak and that the features (or keywords) used in this network are also a part of many other networks. Network 7 shows that UNIX vulnerabilities lead to internal root access, that frequently a user account is needed to exploit the vulnerability, that the objects affected are system files and system utilities, and that buffer overflows are frequent and have complex exploits (as defined by the complexity of exploit feature defined in Section C.4).

Network 8 identifies the class of vulnerabilities where users use escape characters in shell commands to run applications or system programs in violation of policies.

The isolated networks are 2, 3, 4, 6, 9, and 10. These networks identify relationships with keywords that have low centrality values and hence the keywords are infrequently used in other networks. Isolated networks with high density values indicate strong relationships among keywords in isolated groups, and point to dominant features of the database.

The strong isolated networks are 2, 3, and 4, or the *Disclose–Read–Observe*, *Linux*, and *NT Attacks* networks.

Network 2, 3, and 9 are examples of networks that identify sets of keywords that have such high correlations that can effectively be compressed into a single feature. Network 2 identifies that reading, disclosing, observing, displaying, and threats to confidentiality are tightly related. Network 3 reveals that vulnerabilities that appear in one variant of Linux frequently appear in other variants. Network 9 reveals a similar result for the UNIX systems BSDI, AIX, NetBSD, FreeBSD, HP-UX, SunOS, and Linux.

Those results do not imply that the features should be collapsed, and the interpretation of these high correlations is not unique. The result of network 3 is intuitive because many

Linux distributions share a common code base and Linux system utilities frequently cross-compile for multiple Linux distributions.

Network 9 shows that frequently vulnerabilities apply to more than one variant of UNIX system that do not share a common code base. We note that system utilities for UNIX are frequently designed so they will cross-compile for multiple variants of UNIX, and tools such as Imake were designed specifically to allow machine dependencies (such as compiler options, alternate command names, and special make rules) to be kept separate from the descriptions of the various items to be built [Oram and Talbott 1993].

Network 2 shows that vulnerabilities that result in the disclosure of information are frequently exploited remotely and are either operating system independent (i.e. Java), or exist in Windows NT.

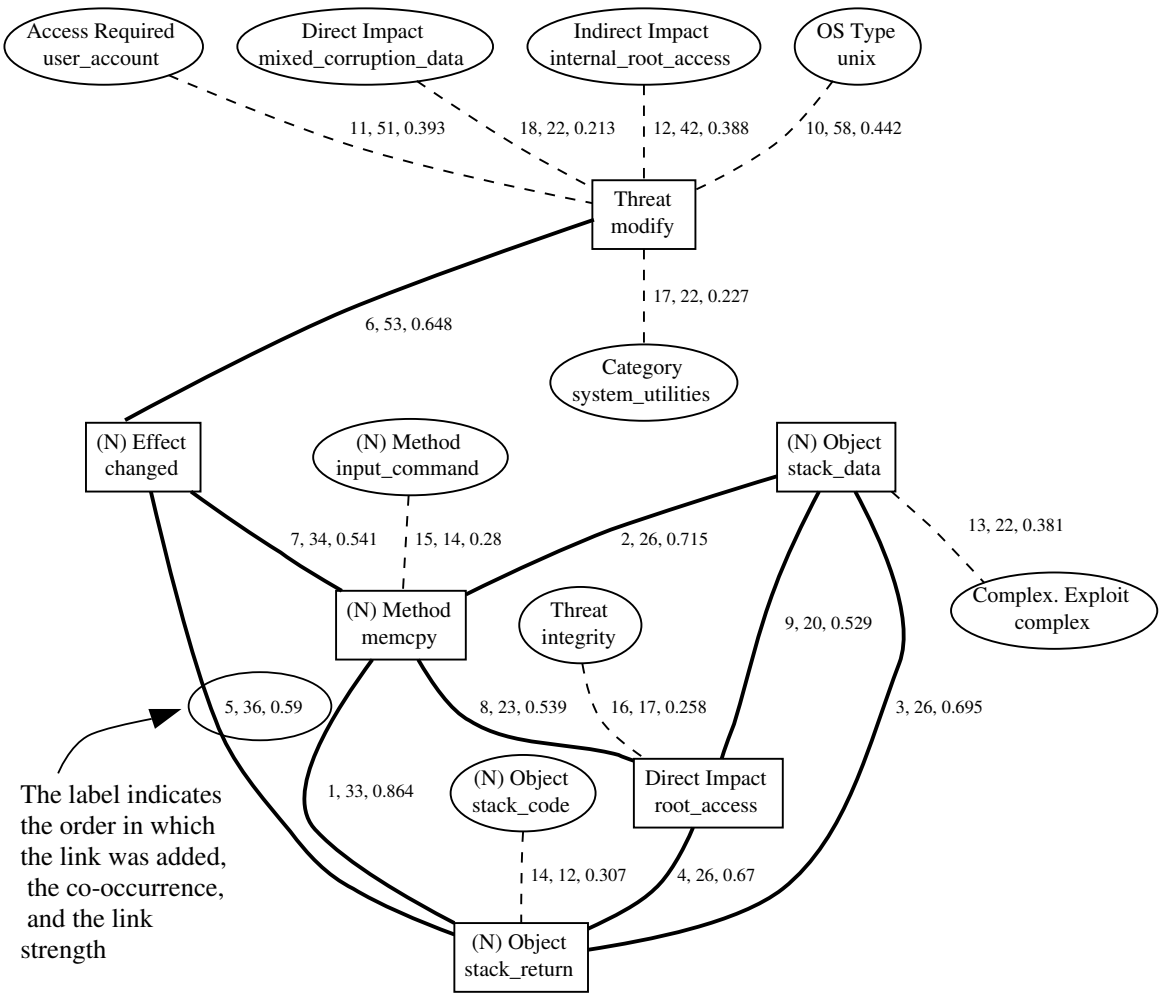


Figure 5.12: Principal network number 1 for co-word analysis.

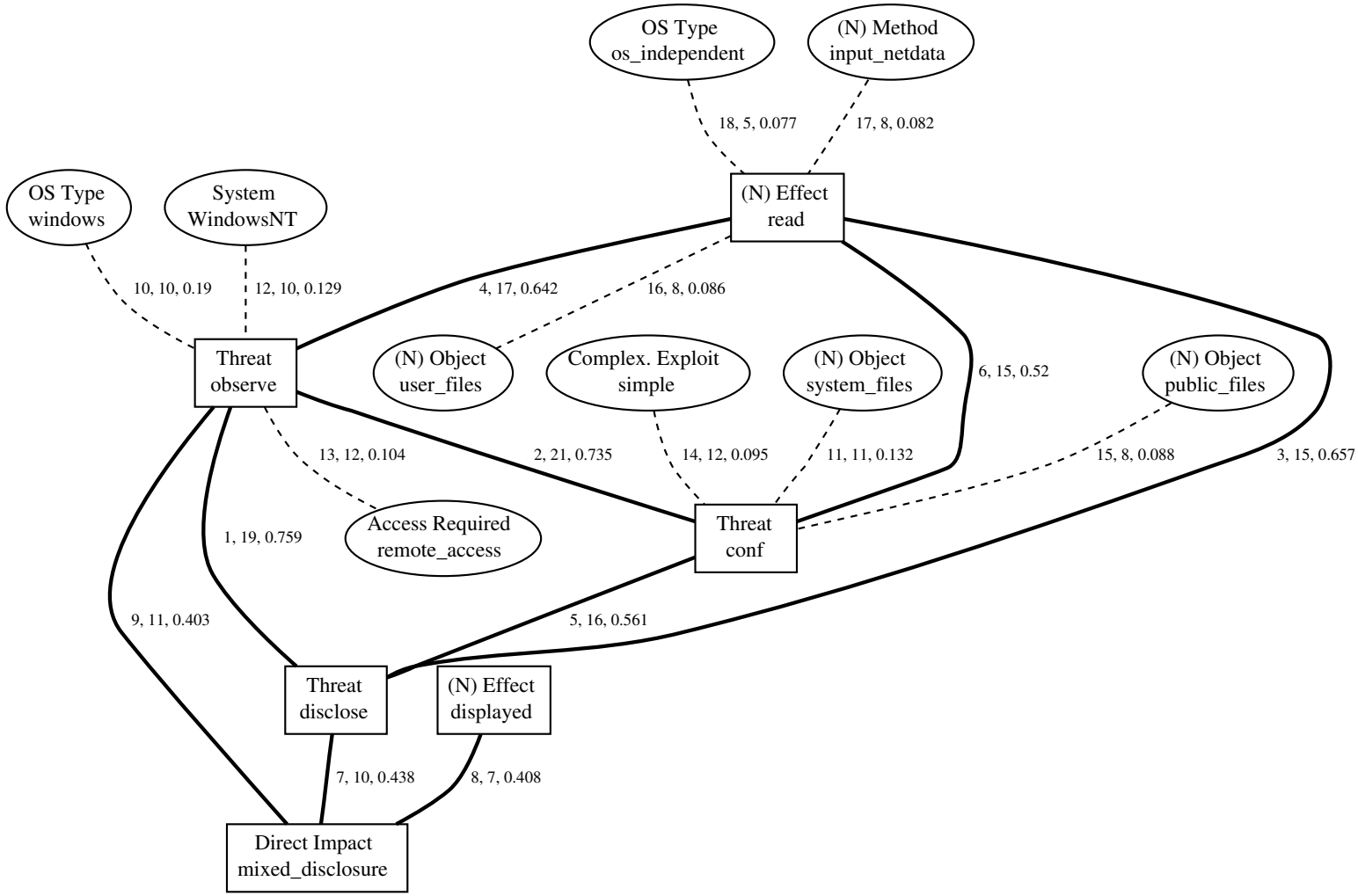


Figure 5.13: Isolated network number 2 for co-word analysis.

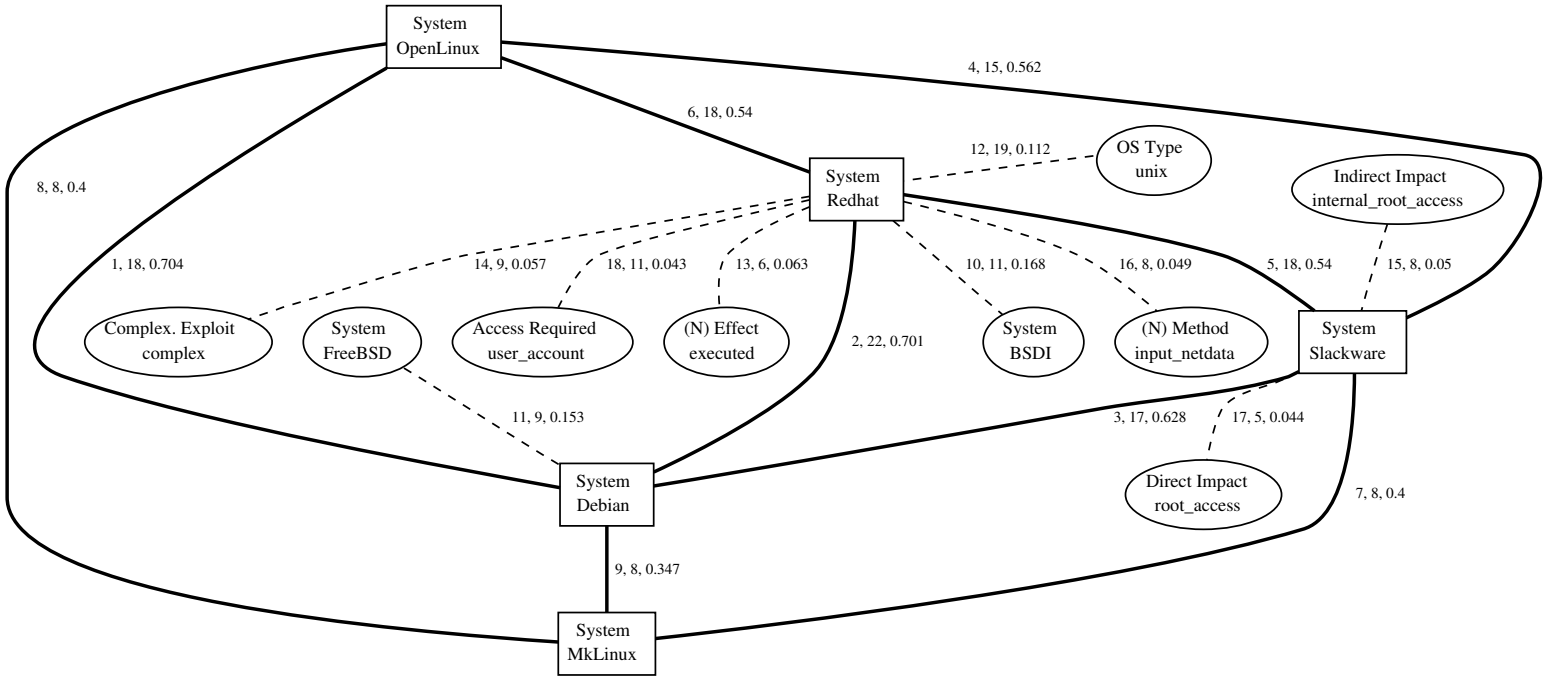


Figure 5.14: Isolated network number 3 for co-word analysis.

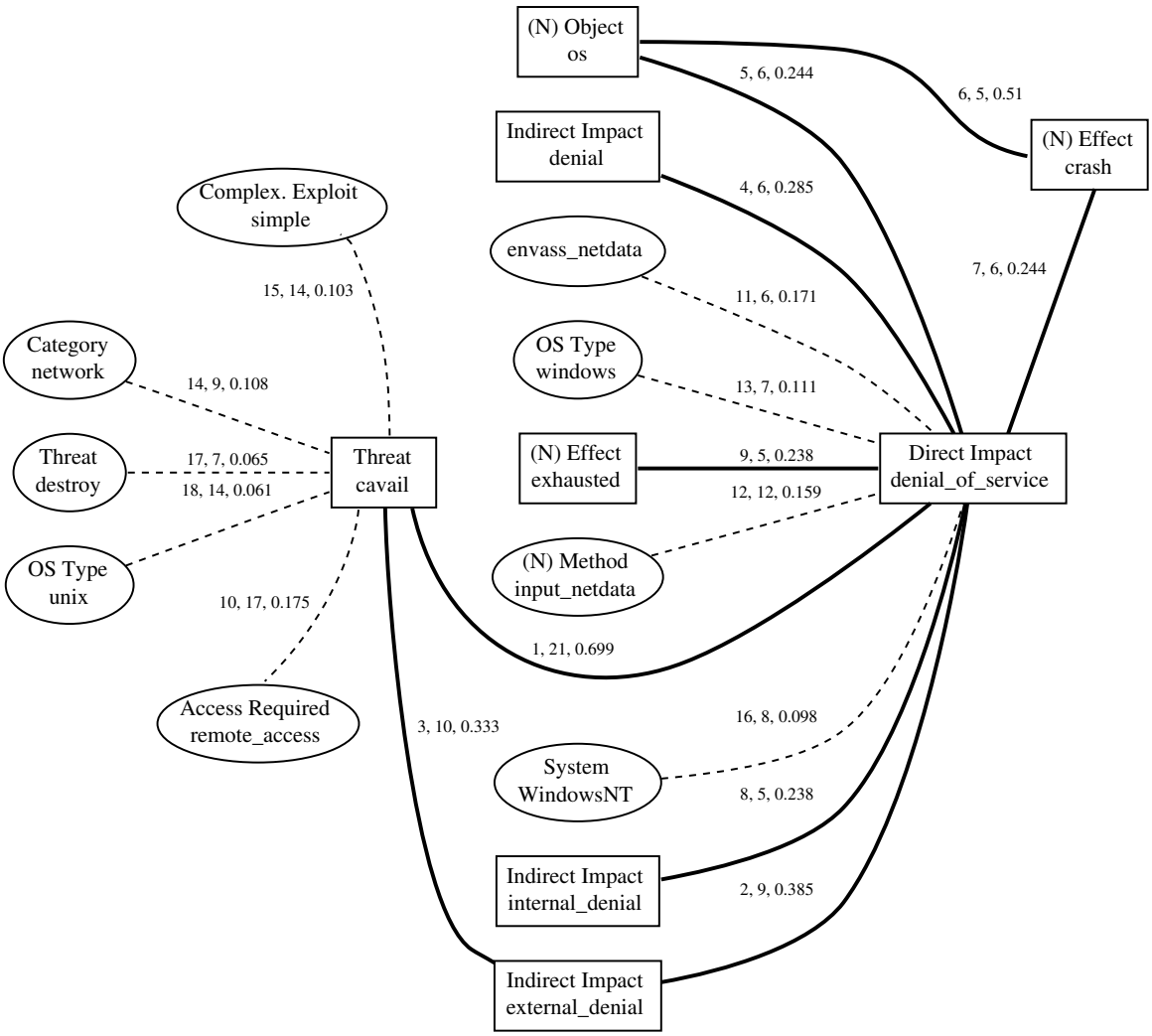


Figure 5.15: Isolated network number 4 for co-word analysis.

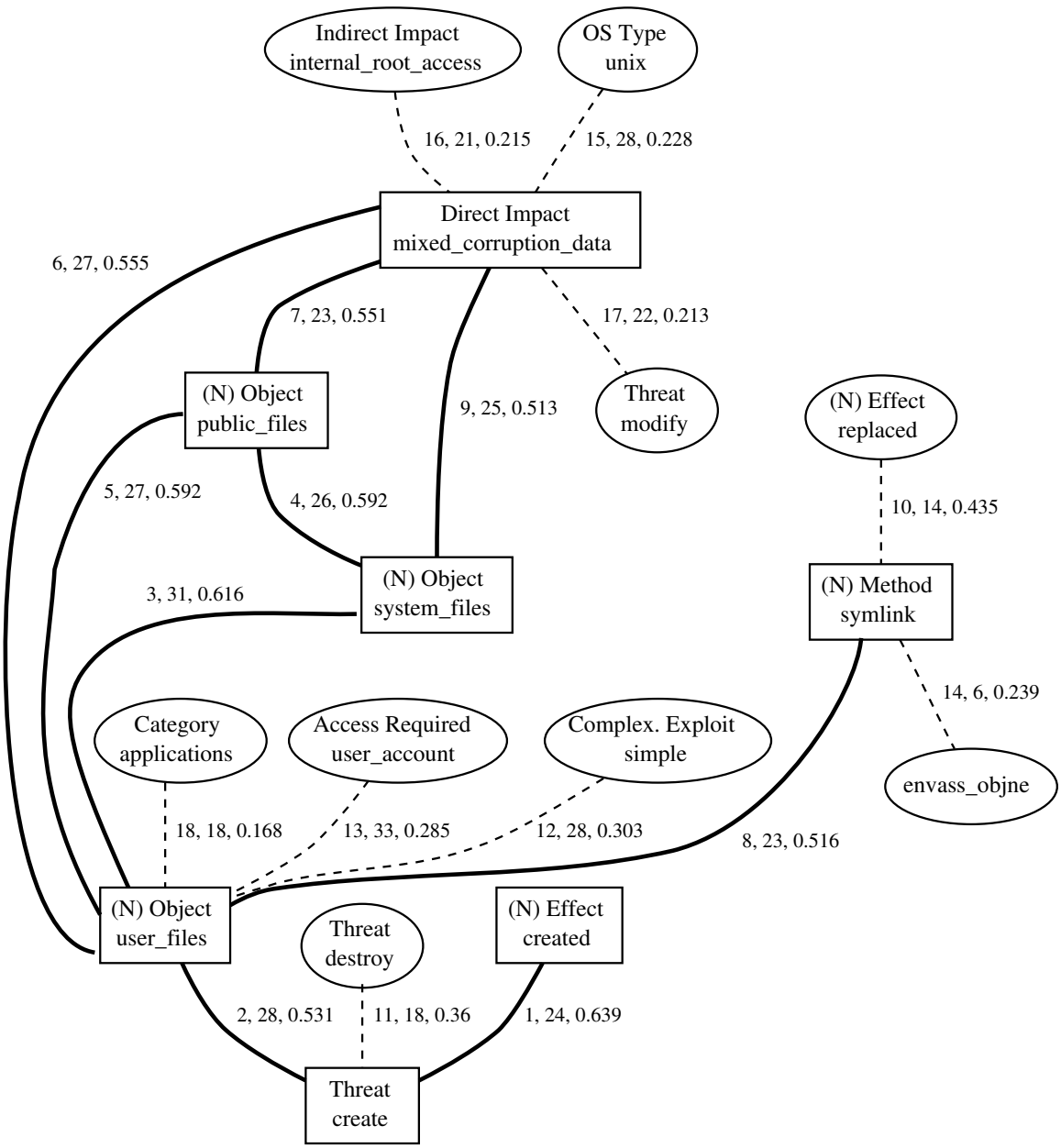


Figure 5.16: Principal network number 5 for co-word analysis.

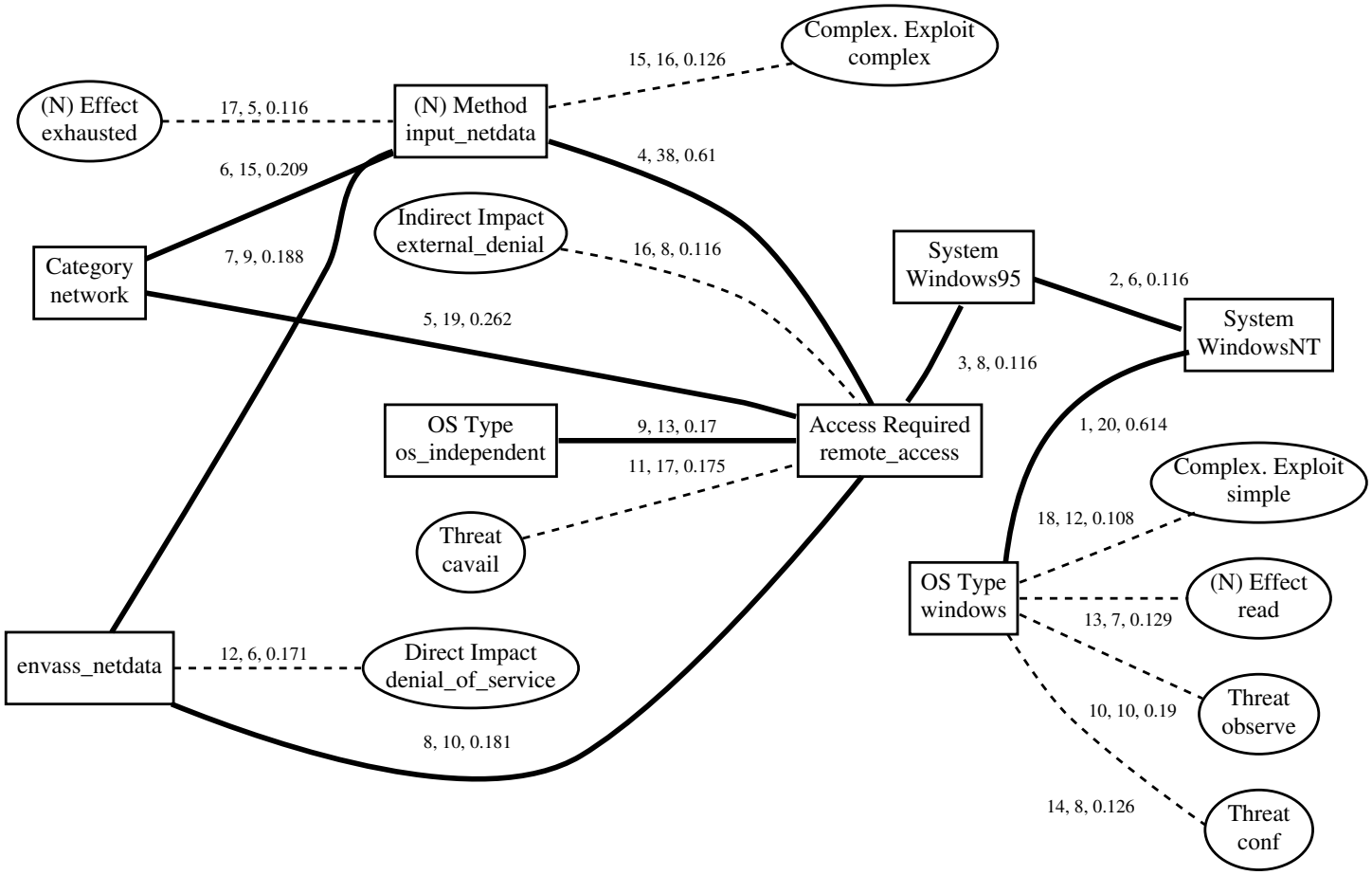


Figure 5.17: Isolated network number 6 for co-word analysis.

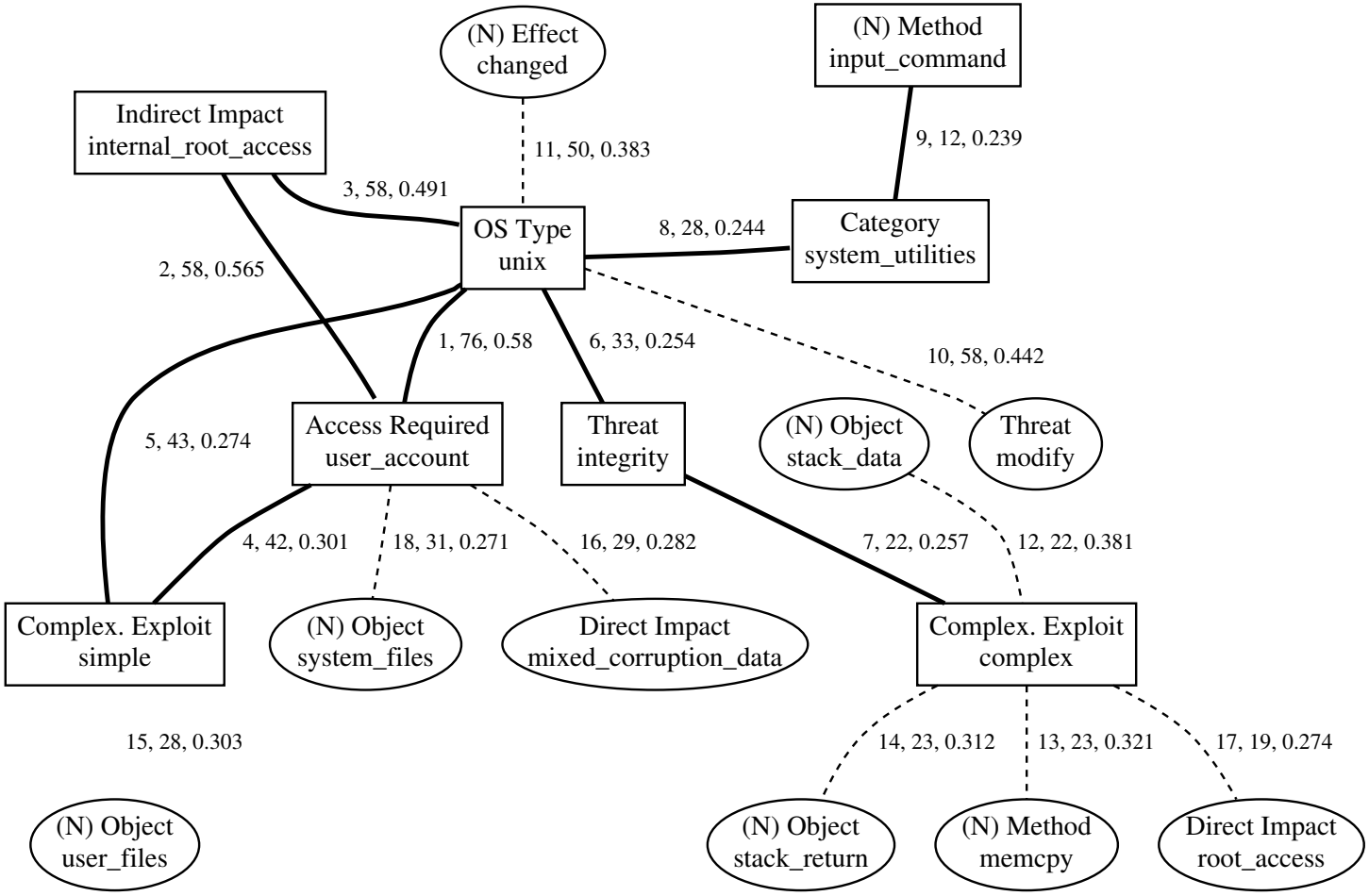


Figure 5.18: Network number 7 for co-word analysis.

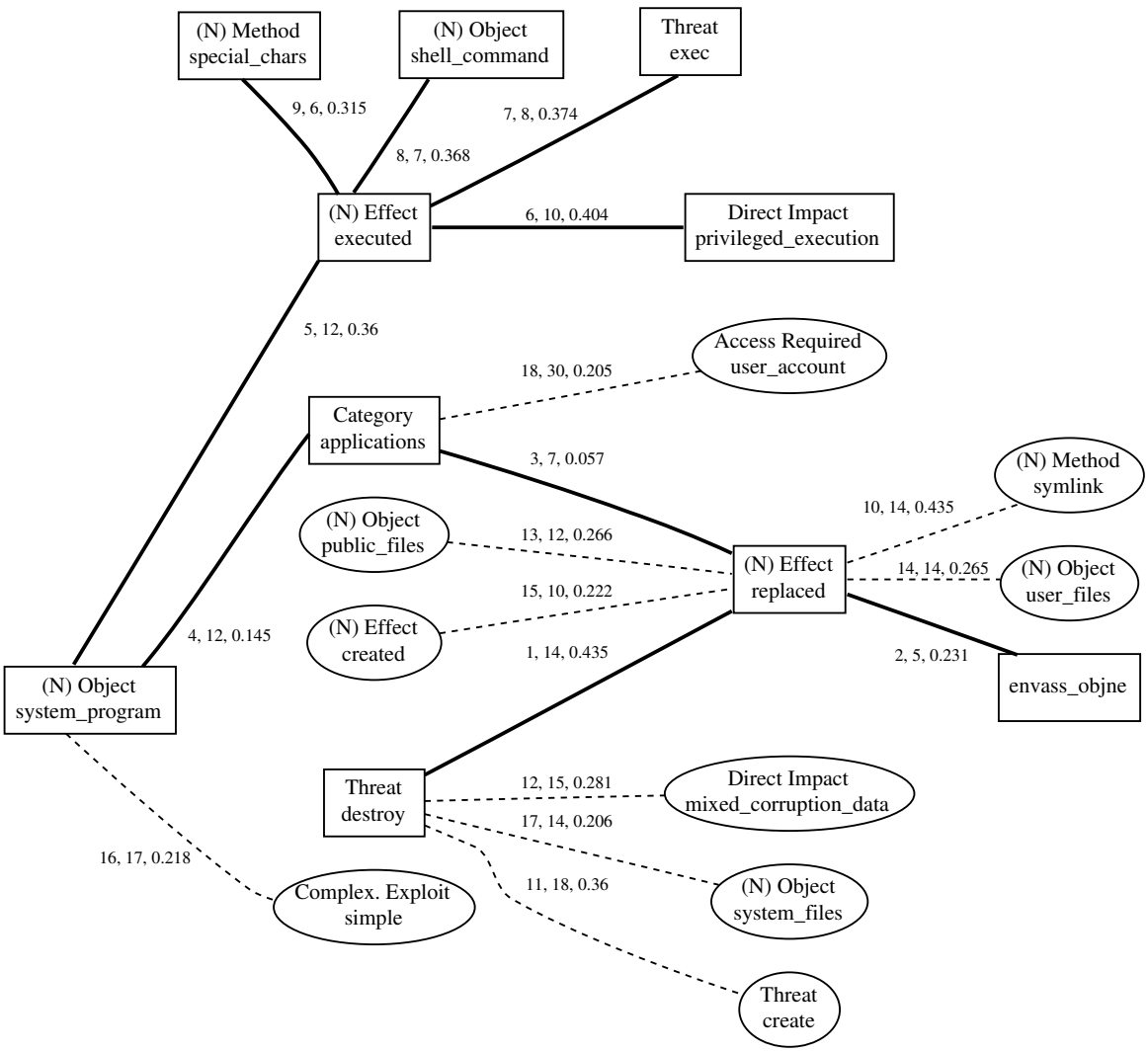


Figure 5.19: Network number 8 for co-word analysis.

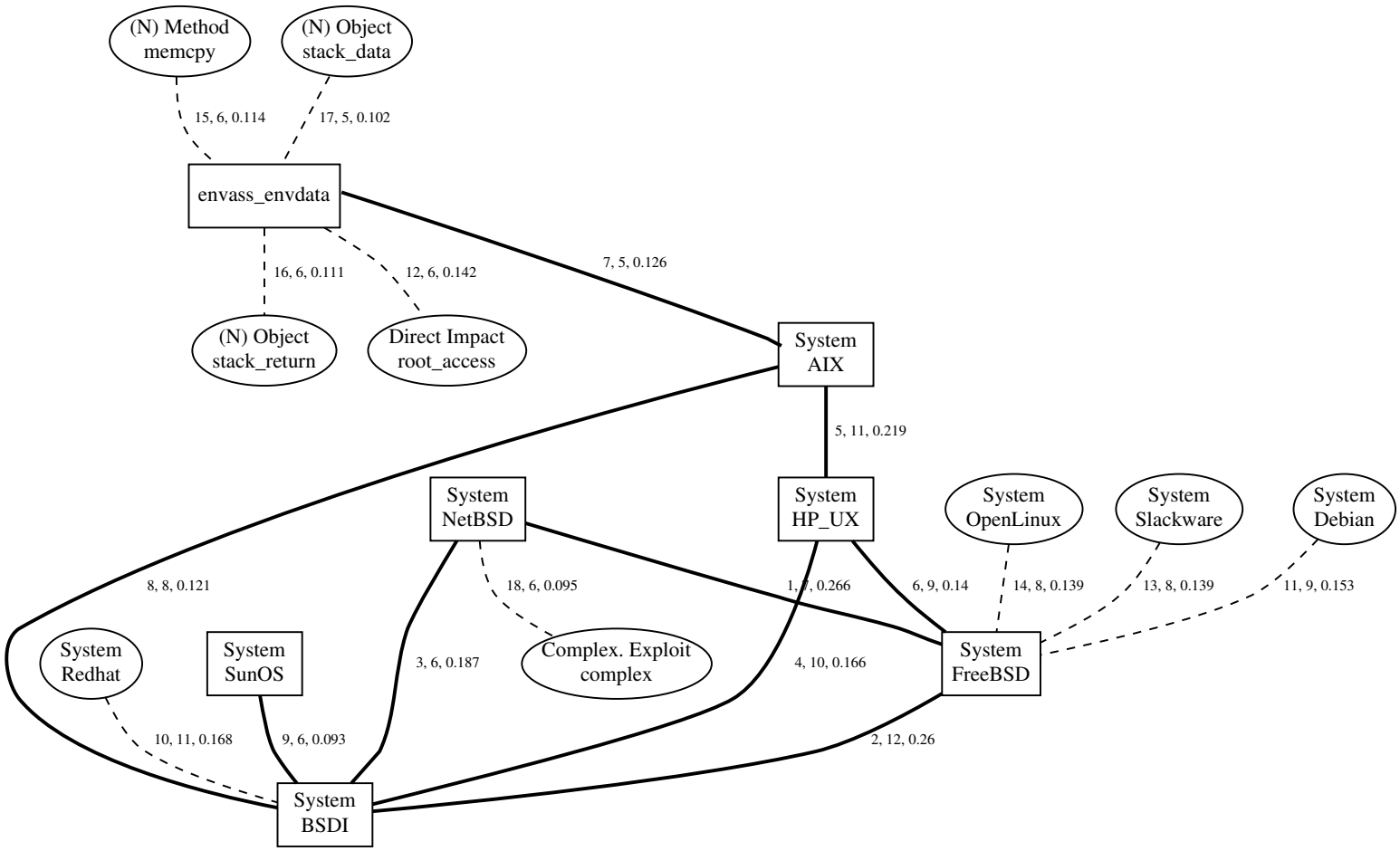


Figure 5.20: Isolated network number 9 for co-word analysis.

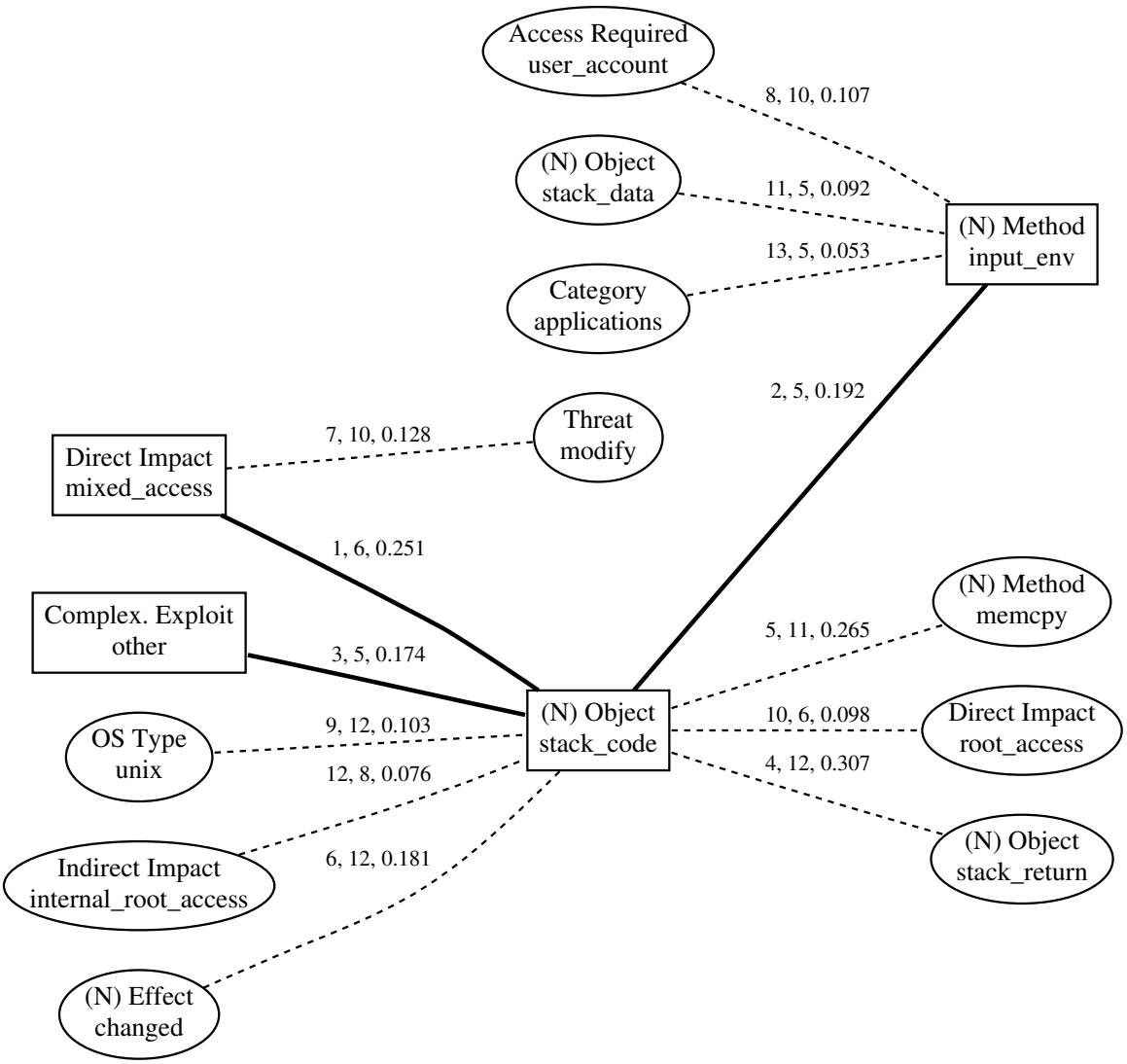


Figure 5.21: Isolated network number 10 for co-word analysis.

5.3.2 Induction of Decision Trees

The class of decision tree induction algorithms can be used to induce decision trees from a set of observations to classify new test cases [Quinlan 1986; Holsheimer and Siebes 1994; Breslow and Aha 1996]. As well as solving the classification problem, induction trees provide insights regarding the predictive structure of the data [Quinlan 1986].

Decision trees are used to classify an observation by traversing a tree along a path from its root to one of its leaves. Internal nodes of the tree are tests on the attributes that determine which path to take during the traversal. The labels associated with the leaves are the classes assigned to observations.

Decision trees are usually induced from the root downwards using a recursive algorithm that searches for the best partitioning of the known samples. A detailed description and analysis of the development of induction trees can be found in [Breslow and Aha 1996; Quinlan 1986; 1993].

Decision trees can be used with the taxonomic characters developed in Chapter 4 and the classifications mentioned in Appendix C to reveal relationships in the samples collected for the vulnerability database described in Section 5.2.

Example 5.2: We applied the top down tree induction algorithm as implemented in MLC++ [Kohavi et al. 1997] with a C4.5's confidence-based pruning of 0.7 [Quinlan 1993] to the entries of the database described in section 5.2. The confidence-based pruning value was chosen because it produced trees of interpretable size.

The fields used were: the direct impact classification described in Section C.2; the operating system classification described in Section C.6; the access required classification described in Section C.3; the vulnerability classification developed in Section 6.1; the environmental assumption features described in Section 4.2; and the features for the nature of vulnerabilities described in Section 4.3. These fields were chosen because they are either the taxonomic characters described in Chapter 4, or classifications using decision trees that satisfy the requirements specified in Section 3.1.

The classification algorithm was able to produce a variety of decision trees with classification error rates below 40%. In this section we present the decision tree generated for predicting the direct impact classification described in Section C.2 from the remaining fields.

This tree, shown in Figure 5.22, was selected from the possible trees because it illustrates some of the conclusions that can be derived from these trees even though the error rates for the classifications produced with these trees are as high as 40%.

Error rates and the number of samples used to generate the tree are shown by the shading and the line thickness of the nodes in the tree as shown by the scales in the lower right corner of the figure. Because this decision tree is not being used to classify new observations, but rather to derive information regarding the predictive structure of the data, we can ignore the paths to leaves that have high error rates, and deduce information from those paths that have low error rates.

Note that the learning algorithm found that the "Object—Stack return" feature is a good predictor for the vulnerabilities that result in the modification of running code (the `root_access` value). The direct impact classification does not use this feature to determine the value of the classification. Hence, the tree reveals a strong connection between vulnerabilities whose exploitation affect the stack of the program (from the tree), and those that result in the dynamic modification of the executable code of the program (from the classification). □

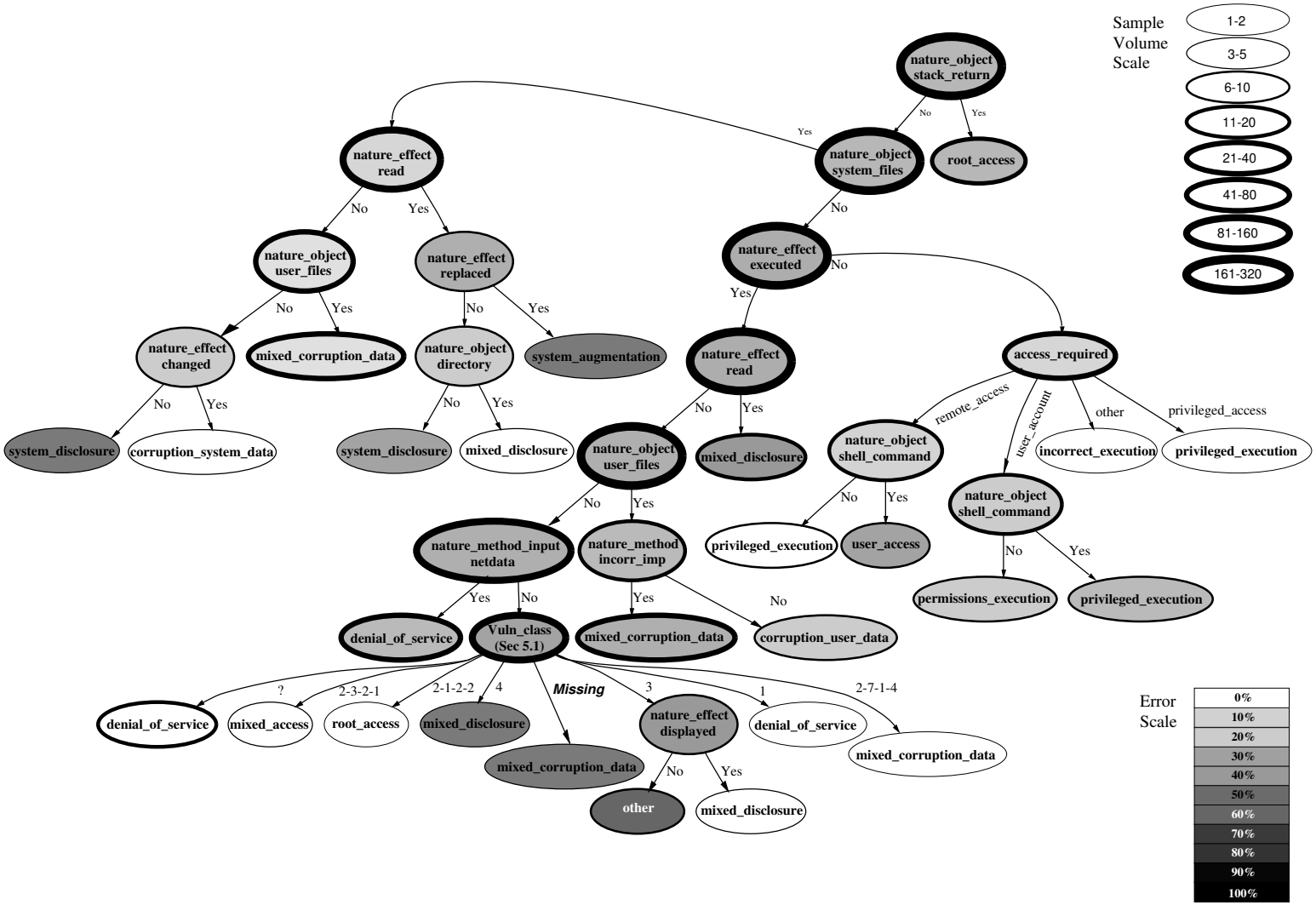


Figure 5.22: A decision tree generated by MLC++ for predicting the direct impact of a vulnerability.

5.3.3 Data Visualization Tools

Multivariate data visualization tools, such as XGgobi [Swayne et al. 1998] or the visualization utilities of Mineset (see `http://www.sgi.com/Products/software/MineSet`), can be effective in the identification of patterns or regularities in vulnerabilities when used with the taxonomic characters listed in Chapter 4 and the classifications listed in Appendix B.

As shown in the following example, multivariate visualization tools can be used to discover or infer information that may not be readily apparent from the data.

Example 5.3: XGobi implements a visualization technique called *linked brushing*. The description of this technique is given in [Swayne et al. 1998] as:

> In brush mode, a user can manipulate a rectangular "paintbrush" to change the glyphs and/or colors of points under the brush... The real power of brushing stems from linking multiple XGobi windows: As a user brushes points and/or lines in one XGobi window, the changes are automatically reflected in all other "linked" XGobi windows.

This technique was applied to the data in the vulnerability database described in Section 5.2. The graph to the left of Figures 5.23 and 5.24 shows the feature that is being "brushed." The graphs on the right of the figures show, automatically, the vulnerabilities for which this feature is set. The graphs to the right show the distribution of the vulnerabilities for the classifiers Direct Impact (see Section C.2), OS Type (see Section C.6), and the vulnerabilities classified with the classification system presented in section 6.1.

In Figure 5.23, the brushed feature is the taxonomic character "Input Type, `netdata`" described in section 4.3.4. The value of this feature is "Yes" when the source of the input that is necessary to affect the the object is network data. When this feature is brushed, the vulnerabilities—in the linked graphs—for which the brushed feature is set are represented with the *plus* glyph (+).

The graphs on the right show that vulnerabilities whose exploitation require network data are not specific to a single direct impact, are not specific to a single operating system, and are predominantly the result of design errors or coding faults. This result is in accordance with the results in [Miller et al. 1990; Miller et al. 1995], where random inputs were not sufficient to crash UNIX network daemons.

In Figure 5.24, the brushed feature is the taxonomic character "Object, `stack_return`" described in section 4.3.1. The value of this feature is "Yes" when the object affected by the exploitation of a vulnerability is the return address, stored in the stack, of a function. When this feature is brushed, the vulnerabilities in the linked graphs for which the brushed feature is set are represented with the *plus* glyph (+).

The graphs on the right show that vulnerabilities whose exploitation affect the return address of a function are mostly limited to a single operating system (UNIX), and mostly have two impacts: root access and mixed access. However, there is no single environmental assumption that is responsible for most of these vulnerabilities.

□

5.4 Chapter Summary

In this chapter we show that given a database of taxonomic characters for a representative population of software vulnerabilities, we can apply statistical and analysis tools to extract patterns, distributions, and regularities that may not be readily obvious. These result in added insights into the nature of vulnerabilities.

As shown in the following examples, in this chapter we applied a statistical analysis technique (co-word analysis), a machine learning technique (induction trees), and a data visualization tool (Xgobi) to demonstrate the experimental hypothesis described in Section 5.1.

Example 5.4: Among the results for application of the co-word analysis technique we can highlight that it reconstructed, from taxonomic characters, groups of vulnerabilities that are intuitive to seasoned computer security professionals. It also shows that vulnerabilities apply to more than one variant of UNIX even if these do not share the same code-base. A possible explanation is that system utilities are commonly designed to be cross-compiled in multiple variants of UNIX. □

Example 5.5: The Xgobi multivariate data visualization tool was effective in the realization that vulnerabilities in network utilities (predominantly in UNIX) are often the result of design and coding faults, rather than mistaken assumptions about the environment in which programs execute. □

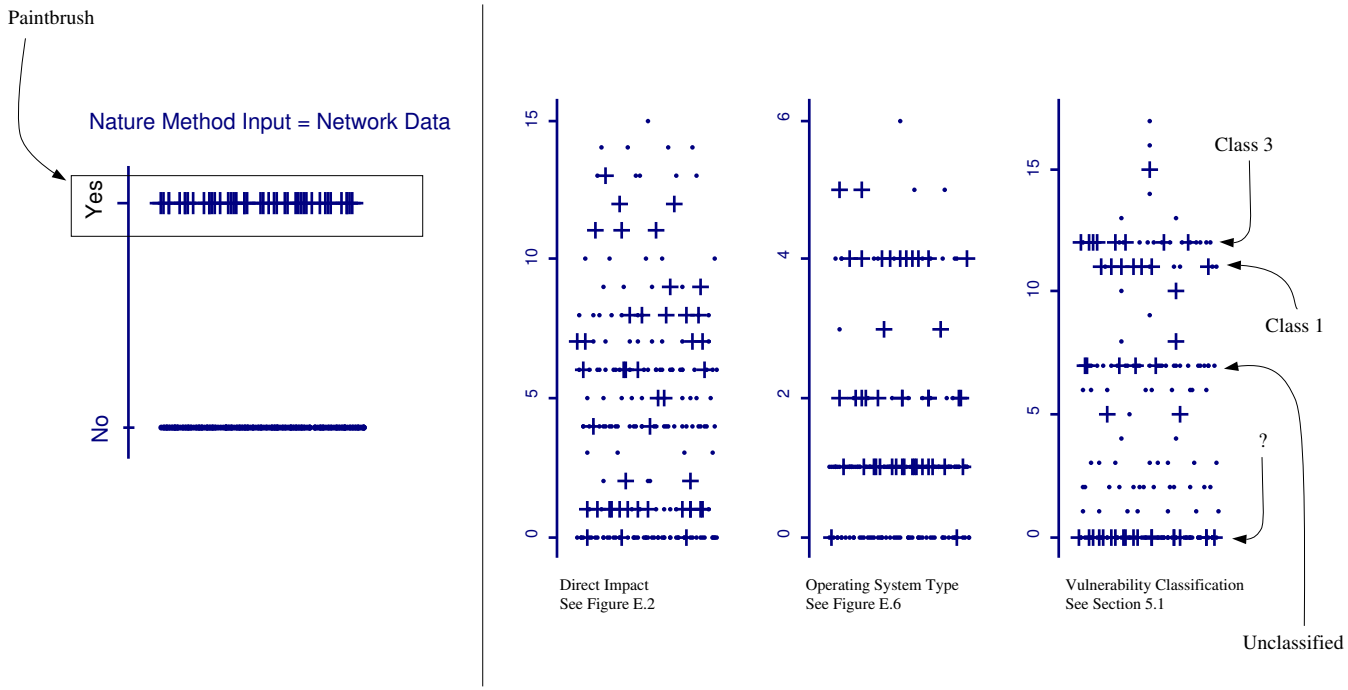


Figure 5.23: The application of visualization techniques to derive knowledge from the vulnerability database. Bush linking Method Input = Network Data.

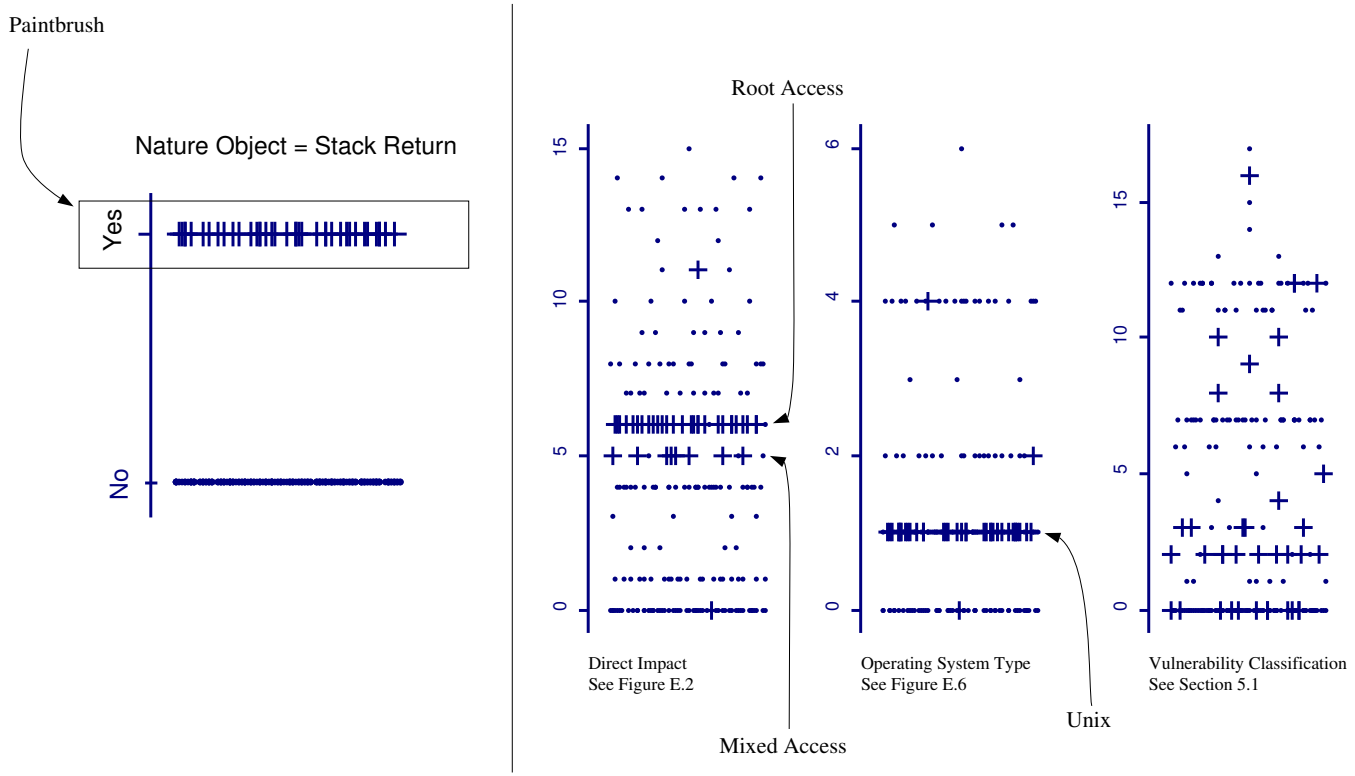


Figure 5.24: The application of visualization techniques to derive knowledge from the vulnerability database. Bush linking Method Object = Stack Return

6 *A PRIORI* CLASSIFICATIONS OF SOFTWARE VULNERABILITIES

As mentioned in Section 3.1.4, classifications can be made *a priori* (i.e. non-empirically from an abstract model) or *a posteriori* (empirically by looking at the data). In this section we present examples of *a priori* classifications.

6.1 A Taxonomy for Software Vulnerabilities

> *Complexity* is the label we will give to the existence of many interdependent variables in a given system. The more variables and the greater their interdependence, the greater the system complexity. Great complexity places high demands on a planner's capacities to gather information, integrate findings, and design effective actions [Dorner 1996].

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; DeMillo et al. 1987; Beizer 1983; Kolawa and Hicken 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 in no algorithm that takes as an input an instance of the environment to determine if the assumption holds or not [Hopcroft and Ullman 1979]), or because the assumption is subjective.

Example 6.1: The following assumptions cannot be decided as described in the preceding paragraph:

- The input string read from a file describes a Turing Machine that will halt [Hopcroft and Ullman 1979].
- The contents of a file are evil (or good).
- The machine will never run out of memory (decided at the beginning of a complex program).

□

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 [Hopcroft and Ullman 1979].

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.

Example 6.2: An informal poll was taken of approximately 50 commercial developers and graduating seniors in the United States and in Bolivia[1]. The poll asked the developers if in their programs they perform the checks needed to verify that their environmental assumptions hold.

Approximately 10% claimed to know and perform the checks necessary for the development of secure programs, 40% admitted that they do not, and 50% did not even know what these checks were, and why they should perform them. □

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 6.1, there exists a decision

[1]The poll included developers from Xerox PARC, Sun Microsystems, Hewlett Packard, Microsoft, Ars Logica, Anderson Consulting, Oracle, IBM, Lotus, and graduating seniors and graduate students of Purdue University and The Catholic University in Bolivia. 12 of these developers and students were in Bolivia. The poll was performed explicitly for this dissertation.

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 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 6.1, vulnerabilities that cannot be classified because of this reason are tagged as unknown.

The classes shown in Figure 6.1 correspond to groups of vulnerabilities as defined in Section 2.1.3. As shown in Figure 6.2, *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 ▲.

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

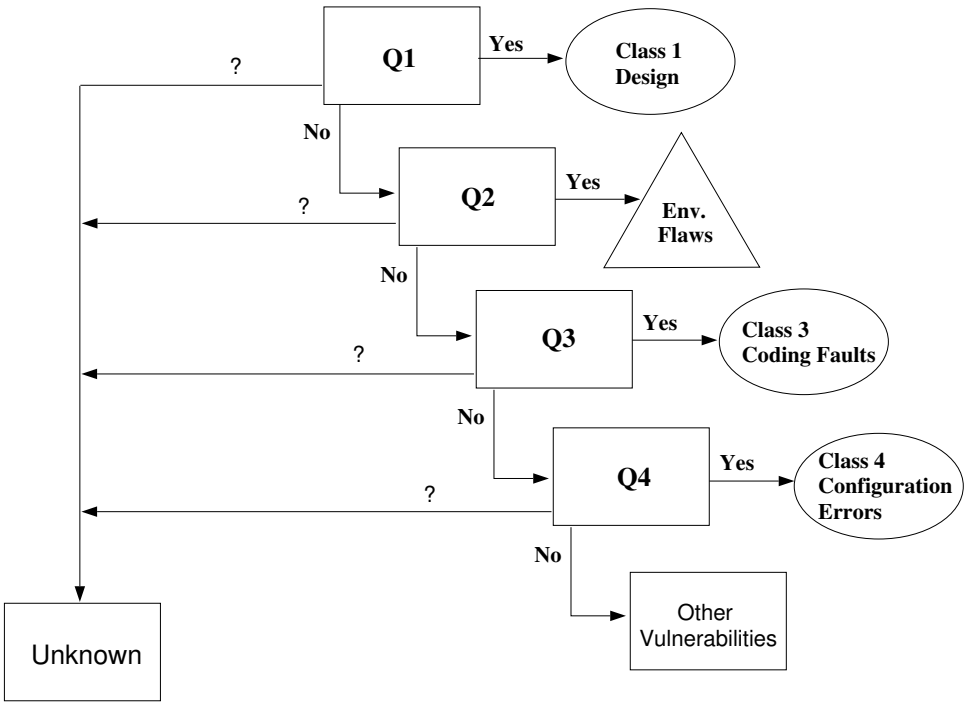


Figure 6.1: A classification for the identification of environmental assumptions made by programmers—Part 1.

possible. However, as we show in this dissertation, 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 $< o_1, o_2, \ldots, o_{n-1}, algorithm >$. In this section, the algorithm is referred to as an attribute constraint.

As shown in Figure 6.3, 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.

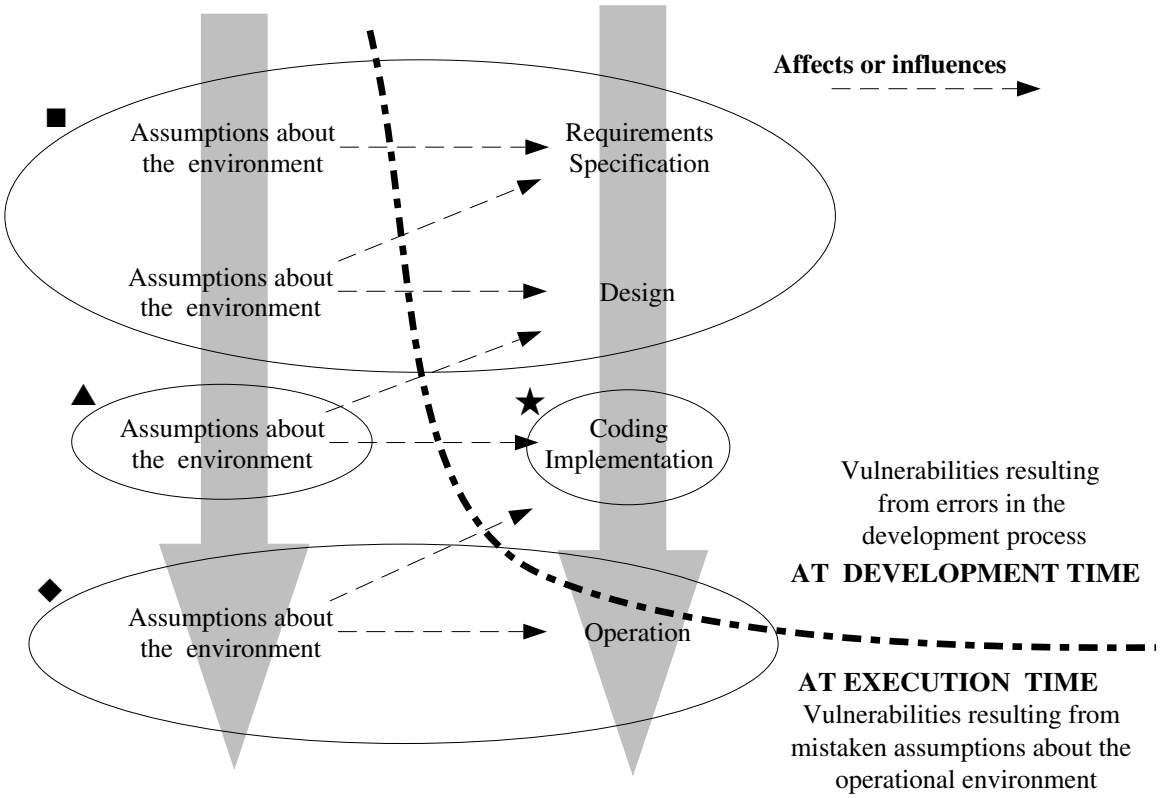


Figure 6.2: The classification developed in this section corresponds to the vulnerabilities in the area identified with the symbol ▲.

Recall from the definition of attribute of an object (see Section 2.1.5), that 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.

Example 6.3: The following class illustrates the use of sets of objects and sets of attributes for the specification of a single environmental assumption made by a programmer for a

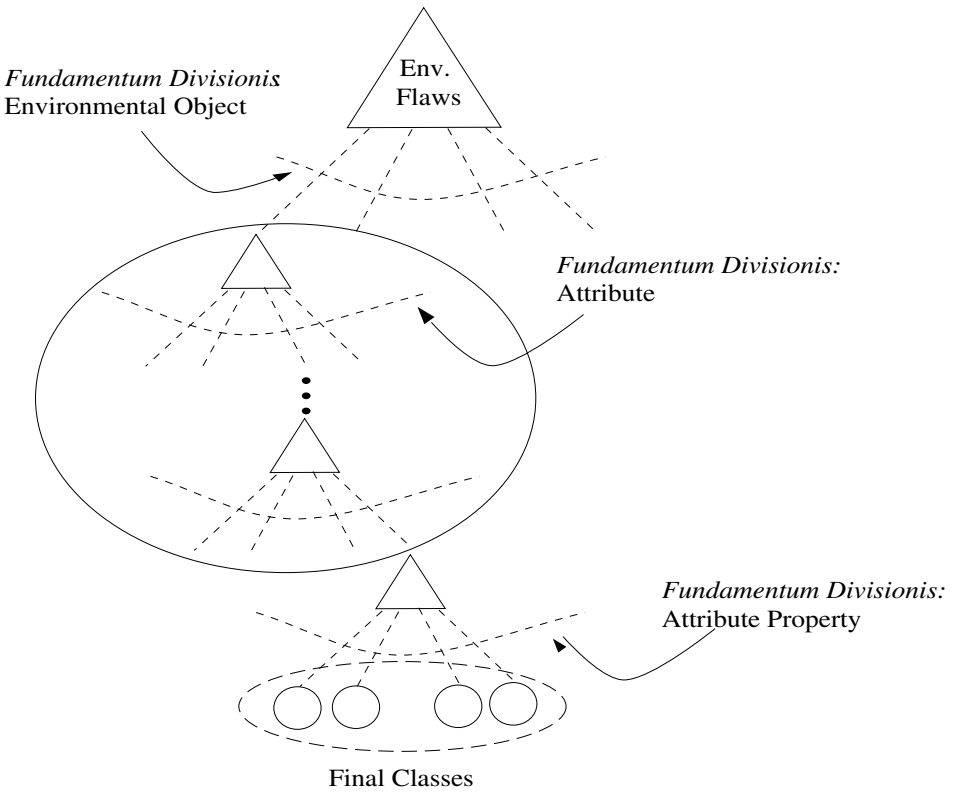


Figure 6.3: A classification for the identification of environmental assumptions made by programmers—Part 2.

system: (Running program, file — Program privilege, file name — program is SETUID or SETGID, and the final object for the name is `/tmp/abc`)

The property "final object for the name" is not considered an attribute of the file name because a function must be applied to determine if the name is a final object (i.e. it does not refer to a late-binding link). □

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 6.5, through 6.12 show 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).

Without loss of generality, the instantiation presented in this section is specific to the vulnerabilities in the database described in Section 5.2, 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. These instances are labeled in the tree with a note number that corresponds to one of the following notes:

Note 2-1-3-1: (Running program — environment — is `system()` safe) The vulnerabilities in this class correspond to those in which the programmer assumed that the environment for the program has the default values. The IFS environment variable should be the default, the PATH environment variable should include only the default system directories, etc.

Note 2-4: (Network stream) A network stream is a TCP or UDP connection that provides a stream of bytes rather than a packet.

Note 2-7-1-4: (Running program — name — is the same object as x) The vulnerabilities in this class correspond to those in which the programmer assumed that two or more operations on a single file name would affect a single file object.

Note 2-7-1-5: (Running program — name — is final) The vulnerabilities in this class correspond to those in which the programmer assumes that a file name refers to an actual file object and not to a late binding reference or symbolic link.

Note 2-7-2-3: (Running program — content — is a known program) The vulnerabilities in this class correspond to those in which the programmer assumes that a given string corresponds to the name of a registered program. These are the programs in the system that are marked as executable (for example, in UNIX files that the `file` utility identifies as executable binaries or scripts), and that are owned by the system.

Note 2-7-2-4: (Running program — content — is a text log file) The vulnerabilities in this class correspond to those in which the programmer assumes that a file is a known log file. If the file exists then it cannot be a type (as indicated by the `file` system utility

in UNIX) other than text, and it must either have a header that identifies it as the valid log type or its content must match a regular structure (or regular expression).

Note 2-7-2-5: (Running program — content — is of a known type) The vulnerabilities in this class correspond to those in which the programmer assumes that a file is of a known type (shell script, text file, executable binary, etc.)

Note 2-9-1-3: (Program string — content — is free of shell meta-characters) The vulnerabilities that correspond to this class are for programs in which the programmer assumes that a string is free of shell escape meta-characters. These strings are typically passed to the shell as the name of a command. In UNIX systems these meta-characters are command separators, pipes, synchronous and asynchronous execution indicator (`; | & && ||`), and command substitution characters (` ‘ `). In Windows NT, these characters are command separators and conditional command execution indicators (`& && ||`).

Note 2-10: (Network IP packet) A network IP packet is a connection that allows the manipulation of network data packets without hiding the packet details.

Note 2-11-1-1: (Directory, running program — directory name, name of user who ran the program — is in valid user space for the user who invoked the program) The vulnerabilities that correspond to this class are for programs that typically run with privileges that exceed those of the user who invoked the program. These program can read and write to directories that the original user would not be allowed to access. In programs that have these vulnerabilities, the programmer assumes that the directory being read, created, or modified ultimately refers to (following all late binding references) a directory that the user who invoked the program would have access to.

Note 2-12-1-1: (File, running program — file permissions, name of user who ran the program — user who invoked the program can read the file) The vulnerabilities that correspond to this class typically run with privileges that exceed those of the user that invoked the program and hence can read files that the user would normally not

be allowed to access. In these vulnerabilities the programmer assumes that the user who invoked the program has access to a file being read.

Note 2-12-1-2: (File, running program — file permissions, name of user who ran the program — user who invoked the program can write to the file) The vulnerabilities that correspond to this class typically run with privileges that exceed those of the user who invoked the program and hence can write to files that the user would normally not be allowed to access. In these vulnerabilities the programmer assumes that the user who invoked the program has write access to a file.

Note 2-12-2-1: (File, running program — file name, program privileges, name of user who ran the program — is a valid temporary file) The vulnerabilities in this class correspond to those in which the programmer assumes that the ultimate object a file name refers to (following all late binding references) is a temporary file and does not exist.

Note 2-12-2-2: (File, running program — file name, program privileges, name of user who ran the program — is in valid space for the user who invoked the program) The vulnerabilities that correspond to this class typically run with privileges that exceed those of the user who invoked the program and hence can read and write files that the user would normally not be allowed to access. In these vulnerabilities the programmer assumes that object that the file name ultimately refers to (following all late binding references) a file that the user would normally be allowed to access or modify.

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).

Example 6.4: In some UNIX systems derived from BSD UNIX, there is a vulnerability that allows unprivileged users to obtain sufficient information from the operating system to generate arbitrary NFS file handles [Krsul et al. 1998] (record `bsd_filehandles`). The `stat()` system call, and related functions, return a four byte field called `st_gen` that is different for each item in the file system, and which is used in the obfuscation of NFS file

handles, making these difficult to guess. Because all information used to generate a file handle is available to users, a user can generate file handles identical to those given to NFS client hosts by accessing file systems on the local server.

The incorrect code in the `vn_stat()` function, called by `stat()` reads:

```
    ...
    sb->st_gen = vap->va_gen;
    sb->st_blocks = vap->va_bytes / S_BLKSIZE;
    return (0);
}
```

A correct implementation will permit only root users to access this number, as shown in the following example source code:

```
    ...
    sb->st_flags = vap->va_flags;
    if (suser(p->p_ucred, &p->p_acflag))
       sb->st_gen = 0;
    else
       sb->st_gen = vap->va_gen;
    sb->st_blocks = vap->va_bytes / S_BLKSIZE;
    return (0);
}
```

This change will cause the `st_gen` field of the `stat` structure returned to unprivileged users to be zero, thus preventing ordinary users from determining file handles simply from the information returned by `stat()`.

If the designer of the system specified that the user should receive a copy of the `st_gen` field, then we would answer yes to question **Q1**. Otherwise the vulnerability must be a coding error because there is no assumption that the programmer can make that removes the vulnerability.

If the designers made it explicit that the value of this field is used to obfuscate the generation of NFS file handles, and clearly meant to keep this value away from the users, then the vulnerability is a code fault. However, this conclusion can be made with information regarding the design specification of the system, and not only with the faulty code. □

The taxonomy described in this section was applied to the vulnerabilities in the vulnerability database described in Section 5.2. 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 6.4 shows the distribution of vulnerabilities classified with the taxonomy presented in this section.

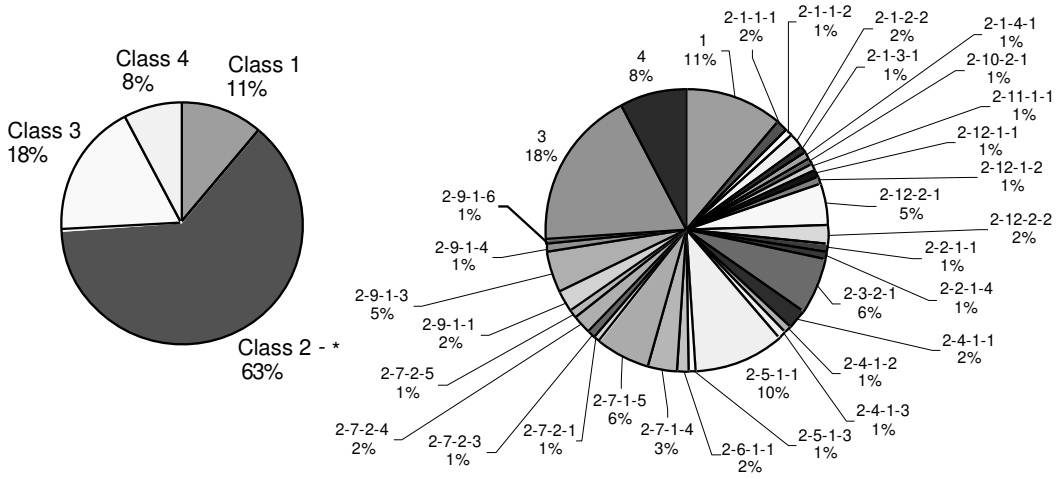


Figure 6.4: Distribution of vulnerabilities classified with the taxonomy presented in this section.

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 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 described in section 5.2. Some classes of vulnerabilities were predicted from these samples by extrapolating from existing classes, and the remaining vulnerabilities were classified. As shown in the following examples, 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.

Example 6.5: The class 2-4-1-2 (Network stream — Content Length — is at least x) was extended from the class 2-4-1-1 (Network stream — Content Length — is at most x). The database had vulnerabilities where a program assumes that the network stream will never have more than a certain number of bytes, and we predicted that a vulnerability exists

where the programmer assumes that the network stream has a least a certain number of bytes. During the classification of the remaining vulnerabilities, we found a vulnerability (`bind` Denial of Service) where the `bind` program receives less information that it expects from the network and goes into a tight endless loop with all interrupts disabled waiting for the rest of the data. □

Example 6.6: Class 2-5-1-3 (Command Line Parameters — Content — is 7 bit ASCII) was generated *a priori* and predicted that some programmer assumes that the input string is 7 bit ASCII and will use its characters as an offset into an array.

During the classification of the remaining vulnerabilities we encountered a vulnerability (`bash` command line vulnerability) where the program stores the input character in an integer variable and uses this variable for processing the command lines of the program. The program uses the value -1 in this variable to indicate the end of a command. When a user gives this program a string with the character code 255 decimal (377 octal) it will serve as an unintended command separator because the character value will be sign-extended when it is assigned to an integer variable. □

Similarly, we found vulnerabilities for the predicted classes 2-11-1-1 (Directory, Running program — Directory name, running program privileges, name of user that ran the program — is in valid user space for the user who invoked the program), 2-1-2-2 (Running Program — Name — length of name is at most x), 2-10-2-1 (Network IP Packets — Data Segment — length is at least x), and 2-12-1-2 (File, Running program — File permissions, user that ran the program — use that invoked the file can write to the file)

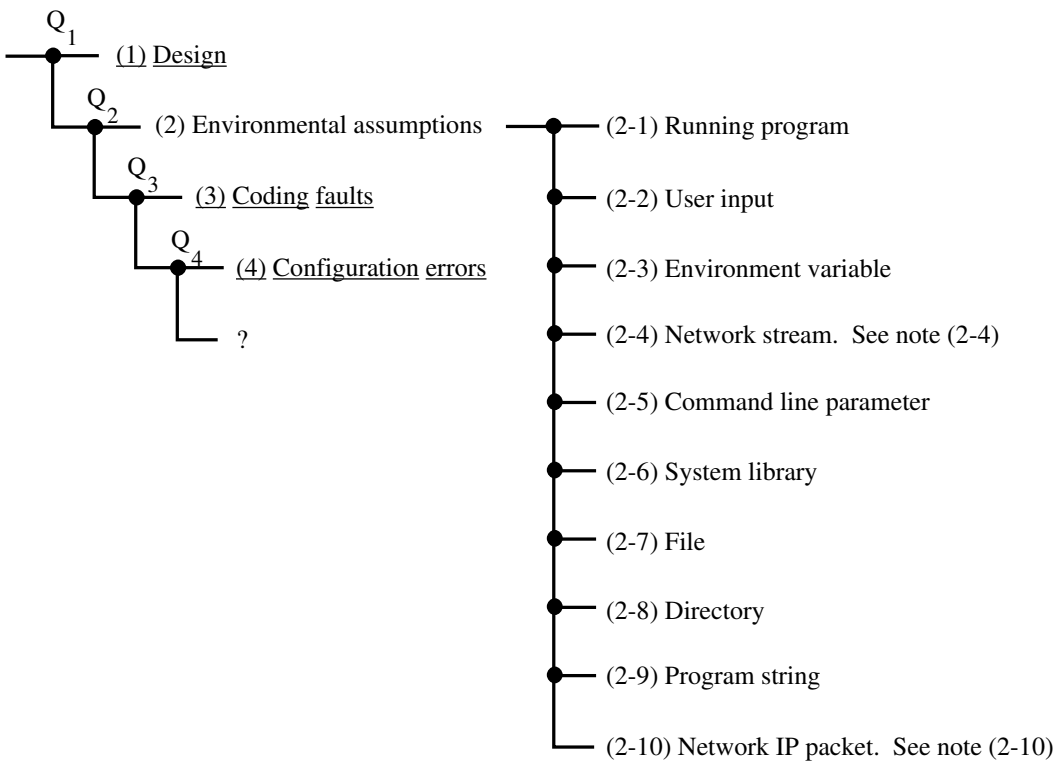


Figure 6.5: Taxonomy of Software Vulnerabilities Top Level

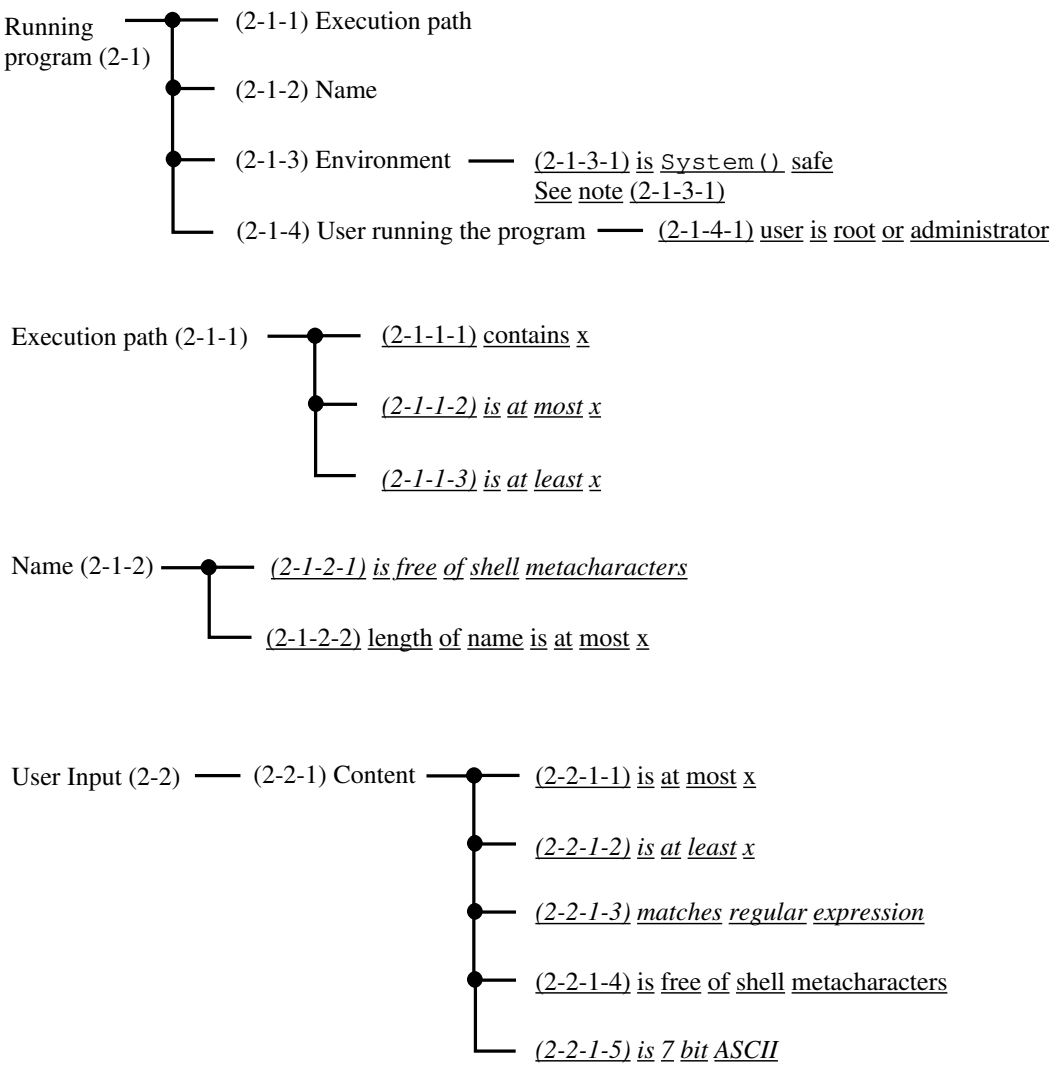


Figure 6.6: Taxonomy of Software Vulnerabilities, Levels 2-1 and 2-2

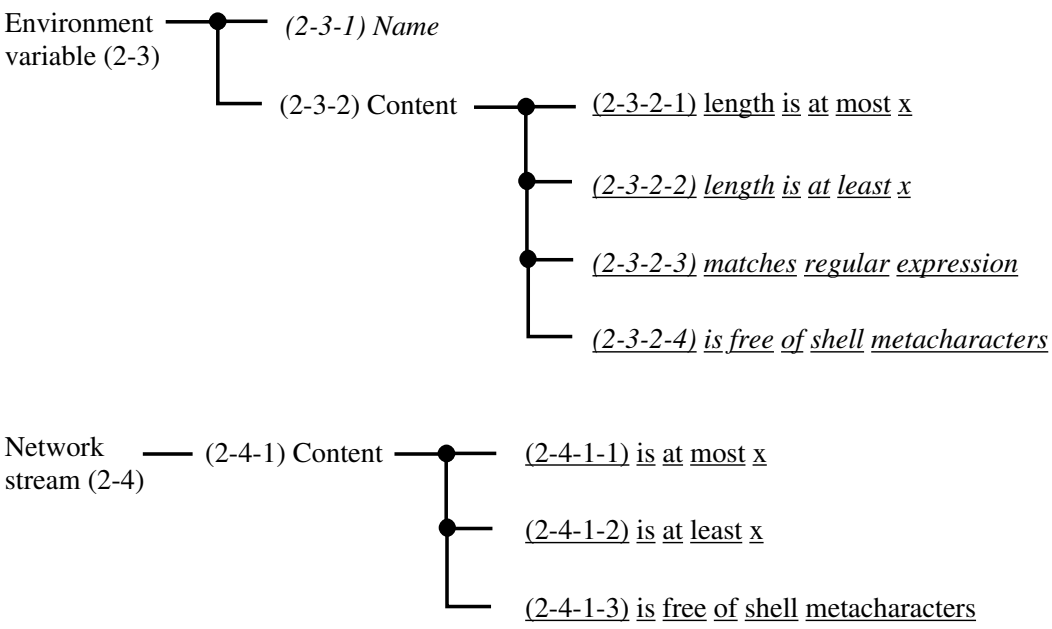


Figure 6.7: Taxonomy of Software Vulnerabilities, Levels 2-3 and 2-4

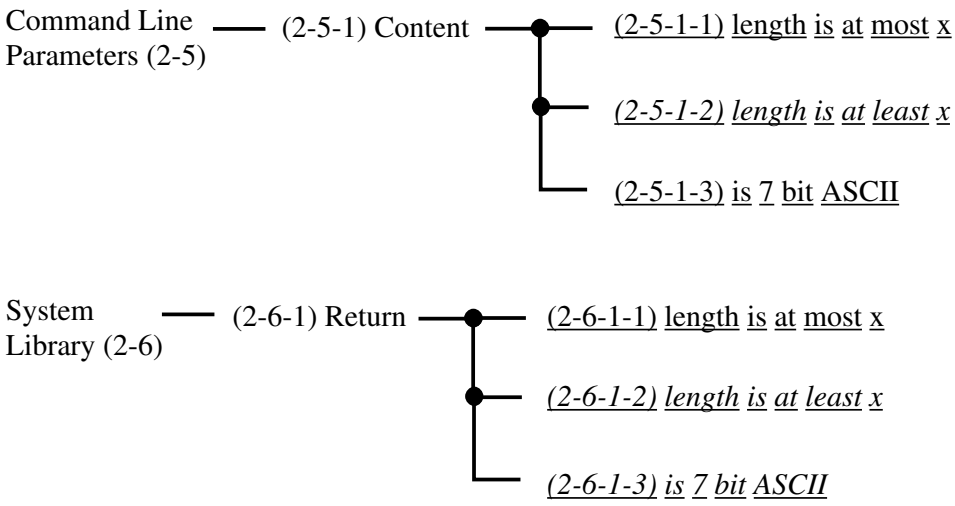


Figure 6.8: Taxonomy of Software Vulnerabilities, Levels 2-5 and 2-6

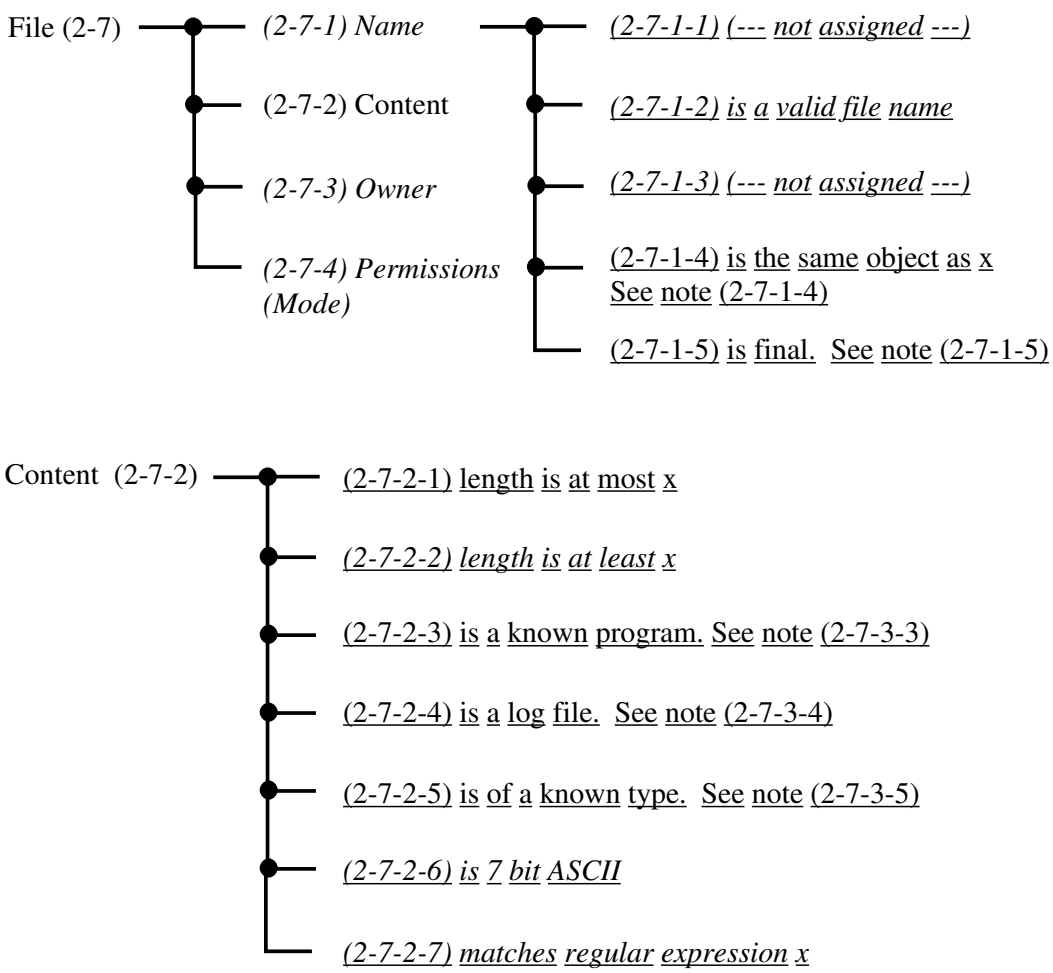


Figure 6.9: Taxonomy of Software Vulnerabilities, Level 2-7

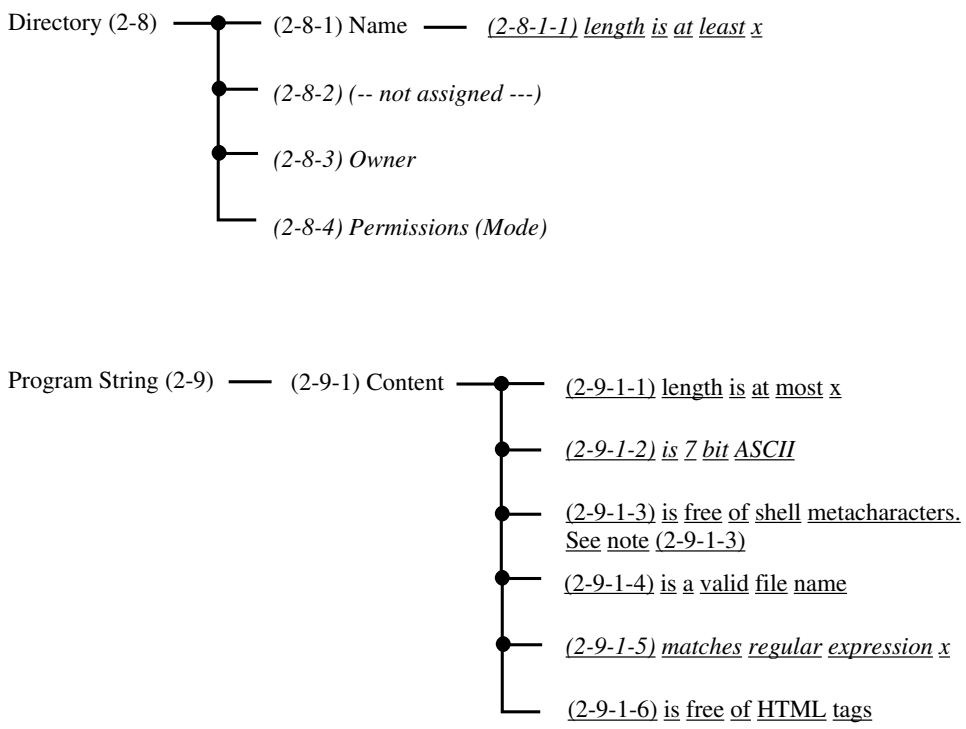


Figure 6.10: Taxonomy of Software Vulnerabilities, Levels 2-8 and 2-9

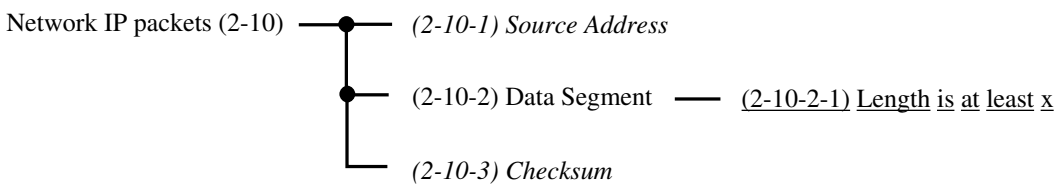


Figure 6.11: Taxonomy of Software Vulnerabilities, Level 2-10

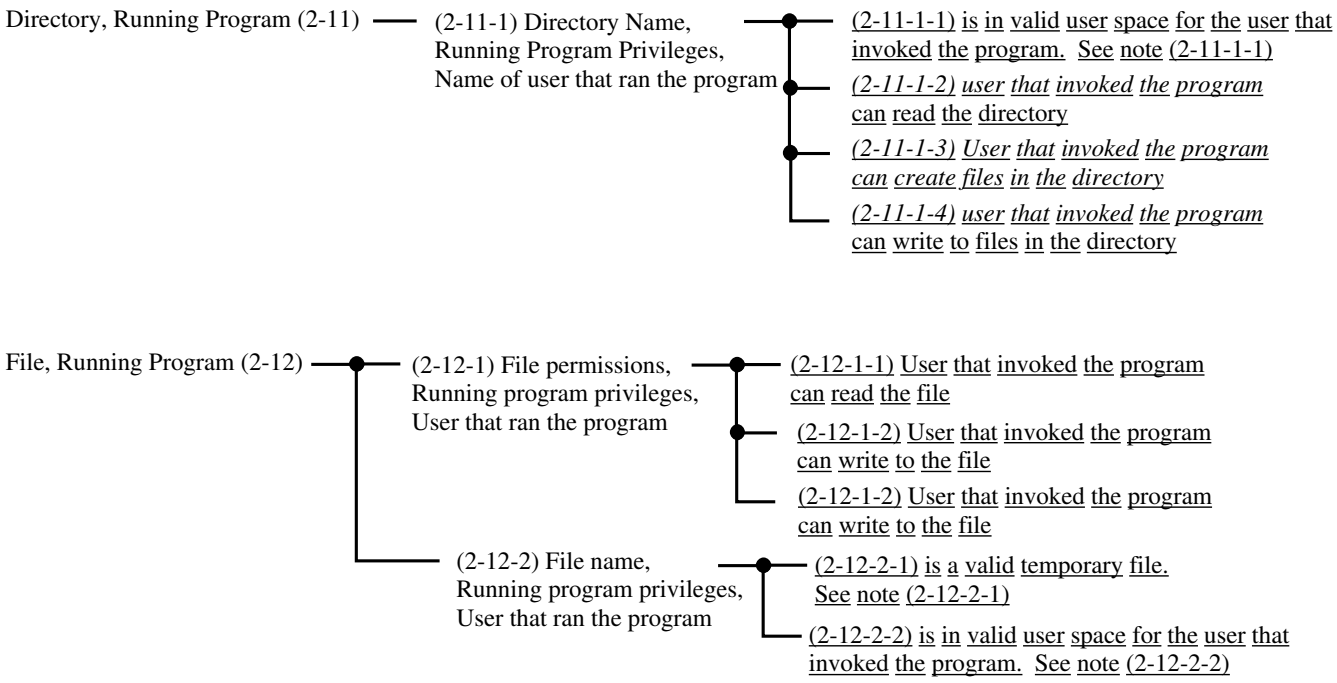


Figure 6.12: Taxonomy of Software Vulnerabilities, Levels 2-11 and 2-12

6.1.1 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]. Section 6.1.3 gives two examples of such formalization.

6.1.2 Application of the Taxonomy of Software Vulnerabilities

The taxonomy of software vulnerabilities presented in Section 6.1 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 6.1, the compiler can be extended to enforce the constraint on the attribute of the object specified for the class. In this section we suggest modifications that can be made automatically to a program to achieve this goal.

Class 2-1-3-1: The compiler can add an assertion, at the point of the pragma or directive that checks that the environment conforms to the definition of "`system()` safety."

Class 2-1-1-1: The compiler can add a check in the beginning of the program to perform a string comparison on the PATH environment variable looking for the required path.

Classes 2-1-1-2, 2-1-1-3, 2-1-2-1 and 2-1-2-2: The compiler can add to the beginning of the program an assertion that checks that the execution path contains at most (or at least) x characters, and that checks that the program name does not contain shell meta-characters or the length is at most x.

Class 2-2-1-1, 2-2-1-2, 2-2-1-3, and 2-2-1-4: When a pragma identifies a particular input statement of the program as making these assumptions, the compiler can generate or replace the input sequence with code that removes meta-characters from the input or truncates lines that are longer than a specified value.

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. Section 6.1.3 gives examples of how the constraints for the classes in the classification can be transformed into policies in this model.

6.1.3 Formalization of the Taxonomy of Vulnerabilities

The taxonomy developed in this section identifies the environmental assumptions informally. A formal definition of the classes in the classification in this taxonomy can be developed by using the model proposed in [Krsul et al. 1998] to specify each class as a policy that must be enforced by the system.

We give two examples of such formalization and argue that similar specifications can be made for the remaining classes. This formalization would allow the policy violation tool

proposed in [Krsul et al. 1998] to detect the violation of these assumptions and hence an environment that can prevent the violation of these assumptions may be possible.

Example 6.7: Define a new class in the classification in Section 6.1 that has the expansion (Running Program — Program Counter (PC) — executes in memory segments explicitly labeled as executable)

There are many variations of programs that contain this vulnerability. The following figure shows three examples of programs that have the buffer overflow vulnerability.

Line	*Form 1*	*Form 2*	*Form 3*
1	`main(int ac,`	`main() {`	`main() {`
2	`  char *av[]) {`	`  p();`	`  p();`
3	`  p(av[1]);`	`}`	`}`
4	`}`		
5		`void p(){`	`void p(){`
6	`void p(char *a){`	`  char b[30];`	`  struct hostent *h;`
7	`  char b[30];`	`  char *p;`	`  sockaddr_in s;`
8			
9	`  strcpy(b,a);`	`  p = getenv("TERM"));`	`  h = gethostbyname(*host);`
10	`}`	`  sprintf(b,"%s",p);`	`  bzero(&s, sizeof s);`
11		`}`	`  s.sin_family =`
12			`       h->h_addrtype;`
13			`  s.sin_port = 25;`
14			`  bcopy(h->h_addr_list[0],`
15			`        &s.sin_addr,`
16			`        h->h_length);`
17			`}`

A program tries to copy data from one object into another, does not check that the destination object is large enough to contain the source object, and uses a routine such as `sprintf` to do the copying.

However, not all programs that share this characteristic are vulnerable. The programs that follow all have buffer overflows but are not vulnerable because either the function never returns—in which case the program never has the opportunity to jump to the code inserted—or the program's buffer is declared static—in which case the program overruns the heap and not the stack.

Line	*Form 1*	*Form 2*
1	`main(int ac,`	`main() {`
2	`  char *av[]) {`	`  p();`
3	`  p(av[1]);`	`}`
4	`}`	
5		`void p(){`
6	`void p(char *a){`	`  static char b[30];`
7	`  char b[30];`	`  char *p;`
8		
9	`  strcpy(b,a);`	`  p = getenv("TERM"));`
10	`  exit(1);`	`  sprintf(b,"%s",p);`
11	`}`	`}`

Figure 6.13 illustrates the ranges where the PC can execute in the cases of non-fragmented code segments, fragmented code segments, and fragmented code segments with dynamic loading of code[2].

Vulnerabilities in the class created for this example are instances of programs where the program counter jumps from an allowable executable region to a region in memory, normally the stack, where it executes arbitrary code. Because there are instances of buffer overflows that cannot be characterized as vulnerabilities, the real issue behind these vulnerabilities is not the buffer overflow but rather what happens when a user can cause the stack pointer to change so that it points back at the stack.

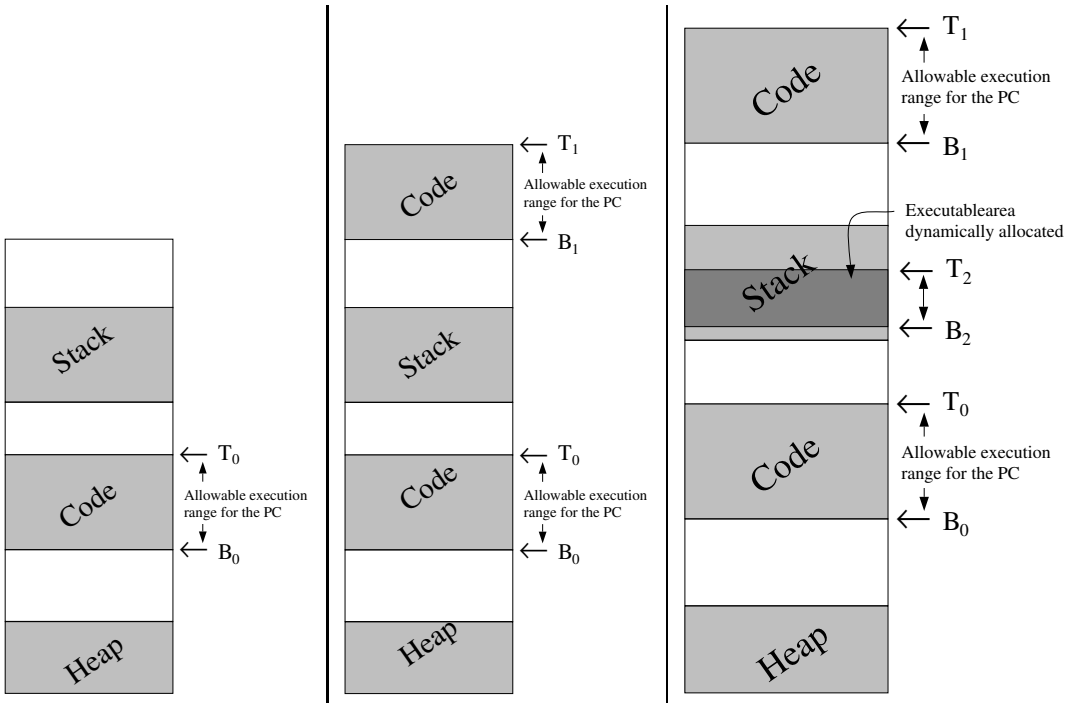
The assumption made by the class defined in this example is that the program's PC should remain within the allowable range. A policy specification for this case can now be generated with the model and notation presented in [Krsul et al. 1998] as follows:

For simplicity, an atomic operation will be axiomatically defined as the execution of any instruction that causes the program counter (PC) to return from a subroutine.

The policy function, shown in Equation 6.1, takes as arguments a system value function, an object value function, and two sets of interest (before and after the execution of an instruction). The function returns `true` if the policy has not been violated and `false` otherwise.

The policy we specify requires that applications execute only the instructions within the bounds defined. The set of interest consists of programs, program counters, and boundaries:

[2]This simplified model does not take into account the system area of memory and this can be incorporated by adding an additional set of segment markers to signal that the PC can execute in system memory.



A program contains three segments that can be distinguished: heap, code, and stack. We expect the PC to remain in the area between the B_0 and T_0 markers.

If the code segment is fragmented, the program counter is expected to remain in the areas between the B_0, T_0, B_1, and T_1 markers.

Dynamic code can be loaded into the stack explicitly (by calling a function that dynamically links the code) creates another area where the PC is allowed to execute: the area between the B_2 and T_2 markers.

Figure 6.13: Programs normally execute code from well defined regions in memory, even if the memory is fragmented or the program contains dynamic executable code.

Programs:

♢ Set of boundaries: b.

♢ Set of program counter locations: pc.

Boundary:

♢ Top of allowed segment: T.

♢ Bottom of allowed segment: B.

$$
\begin{array}{l}
\mathit{Policy} : \mathit{System\ Value\ function} \times \mathit{Object\ Value\ function} \times \\
\qquad \textbf{set of interest} \times \textbf{set of interest} \rightarrow \textbf{boolean} \\
\textbf{fun} \quad \mathit{Policy}\,(\mathit{Value}, v, I_i, I_{i+1}) ::= \\
\qquad \textbf{if}\ \mathit{Value}\,(I_i, v) \leq \mathit{Value}\,(I_{i+1}, v)\ \ \textbf{then} \\
\qquad\qquad \mathit{Policy} := \text{true}; \\
\qquad \textbf{else} \\
\qquad\qquad \mathit{Policy} := \text{false}; \\
\qquad \textbf{fi} \\
\textbf{nuf}
\end{array}
\tag{6.1}
$$

$$
\begin{array}{l}
\mathit{Value} : \textbf{set of interest} \times \mathit{Object\ value\ function} \rightarrow \textbf{integer} \\
\textbf{fun} \quad \mathit{Value}\,(S, v) ::= \\
\qquad \mathit{Value} := \sum_{x \in S} v\,(x, S - x) \quad \forall x \in S; \\
\textbf{nuf}
\end{array}
\tag{6.2}
$$

Program Counter:

◇ Location: l.

The system value function for the policy is shown in Equation 6.2, and the object value function that can be used to implement the desired policy is shown in Equation 6.3.

□

Example 6.8:

Consider the class 2-5-2-1 (Command Line Parameters — Content Type — is 7 bit ASCII). We can specify this constraint with a policy, as required in [Krsul et al. 1998], as follows:

The set of interest is all the programs in the system. We require that each program where the programmers makes the assumption 2-5-2-1 be marked appropriately.

Object Attributes:

◇ A boolean flag indicating if assumption 2-5-2-1 is made: *o.a.*

◇ A set of command line arguments given to the program: *o.c.*

The functions that specify this policy are shown in Equations 6.1, 6.2, and 6.4

□

```
v : object of interest × set of object of interest → integer
fun  v(o, S) ::=
     v := 0;
     if o is a program then
          ∀x ∈ o.pc do
               m := 0;
               ⇒ Check to see if the PC is in a correct range ⇐
               ∀y ∈ o.b do                                          (6.3)
                    m := 1 if x.l ≥ y.B ∧ x.l ≤ y.T;
               od
               ⇒ Violation if we did not find the PC in a valid range ⇐
               v := v − 1 if m = 0;
          od
     fi
nuf
```

```
v : object of interest × set of object of interest → integer
fun  v(o, S) ::=
     v := 0;
     if o.a then
          ⇒ Only check the programs that make assumption 2-5-2-1. ⇐
          v := 0;
          ∀y ∈ o.c do
               ⇒ Check every command line argument. ⇐
               x := 0;
               x < y.length do
                    ⇒ Value of the system drops if a character is not 7 bit ASCII. ⇐
                    v := v − 1 if is7BitAscii(▷y(x, x)) = false
                    x := x + 1;                                     (6.4)
               od
          od
     fi
nuf
```

```
is7BitAscii : character → boolean
fun  is7BitAscii(c) ::=
     is7BitAscii := false;
     ⇒ We define 7 bit ASCII to be 32 - 126. ⇐
     is7BitAscii := true if (t ≥ 32 ∧ t ≤ 126)
nuf
```

6.2 Evolutionary Classification

An evolutionary classification of software vulnerabilities could group vulnerabilities in time and (logical) space much like the current phylogenic classification of living organisms [Maddison and Maddison 1996]. For software vulnerabilities a time-line could be established by considering the release date of programs. This would allow us to determine nearness in time between vulnerable programs.

The computer analogy to *space* could be established by considering, for example, programmers and operating systems. The combination of time of release of a particular version of a program, programmer, and operating system would allow the determination of propinquity of descent—necessary for the development of a phylogeny of software vulnerabilities. For example, Figure 6.14 shows a possible subtree for an evolutionary classifications of software vulnerabilities.

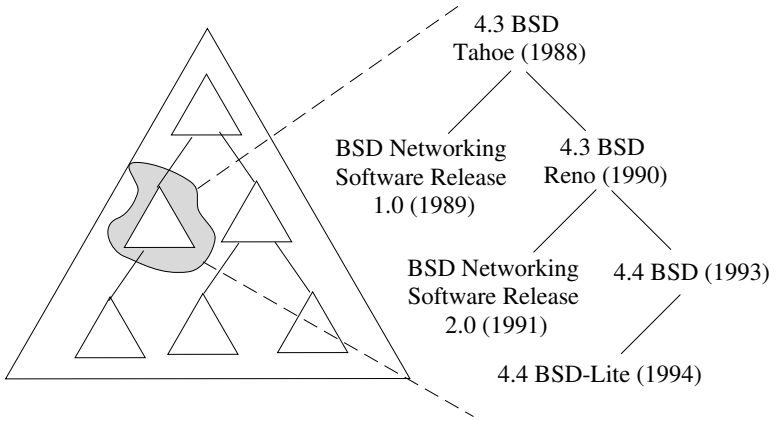


Figure 6.14: A possible subtree of an evolutionary classification of software vulnerabilities. Historical data extracted from [Stevens 1998]

Programmers use, in the development of complex code, a limited set of programming constructs and the programming style, characteristics, and environmental assumptions made by the programmer are likely to be similar in programs written by the same author at approximately the same time. [Krsul and Spafford 1997]. Based on this result, we can postulate that it is possible that similar programs written by the same programmers

at approximately the same time may have similar vulnerabilities. Hence, this classification is likely to have predictive and descriptive value.

The creation of evolutionary classifications for software vulnerabilities is, at this point, a theoretical possibility. A difficulty with this classification is that computer programs do not have to share the characteristics of their predecessors, however these were defined. The resemblance is only a statistical probability.

6.3 A Classification for Software Testing

Wenliang Du started the development of a classification of computer vulnerabilities based on environmental assumptions. This work has resulted in a software testing technique that perturb the environment and observes software system's behavior under this perturbation. The testing technique should reveal whether software systems are making such assumptions [Du and Mathur 1998]. This classification is an example of how the purpose of the classification mandates its form. The first two levels of this classification are shown in Figure 6.15.

This classification divides vulnerabilities into two groups: a group where the environment affects the value of an internal entity (i.e. a variable) that in turn affects the execution of the program, and another group where the environment affects the execution of the program without affecting the value of an internal entity. Subsequent divisions in the classification tree are performed according to the characteristic of the environment, starting with the source of the environment entity (i.e. network, user input, environment variable, etc.)

Du's classification satisfies the requirements for a classification and is useful because each vulnerability class represents a group of vulnerabilities caused by certain environmental assumption, and leads to a specific set of perturbations that can be performed on the environment to test the software for vulnerabilities.

6.4 Chapter Summary

In Section 3.1 we note that taxonomies establish organizing frameworks, are essential for the development of a field, and that a function of these taxonomies is the separation or

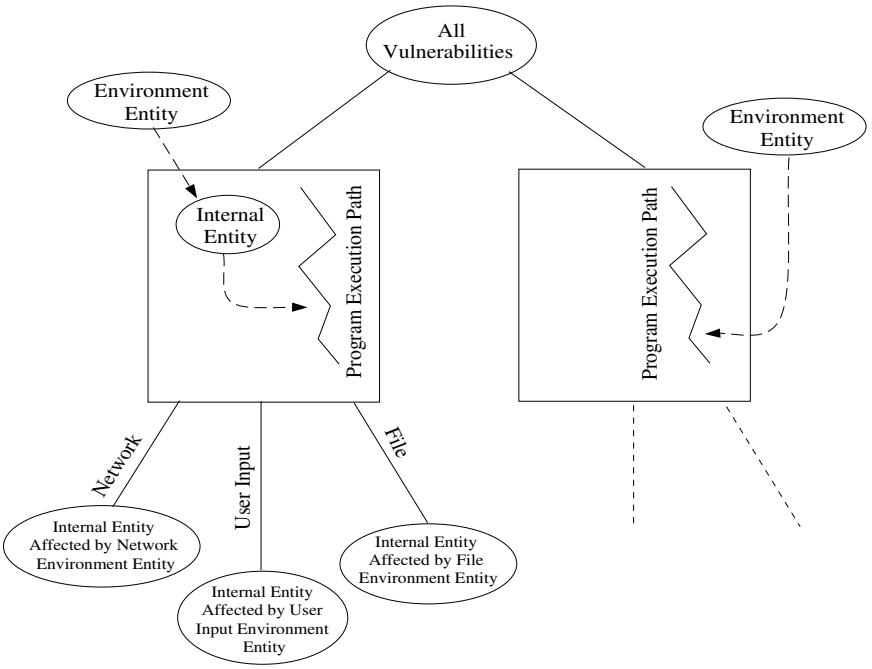


Figure 6.15: An example of a goal-oriented classification for software testing using environmental perturbations.

ordering of specimens so that generalizations can be made about them. We also note that classifications have *explanatory* and *predictive* value.

In this chapter we presented a taxonomy of software vulnerabilites that has these properties, and that constributes to our understanding of the nature of software vulnerabilities. In this chapter we also presented two conceptual goal-oriented classifications that may have similar properties.

7 SUMMARY, CONCLUSIONS, AND FUTURE DIRECTIONS

7.1 Conclusions

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].

As mentioned in Section 1.1, 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 dissertation 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 3.1—to the data collected for the vulnerability database resulted in the classification for software vulnerabilities presented in Section 6.1.

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 6.1.2, 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 dissertation provides the desired increase in our understanding of vulnerabilities.

7.2 Summary of Main Contributions

- Provide a unifying definition of *software vulnerability*.
- Show that existing classifications and taxonomies for software vulnerabilities, or related fields, fail because they do not satisfy the desirable properties for classifications and taxonomies.
- Define the properties of measurements or observations necessary for the development of classifications, and provide a framework for the development of classifications and taxonomies for software vulnerabilities and related fields.
- Collected a representative sample of existing software vulnerabilities. For each sample collected we measured a series of features that can be used for the generation of classifications.
- Developed an extension to the classification of vulnerabilities presented in [Aslam et al. 1996]. Unlike its predecessor, this classification 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.
- Identified patterns and regularities in vulnerabilities with the application of co-word analysis induction decision trees, and data visualization tools. These patterns were identified from the taxonomic characters described in Chapter 4.
- Developed an extensible database of software vulnerabilities, a keyword-analysis tool built based on [Coulter et al. 1997], and a data sets generator for the classification tool LNKnet [Kukolich and Lippmann 1995].

- Suggested approaches for the improvement of software to eliminate most software vulnerabilities that result from a mismatch between the programmer's expectation and the environmental conditions in which programs execute.

7.3 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 6.1 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 dissertation 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.

The co-word analysis results presented in Section 5.3.1 suggest that software vulnerabilities frequently apply to more than one variant of UNIX even when these do not share the same code-base. Statistical techniques such as contingency tables (see [Kaufman and Rousseeuw 1990]) may be useful in confirming this result.

BIBLIOGRAPHY

BIBLIOGRAPHY

Abbott, R. P. et al. 1976. Security Analysis and Enhancements of Computer Operating Systems. Tech. Rep. NBSIR 76-1041, Institute for Computer Science and Technology, National Bureau of Standards.

Abbott, R. P., Chin, J. S., Donnelley, J. E., Konigsford, W. L., Tokubo, S., and Webb, D. A. 1976. Security Analysis and Enhancement of Computer Operating Systems. Tech. Rep. NBSIR 76-1041, Institute for Computer Science and Technology, National Bureau of Standards.

Agrawal, R. and Srikant, R. 1994. Fast Algorithms for Mining Association Rules. In *Proceedings of the 20th International Conference on Very Large Databases*. Santiago, Chile.

Air Force Information Warfare (AFIW) Center. 1996. CMET Vulnerability Database. Unpublished Database.

Albitz, P. and Liu, C. 1992. *DNS and BIND*. Help for Unix System Administrators. O'Reilly & Associates.

Amoroso, E. 1994. *Fundamentals of Computer Security Technology*. Prentice Hall.

Anderson, R. 1994. Why Cryptosystems Fail. Tech. rep., University Computer Laboratory, Cambridge. January.

Aslam, T. 1995. A Taxonomy of Security Faults in the Unix Operating System. M.S. thesis, Purdue University.

Aslam, T., Krsul, I., and Spafford, E. 1996. Use of A Taxonomy of Security Faults. In *19th National Information Systems Security Conference Proceedings*. Baltimore, Maryland.

Audi, R., Ed. 1995. *The Cambridge Dictionary of Philosophy*. Cambridge University Press.

AUSCERT Coordination Center. 1998. Unnamed Vulnerability Database. Unpublished Database.

Bach, M. J. 1986. *The Design of the UNIX Operating System*. Software Series. Prentice-Hall.

Bahr, L. S. and Johnston, B. 1995. *Collier's Encyclopedia: With Bibliography and Index*. P.F. Collier, New York.

Bailey, G. 1987. *Bibliography of Soil Taxonomy.* Wallingford.

Basili, V. and Perricone, B. 1984. Software Errors and Complexity. *Communications of the ACM 27,* 1 (January), 42–52.

Beizer, B. 1983. *Software Testing Techniques.* Electrical Engineering/Computer Science and Engineering Series. Van Nostrand Reinhold.

Bell, D. E. and LaPadula, L. J. 1973. Secure Computer Systems: Mathematical Foundations and Model. Tech. Rep. M74-244, The MITRE Corporation. May.

Bhushan, A., Braden, B., Crowther, W., Heafner, E. H. J., McKenzie, A., Melvin, J., Sundberg, B., Watson, D., and White, J. 1971. *The File Transfer Protocol.* RCF-172.

Bibsey, R., Popek, G., and Carlstead, J. 1975. Inconsistency of a Single Data Value over time. Tech. rep., Information Sciences Institute, University of Southern California. December.

Bier, E. A., Fishkin, K., Pier, K., and Stone, M. C. 1995. Taxonomy of See-Through Tools: The Video. In *Proceedings of the Conference on Human Factors in Computing. Part 2 (of 2).* Vol. 2. ACM, Denver, Colorado, 411–412.

Bishop, M. 1986. Analyzing the Security of an Existing Computer System. In *Proceedings of the Fall Joint Computer Conference.* 1115–1119.

Bishop, M. 1995. A Taxonomy of UNIX System and Network Vulnerabilities. Tech. Rep. CSE-95-10, Department of Computer Science at the University of California at Davis.

Bishop, M. and Bailey, D. 1996. A Critical Analysis of Vulnerability Taxonomies. Tech. Rep. CSE-96-11, Department of Computer Science at the University of California at Davis. September.

Bishop, M. and Dilger, M. 1996. Checking for Race Conditions in File Accesses. *Computing Systems 9,* 2, 131–152.

Borenstein, N. 1992. *MIME (Multipurpose Internet Mail Extensions): Mechanisms for Specifying and Describing the Format of Internet Message Bodies.* RFC-1391.

Breiman, L. 1994. Bagging Predictors. Tech. Rep. TR 421, Department of Statistics, University of California. September.

Breslow, L. A. and Aha, D. W. 1996. Simplifying Decision Trees: A Survey. Tech. Rep. NCARAI Technical Report No. AIC-96-014, Navy Center for Applied Research in Artificial Intelligence, Washington, D.C.

Bretz, R. 1971. *A Taxonomy of Communication Media.* Educational Technology Publications.

Brooks, F. P. 1995. *The Mythical Man-Month.* Addison-Wesley.

Carlstead, J., Bibsey II, R., and Popek, G. 1975. Pattern-Directed Protection Evaluation. Tech. rep., Information Sciences Institute, University of Southern California. June.

CERT Coordination Center. 1997c. CERT Summary CS-97.03. `ftp://ftp.cert.org/pub/cert_summaries/CS-97.03`.

CERT Coordination Center. 1997b. CERT Summary CS-97.05. `ftp://ftp.cert.org/pub/cert_summaries/CS-97.05`.

CERT Coordination Center. 1997a. CERT Summary CS-97.06. `ftp://ftp.cert.org/pub/cert_summaries/CS-97.06`.

CERT Coordination Center. 1998b. CERT Summary CS-98.02. `ftp://ftp.cert.org/pub/cert_summaries/CS-98.02`.

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

CERT Coordination Center. 1998c. Security improvement. `http://www.cert.org/nav/securityimprovement.html`.

CERT Coordination Center. 1998d. Unnamed Vulnerability Database. Unpublished Database.

Cohen, E. 1990. *Programming in the 1990s.* Springer-Verlag.

Cohen, F. 1997a. Information System Attacks: A Preliminary Classification Scheme. *Computers & Security 16,* 1, 29–46.

Cohen, F. 1997b. Information System Defenses: A Preliminary Classification Scheme. *Computers & Security 16,* 2, 94–114.

Cohen, F. B. 1995. *Protection and Security on the Information Superhighway.* John Wiley & Sons, Inc.

Comer, D. 1984. *Operating System Design: The XINU Approach.* Prentice Hall.

Conte, S., Dunsmore, H., and Shen, V. 1986. *Software Engineering Metrics and Models.* The Benjamin/Cummings Publishing Company.

Coulter, N., Monarch, I., and Konda, S. 1997. Software Engineering as Seen Through Its Research Literature: A Study in Co-Word Analysis. To appear in the Journal of the American Society for Information Science (JASIS).

Crosbie, M., Dole, B., Ellis, T., Krsul, I., and Spafford, E. 1996. IDIOT - Users Guide. Tech. Rep. TR-96-050, Purdue University. September.

Daniels, T. 1998. Qualifying Examination Preparation. Interview.

Daniels, T., Krsul, I., Spafford, E., and Tripunitara, M. 1998b. An analysis of some vulnerabilities related to TCP/IP. In preparation.

Daniels, T., Krsul, I., Spafford, E., and Tripunitara, M. 1998a. Computer vulnerability analysis. Submitted to the 21st National System's Security Conference.

Dasgupta, P., LeBlanc, R. J., Ahmad, M., and Ramachandran, U. The Clouds Distributed Operating System. Georgia Tech Technical Report.

Dean, D., Felten, E. W., and Wallach, D. S. 1996. Java Security: From HotJava to Netscape and Beyond. In *Proceedings of the IEEE Computer Society Symposium on Research in Security and Privacy 1996*. Princeton University, 190–200.

Dean, D. and Wallach, D. S. 1995. Security Flaws in the HotJava Web Browser. Tech. Rep. 501-95, Department of Computer Science, Princeton University. November.

DeMillo, R. A. and Mathur, A. P. 1995. A Grammar Based Fault Classification Scheme and its Application to the Classification of the Errors of TeX. Tech. Rep. SERC-TR-165-P, Software Engineering Research Center, Purdue University. September.

DeMillo, R. A., McCracken, W. M., Martin, R. J., and Passafiume, J. F. 1987. *Software Testing and Evaluation.* The Benjamin/Cummings Publishing Company Inc.

Denning, D. E. 1983. *Cryptography and Data Security.* Addison-Wesley Publishing Company.

Denning, D. E. 1987. An Intrusion-Detection Model. *IEEE Transactions on Software Engineering SE-13,* 2 (February), 222–232.

Dijker, B. L. 1996. A Guide to Developing Computing Policy Documents. The USENIX Association for SAGE, the System Administrators Guild.

Dilger, M. 1995. Unnamed Vulnerability Database. Unpublished Database.

DoDCSEC 1985. DoD 5200.28-STD, Department of Defense Trusted Computer Systems Evaluation Criteria.

DoDISPR 1982. DoD 5200.1R, The Department of Defense Information Security Program Regulation.

Dodson, J. 1996. Specification and Classification of Generic Security Flaws for the Tester's Assistant Library. M.S. thesis, University of California at Davis.

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

Du, W. and Mathur, A. P. 1998. Vulnerability testing of software using fault injection. Tech. rep., Purdue University.

Duda, R. and Hart, P. 1973. *Pattern classification and scene analysis.* Wiley, New York.

Durant, W. 1961. *The Story of Philosophy: The Lives and Opinions of the Great Philosophers of the Western World.* Simon and Schuster.

EBRIT 1997. Britannica Online version 97.1.1. `http://www.britannica.com`.

Edwards, D. 1995. *Recent Advances in Descriptive Multivariate Analysis.* Royal Statistical Society Lecture Note Series. Clarendon Press, Oxford, Chapter 7—Graphical Modelling, 135–156.

Eisenstadt, M. 1997. My Hariest Bug War Stories. *Communications of the ACM 40,* 4 (April).

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

Engler, D. 1998. Incorporating application semantics and control into compilation. Published Electronically at `http://www.pdos.lcs.mit.edu/~engler/magik.ps`.

Farmer, D. and Spafford, E. H. September 1991. The COPS Security Checker System. Tech. Rep. CSD-TR-993, Software Engineering Research Center, Purdue University.

Fink, G., Ko, C., Archer, M., and Levitt, K. 1994. Toward a Property-based Testing Environment with Application to Security Critical Software. In *Proceedings of the 4th Irvine Software Symposium.* 39–48.

Firosoft Consulting. 1998. Online Database Firosoft. Published electronically at `http://www.firosoft.com/security/philez/`.

Freund, Y. and Schapire, R. E. 1996. Experiments with a New Boosting Algorithm.

Garfinkel, S. and Spafford, G. 1996. *Practical UNIX and Internet Security*, Second Edition ed. O'Reilly & Associates, Inc.

Gavin, P. 1998. Designing Secure Software. *SunWorld.* Published electronically at `http://www.sun.com/sunworldonline/`.

Ghezzi, C., Jazayeri, M., and Mandrioli, D. 1991. *Fundamentals of Software Engineering*, First ed. Prentice Hall.

Glass, R. L. and Vessey, I. 1995. Contemporary Application-Domain Taxonomies. *IEEE Software 12,* 4 (July), 63–76.

Grolier Incorporated. 1993. *Encyclopedia Americana*, Deluxe Library Edition ed. Grolier Inc.

Haystack Labs, Inc. 1996. Unnamed Vulnerability Database. Unpublished Database.

Hedberg, S. R. 1995. The Data Gold Rush. *BYTE.*

Hertz, J., Krogh, A., and Palmer, R. G. 1991. *Introduction to the Theory of Neural Computation.* Addison-Wesley Publishing Company.

Holsheimer, M. and Siebes, A. 1994. Data Mining: The Search for Knowledge in Databases. Tech. Rep. CS-R9406, Centrum voor Wiskunde en Informatica.

Hopcroft, J. and Ullman, J. 1979. *Introduction to Automata Theory, Languages, and Computation*, First ed. Addison-Wesley.

HOWARD, J. D. 1997. An Analysis of Security Incidents On The Internet: 1989 - 1995. Ph.D. thesis, Carnegie Mellon University.

IEEE. 1990. ANSI/IEEE Standard Glossary of Software Engineering Terminology. IEEE Press.

INFILSEC SYSTEMS SECURITY. 1998. Online Database INFILSEC Vulnerability Engine. Published electronically at `http://www.infilsec.com/vulnerabilities/`.

INTERNET SECURITY SERVICES. 1998. Online Database X-Force. Published electronically at `http://www.iss.net/xforce/`.

JAIN, A. K. AND DUBES, R. C. 1988. *Algorithms for Clustering Data.* Prentice Hall.

KAO, I.-L. AND CHOW, R. 1995. Enforcement of Complex Security Policies with BEAC. In *Proceedinngs of the 18th National Information Systems Security Conference.* Vol. I. National Institute of Standards and Technology/National Computer Security Center, 1–10.

KAUFMAN, L. AND ROUSSEEUW, P. J. 1990. *Finding Groups in Data.* Wiley Series in Probability and Mathematical Statistics. John Wiley & Sons, Inc., New York.

KDDSIF 1998. S*i*ftware: Tools for Data Mining and Knowledge Discovery. `http://info.gte.com/~kdd/siftware.html`.

KERNIGHAN, B. W. AND RITCHIE, D. M. 1988. *The C Programming Language*, Second Edition ed. Prentice Hall Software Series. Prentice Hall.

KICZALES, G., JOHN LAMPING, A. M., MAEDA, C., LOPES, C., MARC LOINGTIER, J., AND IRWIN, J. 1997. Aspect-Oriented Programming. Tech. Rep. PARC Technical Report SPL97-008 P9710042, Xerox PARC. February.

KNUTH, D. E. 1989. The Errors of TEX. *Software—Practice and Experience 19,* 7 (July), 607–685.

KOHAVI, R., SOMMERFIELD, D., AND DOUGHERTY, J. 1997. Data Mining using MLC++, a Machine Learning Library in C++. *International Journal of Artificial Intelligence Tools 6,* 4, 537–566.

KOLAWA, A. AND HICKEN, A. 1997. Insure++ A Tool To Support Total Quality Software. `http://www.parasoft.com/insure/papers/tech.htm`.

KRSUL, I. 1998. Vulnerability Database User Manual. Tech. Rep. COAST Technical Report No. 98-08, Purdue University. May.

KRSUL, I., DANIELS, T., DU, W., AND WILSON, A. 1998. COAST Vulnerabilty Database.

KRSUL, I., DANIELS, T., AND WILSON, A. 1998. COAST Vulnerability Database. Available by request from the COAST Laboratory.

KRSUL, I. AND SPAFFORD, E. 1997. Authorship analysis: identifying the author of a program. *Computers & Security 16,* 3, 233–257.

Krsul, I., Spafford, E., and Tuglular, T. 1998. A New Approach to the Specification of General Computer Security Policies. Tech. Rep. COAST Technical Report 97-13, COAST Laboratory, Department of Computer Sciences, Purdue University. January.

Kukolich, L. and Lippmann, R. 1995. *LNKnet User's Guide.* MIT Lincoln Laboratory, Carleton Street, Cambridge, Massachusetts.

Kumar, S., , and Spafford, E. 1995. A Taxonomy of Common Computer Security Vulnerabilities Based on their Method of Detection. Tech. rep., Purdue University.

Kumar, S. 1995. Classification and Detection of Computer Intrusions. Ph.D. thesis, Purdue University.

Kumar, S. and Spafford, E. 1994. A Pattern Matching Model for Misuse Intrusion Detection. In *17th National Computer Security Conference.*

Landwher, C., Bull, A., McDermott, J., and Choi, W. 1993. A Taxonomy of Computer Program Security Flaws. Tech. Rep. NRL/FR/5542–93-9591, Naval Research Laboratory. November.

Leveson, N. 1994. High-pressure Steam Engines and Computer Software. *Computer 27,* 10 (October), 65–73.

Leveson, N. 1995. *Safeware: System Safety and Computers.* Addison-Wesley Publishing Co.

Levy, M. and Salvadori, M. 1992. *Why Buildings Fall Down.* W. W. Norton & Company.

Linde, R. R. 1975. Operating system penetration. In *National Computer Conference.*

Longley, D. and Shain, M. 1990. *The Data and Computer Security Dictionary of Standards, Concepts, and Terms.* Macmillan Stockton Press.

Longstaff, T. 1997. Update: CERT/CC Vulnerability Knowledgebase. Technical presentation at a DARPA workshop in Savannah, Georgia.

Maddison, D. and Maddison, W. 1996. The Tree of Life: A distributed Internet project containing information about phylogeny and biodiversity. Internet address: `http://phylogeny.arizona.edu/tree/phylogeny.html`.

Marick, B. 1990. A survey of software fault surveys. Tech. Rep. UIUCDCS-R-90-1651, University of Illinois at Urbana-Champaign. December.

Marick, B. 1995. *The Craft of Software Testing.* Prentice Hall.

McGraw, G. and Felten, E. W. 1997. *Java Security: Hostile Applets, Holes and Antidotes.* John Wiley & Sons, Inc.

Miller, B., Fredrikson, L., and So, B. 1990. An Embirical Study of the Reliability of UNIX Utilities. *Communications of the ACM 33,* 12 (December), 32–44.

Miller, B. P., Koski, D., Lee, C. P., Maganty, V., Murthy, R., Natarajan, A., and Steidl, J. 1995. Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services. Tech. rep., Computer Science Department, University of Wisconsin. November.

Mockapetris, P. 1987. *Domain Names – Concepts and Facilities.* RFC-1031.

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

Netscape Communications Corporation. 1996. Unnamed Vulnerability Database. Unpublished Database.

Neumann, M. 1995. Unnamed Vulnerability Database. Unpublished Database.

Olivier, M. and Vonsolms, S. H. 1994. A Taxonomy for Secure Object-Oriented Databases. *ACM Transactions on Database Systems 19,* 1 (March), 3–46.

Oman, P. and Cook, C. 1990. A Taxonomy for Programming Style. In *Eighteenth Annual ACM Computer Science Conference Proceedings.* ACM, 244–247.

Oman, P. and Cook, C. 1991. A Programming Style Taxonomy. *Journal of Systems Software 15,* 4, 287–301.

Oram, A. and Talbott, S. 1993. *Managing Projects with make.* O'Reilly & Associates, Inc.

Ostrand, T. and Weyuker, E. 1984. Collecting and Categorizing Software Error Data in an Industrial Environment. *The Journal of Systems and Software 4,* 289–300.

OXFORD 1998. The Oxford English Dictionary. `http://oed.purdue.edu/`.

Perrow, C. 1984. *Normal Accidents: Living With High-Risk Technologies.* Basic Books.

Perry, T. and Wallich, P. 1984. Can Computer Crime be Stopped. *IEEE Spectrum 21,* 5.

Petrosky, H. 1985. *To engineer is human: the role of failure in successful design.* St. Martin's Press.

Polk, W. T. 1992. Automated Tools for Testing Computer System Vulnerability. Unknown if a published version of the paper exists.

Popper, K. R. 1969. *Conjections and Refutations.* Routledge and Kegan Paul.

Postel, J. 1980. *User Datagram Protocol.* RFC-793.

Postel, J. 1981a. *Internet Protocol – DARPA Internet Program Protocol Specification.* RFC-791.

Postel, J. 1981b. *Transmission Control Protocol – DARPA Internet Program Protocol Specification.* RFC-793.

Power, R. 1996. Current And Future Danger: A CSI Primer of Computer Crime & Information Warfare. CSI Bulletin.

Quinlan, J. R. 1986. Induction of Decision Trees. *Machine Learning*, 81–106.

Quinlan, J. R. and Cemeron-Jones, R. M. 1995. Oversearching and Layered Search in Empirical Learning. In *Proceedings Fourteenth International Joint Conference on Artificial Intelligence*. Morgan Kaufmann, Montreal, 1019–1024.

Quinlan, R. J. 1993. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers, Inc., Los Altos, California.

Roskos, J. E., Welke, S. R., Boone, J. M., and Mayfield, T. 1990. A Taxonomy of Integrity Models, Implementations and Mechanisms. In *Proceedings of the 13th National Computer Security Conference*. Vol. II. National Institute of Standards and Technology/National Computer Security Center, 541–551.

Schlager, N. 1994. *When Technology Fails: Significant Technological Disasters, Accidents, and Failures of the Twentieth Century*. Gale Research Inc.

Schuba, C. L., Krsul, I. V., Kuhn, M. G., Spafford, E. H., Sundaram, A., and Zamboni, D. 1997. Analysis of a Denial of Service Attack on TCP. In *Proceedings IEEE Symposium on Security and Privacy*.

SECJAVA 97. Secure Computing with Java: Now and the Future. Java Whitepaper Published Electronically at `http://www.javasoft.com/marketing/collateral/security.html`.

Sethi, R. 1989. *Programming Languages Concepts and Constructs*. Addison–Wesley Publishing Company.

Simpson, G. G. 1945. *The Principles of Classification and a Classification of Mammals*. New York.

Simpson, G. G. 1961. *Principles of Animal Taxonomy*. Columbia University Press.

Smith, D. 1994. Enhancing Security of Unix Systems. `http://www.usq.edu.au/ww94/papers/unix-security.html`.

Spafford, E. H. 1989. The Internet Worm Program: An Analysis. *Computer Communication Review 19,* 1 (January).

Spencer, D. 1983. *The Illustrated Computer Dictionary*, First ed. Merrill Publishing Co.

Stern, H. 1991. *Managing NFS and NIS*. O'Reilly & Associates, Inc.

Sterne, D. F., Branstad, M. A., Hubbard, B. S., Mayer, B. A., and Wolcott, D. M. 1991. An Analysis of Application Specific Security Policies. In *Proceedings of the 14th National Computer Security Conference*. Vol. I. 25–36.

Stevens, R. W. 1998. UNIX *Network Programming*, Second Edition ed. Prentice Hall.

Stout, B. 1998. Online Database Known NT Exploits. Published electronically at `http://www.emf.net/~ddonahue/NThacks/ntexploits.htm`.

Sun Microsystems Inc. 1988. *RPC: Remote Procedure Call Protocol Specification.* RFC-1050.

Sun Microsystems Inc. 1989. *NFS: Network File System Protocol Specification.* RFC-1094.

Sun Microsystems Inc. 1997. Unnamed Vulnerability Database. Unpublished Database.

Swayne, D. F., Cook, D., and Buja, A. 1998. XGobi: Interactive Dynamic Data Visualization in the X Window System. *Journal of Computational and Graphical Statistics 7,* 1 (March).

Tanenbaum, A. S. 1987. *Operating Systems Design and Implementation.* Prentice Hall.

Thompson, W. R. 1852. Philosophical Foundations of Systematics. *Canadian Entomologist 84*, 1–16.

VDBBOD 1998. Online Database The Brotherhood of Darkness Exploit Archive. Published electronically at `http://www.ilf.net/brotherhood/filez/xploits.html`.

VDBBUG 1998. Online Database Security Bugware. Published electronically at `http://161.53.42.3/~crv/security/bugs/Others/other.html`.

VDBDOP 1998. Online Database Dop's terribly geeky page of naughty hacks. Published electronically at `http://www.geek-nation.com/~dop/`.

VDBELI 1998. Online Database Elitehackers.org. Published electronically at `http://www.elitehackers.org/Exploits/index.html`.

VDBEXP 1998. Online Database Exploit World. Published electronically at `http://www.dhp.com/~fyodor/sploits_all.html`.

VDBFIR 1998. Online Database Rootshell. Published electronically at `http://www.rootshell.com/`.

VDBFKI 1998. Online Database Future Kill Security Database. Published electronically at `http://main.succeed.net/~kill9/security/database/index.html`.

VDBKAO 1998. Online Database Kao's UNIX Security Library. Published electronically at `http://www.tacd.com/exploit/expmain.htm`.

VDBLEG 1998. Online Database The Legacy Hacking Archive II. Published electronically at `http://www.jabukie.com/ArchiveII.html`.

VDBNN1 1998. Unnamed Online Database. Published electronically at `http://www.cultdeadcow.com/~gauss/exploits/`.

VDBNN2 1998. Unnamed Online Database. Published electronically at `http://get.your.exploits.org/`.

VDBNN3 1998. Unnamed Online Database. Published electronically at `http://www.enslaver.com/exploit/`.

VDBNN4 1998. Unnamed Online Database. Published electronically at `http://www.outpost9.com/exploits/hp.html`.

VDBNN5 1998. Unnamed Online Database. Published electronically at `http://www.cs.iastate.edu/~ghelmer/unixsecurity/unix_vuln.html`.

VDBNN6 1998. Unnamed Online Database. Published electronically at `http://www.virtual-pc.com/spartan/plaguez/hc1.htm`.

VDBNZE 1998. Online Database NegativeZero Exploit PAge. Published electronically at `http://www.negativezero.com/exploits/`.

Wall, L. and Schwartz, R. 1990. *Programming Perl*, First ed. O'Reilly & Associates, Inc.

Walsh, N. 1994. *Making TEX Work*. O'Reilly & Associates.

WEBOL 1998. Merriam-Webster OnLine: WWWebster Dictionary. `http://www.m-w.com/dictionary.htm`.

Weiss, S. and Kulikowski, C. 1991. *Computer systems that learn: classification and prediction methods from statistics, neural nets, machine learning, and expert systems*. M. Kaufmann Publishers.

Weissman, C. 1995. *Information Security: An Integrated Collection of Essays*. IEEE Computer Society Press, Los Alamitos, California, Chapter Essay 11: Penetration Testing, 269–296.

Whittaker, J. 1989. Creativity and Conformity in Science: Titles, Keywords, and Co-Word Analysis. *Social Science in Science 19*, 473–496.

Young, M. and Taylor, R. N. September 1991. Rethinking the Taxonomy of Fault Detection Techniques. Tech. rep., Software Engineering Research Center, Purdue University.

APPENDICES

A SCHEMAS FOR PRIOR VULNERABILITY DATABASES

A.1 Vulnerabilty Databse at ISS

The vulnerability database at ISS [Internet Security Services 1998] is freely available in the Internet and contains information organized according to the following flat-database schema:

Tag Name: Text

Date Entered: Month/Year

Name: Text

Date Reported: Month/Year

Brief Description: Text

Risk Level: Integer in the range (0–10)

Affected Item: Text

Platforms: One of Unknown, AIX, BSD, HPUX, IRIX, Linux, Solaris, SunOS, Windows, Other

Detailed Description: Text

Discovered By Text

Corrective Actions: Text

Fix Time: One of Unknown, 0.0–0.5 hours, 0.5–1.0 hours, 1.0–1.5 hours, 1.5–2.0 hours, 2.0–2.5 hours, 2.5–3.0 hours, More than 3.0 hours

References: Text

Consequences: One of None, Gain Privileges, Denial of Service, Execute As., Chmod, Chown, Write Files, Read Files, Account Info., Other, Access Resources, Bypass

Comments: Text

Exploit: Text

Contact: Text

A.2 Vulnerabilty Databse at INFILSEC

The vulnerability at INFILSEC [INFILSEC Systems Security 1998] is freely available in the Internet and contains information organized according to the following flat-database schema:

Vulnerability Name: Text	**Exploit by:** Text
Systems Affected: Text	**Exploit Author:** Text
Component: Text	**Fix by:** Text
Impact: Text	**Fix Author:** Text
Author: Text	**Rererences:** Text
Description: Text	

A.3 Vulnerabilty Databse of Michael Dilger

The database maintained by Michael Dilger contains information organized according to the schema:

Revision: This should be incremented if the data is changed. You need not increment this number unless the old revision has been distributed or used elsewhere in some manner.

Vulnerability Name: This should be a simple 2-8 word title. It should include reference to the component vulnerable.

Brief Description: A brief (1 sentence to several paragraphs) high-level description of the attack.

System Impact: What internal systems are affected and to what extent (if relevant)

Vulnerable Systems: Vulnerable versions or revisions (hardware, OS, software, etc). State also for each system one of: verified, unverified, trusted source

Vulnerable States: State in which the affected software is vulnerable (configuration options, etc).

Components: what files/systems/protocols are employed by the attack

Keywords: Key words to index this record with

Introduced: When was the vulnerability introduced. One of: Design flaw, Implementation flaw, Trojaned during implementation, Trojaned after implementation, Configuration flaw.

Announced: When and where this attack was announced.

Vulnerability Class: One or more of:

- Improper choice of protection domain: For example, allowing an ordinary user to modify a log
- Improper isolation of implementation detail. For example, an ordinary user being able to bypass the operating system controls and write to a specific absolute address in memory
- Improper change: Allowing the use of inconsistent data (one process reading a file that another process is altering)
- Improper naming: For example, two programs have the same name
- Improper deallocation or deletion: For example, allowing the reuse of disk space containing confidential information without clearing that information first
- Improper validation: For example, the wrong number or type of parameters, parameters in the wrong order, or being too large or too small, memory references to inaccessible locations,
- Improper indivisibility: For example, non-atomic operations that should be atomic
- Improper sequencing: For example, allowing race conditions among processes vying for resources
- Improper choice of operand or algorithm: For example, unfair scheduling algorithms that starve processes

Protection Domains Involved: One of:

- single, unprivileged protection domain: This means the program does not cross protection domain boundaries, so only the user executing it can be hurt.
- system protection domain: This is a user-level program that is normally run by a system user such as daemon; it is a separate category because it indicates the

damage may affect the system and thus (indirectly) more than one user, but that no protection domain boundaries are crossed

- privileged protection domain: This means the flaw requires the program to be run by the superuser; no ordinary user running the program can trigger it.
- multiple protection domains: This means that the flaw requires access to multiple protection domains; for example, a setuid program. If the combination of 2 programs running at non-privileged but distinct levels causes a problem, that also fits this class.

Access required: This indicates what access an attacker must have to carry out the attack. One of: remote network, local network, user account, physical.

Intrusion Level: This indicates the effect of the intrustion, what the attacker can do with the vulnerability. One of: low-privileged information, denial of service, misplaced trust, [localized] privileged information (read), [localized] data alteration (write), localized access (read/write), user access (user), group access (group), root access (read/write, not localized).

Background: This describes the normal intended operation of the system that fails under the given attack.

Attack description: A full description of the attack, low-level details included.

Vulnerability Analysis: An analysis of the vulnerability itself

Vulnerability Detection: How to detect if you are vulnerable

Vulnerability Correction Admin: How to patch, upgrade, or otherwise fix the problem, from a system administrator's point of view.

Vulnerability Correction Source: How to fix the problem from a system designer's point of view.

Attack Detection: How to detect if you have been attacked, as well as real-time detection

Attack Clean Up: How to clean up after the attack

Related Attacks: Other attacks in this database that are closely related

References: Papers, articles, etc.

Advisories: Names of advisories for further information

Contacts: Vendors and people to contact for further information

Attack Script: Reference to a script which demonstrates the attack

A.4 Eric Miller's Database

Eric Miller's database organizes vulnerabilities according to the following flat-database schema:

Tag Name: Text

Program Exploited: Text

Impact: Text

Requirements: Text

Description: Text

Difficulties to Execution: Text

A.5 The CMET Database at the AFIW

The CMET database of vulnerabilities constructed by the AFIW is organized according to the following relational schema:

Table: ADVISORY		
Description: Table to hold information necessary to link to an advisory, such as local filename, net path/filename, etc.		
Field List	**Data Type**	**Size**
Advisory_ID	Text	50
Date	Date/Time	8
Name	Text	60
Time	Text	4
Remote Site	Text	50
Remote Filename	Text	50
Remote Ext	Text	60
Local_Source	Text	50
Relationships: Attached_Advisory_Table-one to many; on Advisory_ID		

Table: Attached_Advisory_Table		
Description: Table which links an advisory to a vulnerability by common vulnerability_id		
Field List	**Data Type**	**Size**
Counter	Number(Long)	4
Vulnerability_ID	Number(Integer)	2
Advisory_ID	Text	50
Relationships: ADVISORY- many to one; on Advisory_ID, VULNERABILITY- many to one; on Vulnerability_ID		

Table: Attached_OS_Table		
Description: Table which links an operating system to a vulnerability by common vulnerability_id		
Field List	**Data Type**	**Size**
Counter	Number (Long)	4
Vulnerability_ID	Number (Integer)	2
OS_ID	Number (Double)	8
Relationships: OPERATING_SYSTEM- many to one; on OS_ID, VULNERABILITY- many to one; on Vulnerability_ID		

Table: Attached_Platform_Table		
Description: Table which links a platform to a vulnerability by common vulnerability_id		
Field List	**Data Type**	**Size**
Counter	Number (Long)	4
Vulnerability_ID	Number (Integer)	2
Platform_ID	Number (Double)	8
Relationships: PLATFORM- many to one; on Platform_ID, VULNERABILITY- many to one; on Vulnerability_ID		

Table: COUNTERMEASURE		
Description: Table containing information about a countermeasure with a link to a vulnerabilitFy		
Field List	**Data Type**	**Size**
CounterMeasure_ID	Number (Double)	8
Vulnerability_ID	Number (Integer)	2
Name	Text	50
Description	Memo	—
Detection_Method	Text	255
Relationships: VULNERABILITY- many to one; on Vulnerability_ID		

Table: OPERATING_SYSTEM		
Description: Table containing operating system information		
Field List	**Data Type**	**Size**
OS_ID	Number (Double)	8
Name	Text	50
Version	Text	50
Relationships: Attached_OS_Table- one to many; on OS_ID		

Table: PLATFORM		
Description: Table containing different types and models of hardware		
Field List	**Data Type**	**Size**
Platform_ID	Number (Double)	8
Type	Text	50
Model	Text	50
Relationships: Attached_Platform_Table- one to many; on Platform_ID		

Table: HACKER_TOOL		
Description: Table containing information about a hacker tool, linked to vulnerability and a script text file		
Field List	**Data Type**	**Size**
Tool_ID	Number(Integer)	2
Name	Text	18
Vulnerability_ID	Number(Integer)	2
Aliases	Text	255
Location	Text	50
Result	Text	50
Usage	Text	50
Shell	Text	50
Programs_Needed	Text	50
Environment_Variables	Text	50
Exploitation_type	Text	50
Pseudo_Code	Text	50
Errors_in_Tool	Text	50
Functionality	Text	50
Consistency	Yes/No(Boolean)	1
Ease_of_Use	Text	50
Part_of_Setup	Yes/No(Boolean)	1
Detection_Method	Text	255
Additional_Information	Memo	—
Keywords	Text	50
Published	Yes/No(Boolean)	1
Script	Text	50
Relationships: VULNERABILITY- many to one; on Vulnerability_ID		

Table: VULNERABILITY		
Description: Table containing information about vulnerabilities, primary table in application.		
Field List	**Data Type**	**Size**
Vulnerability_ID	Number (Integer)	2
CounterMeasure_ID	Number (Integer)	2
Name	Text	100
Date	Date/Time	8
Description	Text	255
Detection_Method	Text	255
System_Impact	Text	255
OS_ID	Number (Integer)	2
Platform_ID	Number (Double)	8
Tool_ID	Number (Double)	8
Advisory_ID	Text	50
ReportCounter	Number (Integer)	2
Relationships: HACKER_TOOL- one to many; on Vulnerability_ID, COUNTERMEASURE- one to many; on Vulnerability_ID, Attached_Advisory_Table- one to many; on Vulnerability_ID, Attached_Platform_Table- one to many; on Vulnerability_ID, Attached_OS_Table- one to many; on Vulnerability_ID		

A.6 Mike Neuman's Database

The database maintained by Mike Neuman has the following flat flat-database schema:

Attack Type [Host/Network] (Does this attack require an 'account' on host?)	**Operating System(s)** (List of OS's eg. SunOS4.1.3_U1, IRIX5.3)
Level of Vuln [Root/D.O.S] (Root, denial of service, user eg. daemon,uucp)	**Program/Function Exploited** (eg. tprof, netscape, setreuid(), system())

Race Condition [Y/N] (Race conditions are a different "breed")

Exploit Available [Y/N] (Is an exploit script available?)

Exploit Script Restrictions (Was it given with any restrictions? FYEO, etc)

Error Source Available [Y/N] (Is the broken source code available?)

Source Status [Copyright] (Is it copyrighted by AT&T, Sun, etc?)

Patch Available [Y/N] (Is a patch available to fix the problem?)

Officially Fixed By (What was the official patch number?)

Source of Info [8lgm,etc] (Where this bug was found 8lgm, self, bugtraq...)

Description of Problem Text (A description of exactly what is exploited.)

Impact Description Text (A short description of what can be done with the 'hole'.)

Exploit Script Text (A script/instructions that demonstrates this vulnerability)

Erroneous Source (A listing of the source code, heavily commented to show the error)

Patch to Source Text (A patch(1) style patch to fix the problem, commented to show why the patch fixes the error)

B VULNERABILITY CLASSIFICATIONS - DETAILED LIST

B.1 Aslam Classification

Aslam [Aslam 1995] developed a taxonomy of security faults in the UNIX operating system . His work includes a classification scheme that allows a grouping of vulnerabilities. The classification was later refined by Ivan Krsul at the COAST laboratory [Aslam et al. 1996]. The labeling of the classes in this classification is as shown in figures B.1 and B.2.

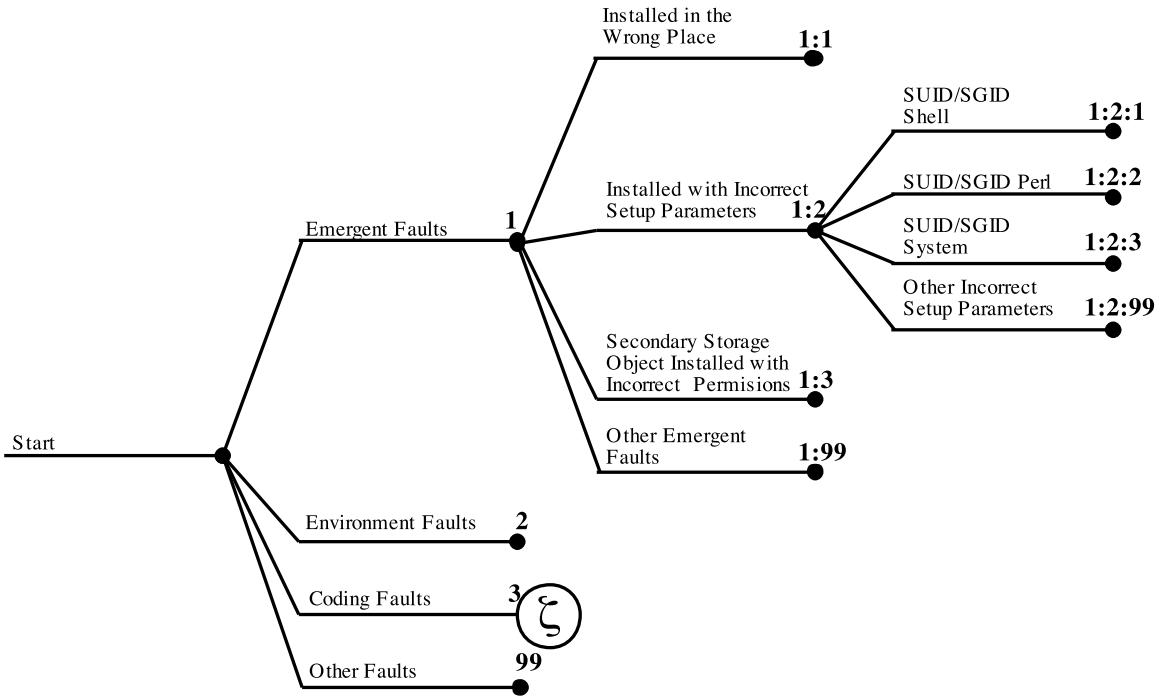


Figure B.1: Aslam Classification decision tree (part 1 of 2) for the `classification` feature.

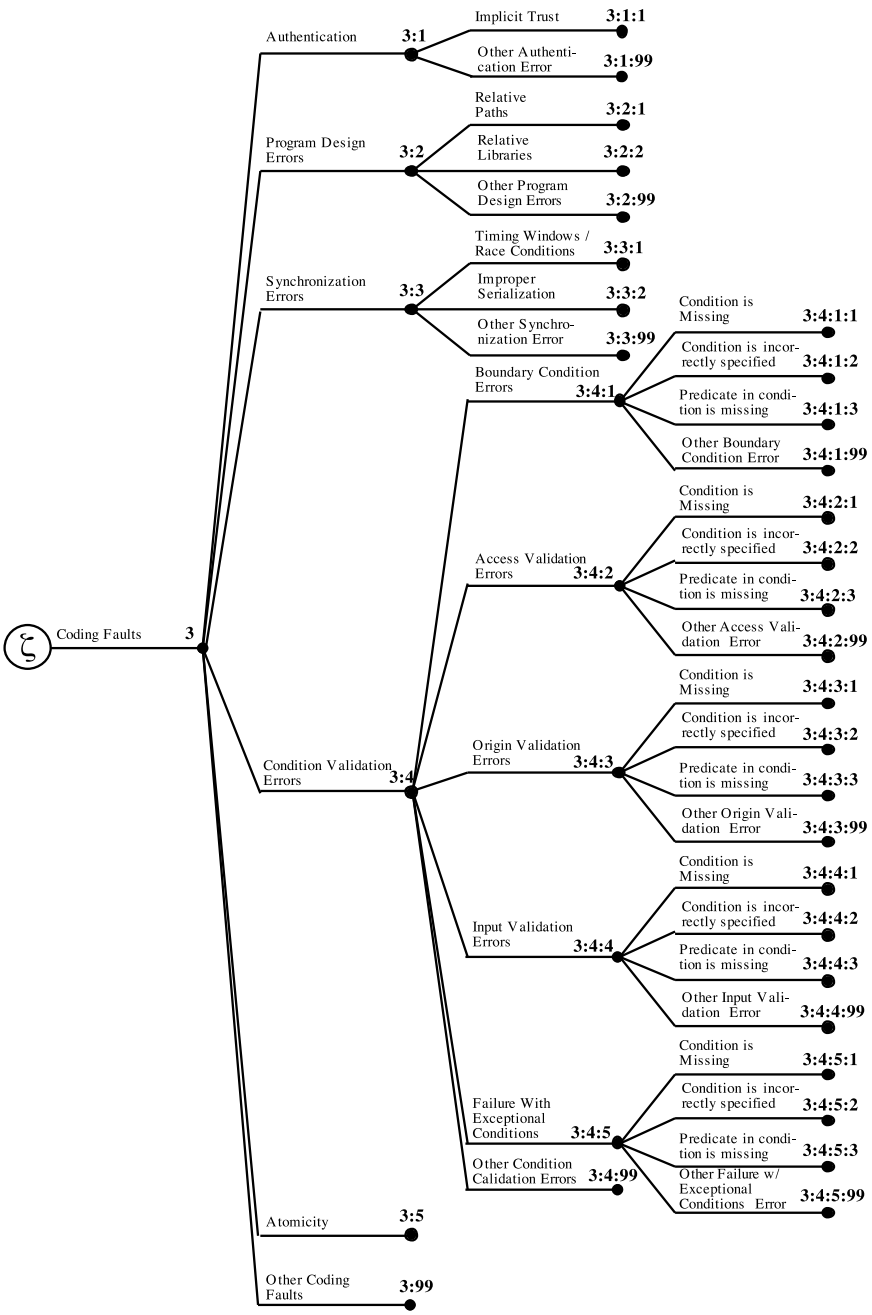


Figure B.2: Aslam Classification decision tree (part 2 of 2) for the `classification` feature.

B.2 Knuth Classification

Donald Knuth, author of the TEX typesetting system kept a detailed log of all the faults fixed in the TEX system for a period of over ten years [Knuth 1989] and developed a detailed classification of types of faults found in his system:

- Algorithm awry.
- Blunder or botch.
- Cleanup for consistency or clarity. *Note: Does not represent a fault.*
- Data structure debacle.
- Efficiency enhancement. *Note: Does not represent a fault.*
- Forgotten function.
- Generalization or growth of ability. *Note: Does not represent a fault.*
- Interactive improvement. *Note: Does not represent a fault.*
- Language liability.
- Mismatch between modules.
- Promotion of portability. *Note: Does not represent a fault.*
- Quest for quality. *Note: Does not represent a fault.*
- Reinforcement of robustness.
- Surprising scenario.
- Trivial typo.

B.3 Grammar-based Classification

DeMillo and Mathur developed a grammar-based fault classification scheme that takes into account that syntax is the carrier of semantics [DeMillo and Mathur 1995]. 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.

- Spurious Entity. A fault whose correction requires the removal of its characteristic substring.
- Missing Entity. A fault whose correction requires the insertion of a syntactic entity into the incorrect program.

- Misplaced Entity. A fault whose correction requires a change in its position within the code.
- Incorrect Entity. When a fault cannot be classified as a missing entity, a spurious entity, or misplaced entity then it is classified as a incorrect entity.

B.4 Endres Classification

This classifier was developed by Endres in [Endres 1975] as an analysis of errors in system programs.

- Group A
  - Machine configuration and architecture
  - Dynamic behavior and communication
  - Functions offered
  - Output listing and format
  - Diagnostics
  - Performance
- Group B
  - Initialization
  - Addressability
  - Reference to names
  - Counting and calculating
  - Masks and comparisons
  - Estimation of range limits
- Group C
  - Placing of an instruction within a module
  - Spelling errors in messages and commentaries
  - Missing commentaries or flowcharts
  - Incompatible status of macros or modules
  - Not classifiable

B.5 Ostrand and Weyuker's Classification

Ostrand and Weyuker's proposed an attribute categorization scheme for the classification of faults [Ostrand and Weyuker 1984].

- Major category
  - Data definition
  - Data handling
  - Decision
  - Decision and processing
  - Documentation
  - System
  - Not an error
- Type
  - Address
  - Control
  - Data
  - Loop
- Presence
  - Omitted
  - Superfluous
  - Incorrect
- Use
  - Initialize
  - Set
  - Update

B.6 Basili and Perricone Classification

This classifier was defined by Basili and Perricone in [Basili and Perricone 1984].

- Initialization
- Control structure

- Interface
- Data
- Computation

B.7 Origin and causes

This classifier was originally defined in [Longstaff 1997] and attempts to identify the origins of the vulnerability.

- Lack of training
- Procedures not followed
- Problem re-introduced
- Bug fix not propagated
- Inconsistent specifications
- Debug code not removed
- From [Eisenstadt 1997]: Faulty assumption/model or misdirected blame.

B.8 Access required

This classifier was originally defined in [Longstaff 1997] and defines the access that is required to exploit the vulnerability.

- Remote using a common service
- Trusted system
- User account
- Physical access
- Privileged access

B.9 Category

This classifier attempts to identify the system component that a vulnerability belong to.

- General system software
- General system utilities

- Logging software
- Software that deals with electronic mail
- Software that deals with networking
- Cryptographic software

B.10 Ease of Exploit

This classifier was originally defined in [Longstaff 1997] and attempts to identify the difficulty of exploiting the vulnerability.

- Simple command
- Toolkit available
- Expertise required
- Must convince a user to take an action
- Must convince an administrator to take an action

B.11 Impact

This classifier attempts to identify the impact of the vulnerability. This classifier 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.

- Access to data
  - Access to administrative or system data
  - Access to user level data
  - Loss of data
    - System data is lost or corrupted by the exploitation of a vulnerability
    - User data is lost or corrupted by the exploitation of a vulnerability
- Execution of commands
  - Execution of administrative or system commands
    - Generalized root access
      - Internal users can obtain generalized root access

        - External users can obtain generalized root access
    - Execution of specific system commands
        - Internal users can execute specific system commands
        - External users can execute specific system commands
- Execution of user level commands
    - Software that is running on behalf of the user can execute a user level command in violation of access controls set by administrators
    - Internal users can execute user level commands in violation of access controls set by administrators
    - External users can execute user level commands in violation of access controls set by administrators
- Execution of code
    - Execution of machine language code with system privileges
        - Internal users can execute machine language code with privileges
        - External users can execute machine language code with privileges
    - Execution of machine language code with user privileges
        - Internal users can execute machine language code
        - External users can execute machine language code
    - Execution of scripts with system privileges
        - Internal users can execute scripts with privileges
        - External users can execute scripts with privileges
    - Execution of scripts with user privileges
        - Internal users can execute scripts
        - External users can execute scripts
- Denial of service
    - System resources are exhausted
    - System resources are eliminated

B.12 Threat

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.

- Threats to availability and usefulness
  - Destroy, damage or contaminate
  - Deny, prolong or delay use of access
- Threats to integrity and authenticity
  - Enter, use or produce false data
  - Modify, replace or reorder
  - Misrepresent
  - Repudiate
  - Misuse or fail to use as required
- Threats to confidentiality and possessions
  - Access
  - Disclose
  - Observe or monitor
  - Copy
  - Steal
- Exposure to threats
  - Endanger by exposure to any of the other threats

B.13 Complexity of Exploit

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.

- Exploitation is a simple sequence of commands or instructions
- Exploitation requires a complex set or large number of commands or instructions.
- The exploitation requires timing and synchronization. Typically requires a script that tries several times and may require slowing the system.

B.14 Cohen's Attacks

This classification is a subset of a large (100) list of attacks possible on a system published in [Cohen 1997a; 1995].

- Errors and omissions. Erroneous entries of missed entries by designers, implementers, maintainers, etc. Forgetting to eliminate default passwords, incorrectly setting protections, etc.
- Trojan horse: A component (HW/SW) that has unadvertised effects
- Invalid value on call: Pass invalid values to system calls to break OS
- Undocumented or unknown function exploitations: Same as vernacular meaning
- Implied trust attack: Programs inappropriately trust other programs
- Imperfect daemon exploits: Attacking a daemon that is not perfect
- Data diddling: Illicitly modify data to trick OS into producing wrong results
- Data aggregation: Combine seemingly innocuous data to get valuable information
- Process bypassing: Bypassing some control process that has inadequate controls
- Input overflow: Attack a program that does not check length of input
- Error-induced misoperation: Errors caused be attack induce incorrect operations
- Audit suppression: Audit trails are prevented from operating properly
- Induced stress failure: Bang on system until they start making mistakes
- Hardware-system failure-flaw: Known hardware or system flaws are exploited
- Network service and protocol: Characteristics of network services are exploited
- Distributed coordinated attacks: Attackers use intermediate systems to attack
- Interprocess communication attacks: Interprocess communication channels are attacked.
- Race conditions: Interdependent sequences of events are interrupted by other events that destroy critical dependencies
- Inappropriate defaults: Default values leave system open to attack

B.15 Cohen's Attack Categories

This classification was developed in [Cohen 1995] and identifies the category of attacks possible given a system that contains a vulnerability.

- Illicit modification of information
- Information gets to places it should not go to
- Failure to provide a service

B.16 Perry and Wallich Attack Classification

This is a matrix-based classification scheme in two dimensions: Potential perpetrators and potential effects. Selected values in the matrix are valid entries [Perry and Wallich 1984].

	Operators	**Programmers**	**Data Entry**	**Internal**	**Outside**	**Intruders**
Physical Destruction	*Bombing Short Circuits*					
Information Destruction	*Erasing Disks*	*Malicious Software*			*Malicious Software*	*Via Modem*
Data Diddling		*Malicious Software*	*False Data Entry*			
Theft of Services		*Theft as User*		*Unauthorized Action*	*Via Modem*	
Browsing	*Theft of Media*			*Unauthorized Action*	*Via Modem*	
Theft of Information				*Unauthorized Action*	*Via Modem*	

B.17 Howard's 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 "link" between attackers and objectives is established through an operational sequence as follows: *Attackers* $\Rightarrow$ *tools* $\Rightarrow$ *access* $\Rightarrow$ *results* $\Rightarrow$ *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. Table B.1 shows the possible values for each level in the tree.

Table B.1: Values allowed for each level of the Howard's Process Based Taxonomy of Network Vulnerabilities

Level	**Choice of Values**
Attackers	Hackers, spies terrorists, corporate raiders, professional criminals, or vandals
Tools	User command, script or program, autonomous agent, toolkit, distribution tool, or data tap.
Access (1 of 4)	Implementation vulnerability, design vulnerability, and configuration vulnerability.
Access (2 of 4)	Unauthorized access, or unauthorized use.
Access (3 of 4)	Processes.
Access (4 of 4)	Files, or data in transit.
Results	Corruption of information, disclosure of information, theft of service, denial of service.
Objectives	Challenge or status, political gain, financial gain, or damage.

B.18 Dodson's Classification Scheme

This classification was developed in [Dodson 1996] for the classification of computer vulnerabilities such that the classes identify generic flaws in software. These can be detected

by using the Tester's Assistant, a tool used to automate software testing [Fink et al. 1994]. This classification is an extension of that proposed in [Aslam 1995].

In this classification each class consists of a 20-tuple of ones and zeros. Each element in the 20-tuple indicates if a question, from the list that follows, was answered with a yes or no answer.

1. Did the error occur when a process attempted to read or write beyond a valid address boundary?
2. Did the error occur when a system resource was exhausted?
3. Did the error result from an overflow of a static-sized data structure?
4. Did the error occur when a subject invoked an operation on an object outside its access domain?
5. Did the error occur as a result of reading or writing to/from a file or device outside a subject's access domain?
6. Did the error result when an object accepted input from an unauthorized subject?
7. Did the error result because the system failed to properly or completely authenticate a subject?
8. Did the error occur because a program failed to parse syntactically correct input?
9. Did the error result when a module accepted extraneous input fields?
10. Did the error result when a module did not handle missing input fields?
11. Did the error result because of a field-value correlation error?
12. Did the error result because the system failed to handle an exceptional condition generated by a functional module, device, or user input?
13. Is the error exploited during a timing window between two operations?
14. Did the error result from inadequate or improper serialization of operations?
15. Did the error result from an interaction in a specific environment between functionally correct modules?
16. Did the error result only when a program is executed on a specific machine, under a particular configuration?
17. Did the error occur because the operational environment is different from what the software was designed for?

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

The classification is only applied to eight vulnerabilities and elements 2, 7, 9, 10, 16, 18, 19, and 20 are not set for any of these vulnerabilities.

C IMPROVEMENTS ON PRIOR CLASSIFICATIONS

As stated in Section 3.1.2, there are four properties that must be satisfied by the taxonomic characters (or features) that are used in classifications: *Objectivity* (the features must be identified from the object known and not from the subject knowing), *Determinism* (there must be a clear procedure that can be followed to extract the feature), *Repeatability* (several people extracting the same feature for the object must agree on the value observed), and *Specificity* (the value for the feature must be unique and unambiguous). As shown in Section 3.2, some of the prior classifications presented were ambiguous, were not repeatable, or not specific.

A solution to these problems is to correct these classifications so they will satisfy as many—if not all—of the desirable properties. For example, classifications that are ambiguous can be fixed by providing instructions that will resolve the ambiguities. Classifications that are not deterministic can be fixed by providing a procedure that must be followed for the determination of the class.

In this section we present modified versions of the classifications listed in Section 3.2 providing decision trees that will both remove ambiguities and provide a deterministic procedure that can be used to determine the value of a class.

The selection of values using these trees does not solve the problem of objectivity because the nodes in the decision trees may have more than one fundamentum divisionis.

We argue that it is desirable to have these *fixed* classifications rather than the old. Any analysis with the old classifications is likely to be biased and contested because different researchers can come to different conclusions based on the values they choose for their classes, and there is no procedure that can resolve such conflicts.

In the development of these decision trees we chose questions that are as objective as possible and where appropriate we have chosen to annotate the decision procedure to clarify doubts and increase the quality of these features. These annotations are indicated by a number inside a circle in the lower right corner of the corresponding decision.

C.1 Indirect Impact

This classification identifies the indirect or ultimate impact of the vulnerability. Indirect impacts are those that ultimately result from the exploitation of the vulnerability. See Figure C.1.

C.2 Direct Impact

This classification identifies the direct impact of the vulnerability. Direct impacts are those that are felt immediately after the vulnerability is exploited. See Figure C.2

C.3 Access Required

This classification was originally defined in [Longstaff 1997] and identifies the access that is required to exploit the vulnerability. See Figure C.3

C.4 Complexity of Exploit

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. See Figure C.4

1. The notion of a *simple sequence of commands* will, of course, vary from person to person. For this classification a *simple sequence of commands* is considered to be a linear sequence of commands (i.e. no loops, gotos, etc.) of no more than a dozen commands. Also, these commands must be commands supported by the operating system, common applications and utilities. Commands that involve scripts and applications that the exploiter must compile, install, etc., do not qualify.
2. Shell scripts, command interpreter source files and macros all qualify. Programs that are implemented in a general purpose programming language (including languages such as Perl) do not qualify.
3. Typically requires a script or application that tries several times and may require slowing down the system.
4. Applications that the exploiter must compile, install, etc.

C.5 Category

This classification identifies the system component that a vulnerability belongs to. See Figure C.5

1. For this classification the operating system is the kernel and all the utilities that are common to all distributions of that operating system, and that are minimally required for its operation.

C.6 OS Type

This classification identifies the class of operating systems that are affected by the vulnerability. See Figure C.6

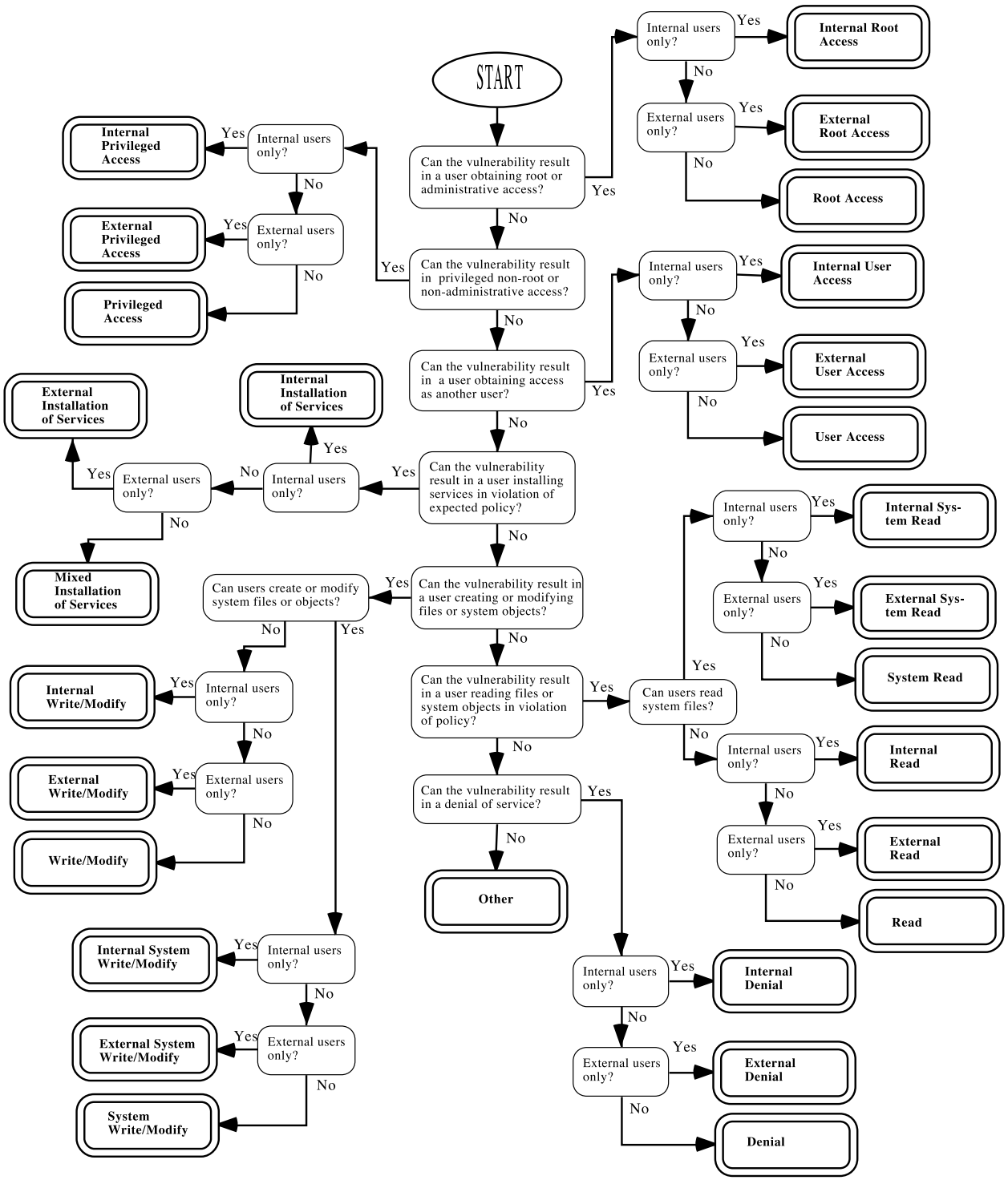


Figure C.1: Selection decision tree for the `indirect_impact` classification.

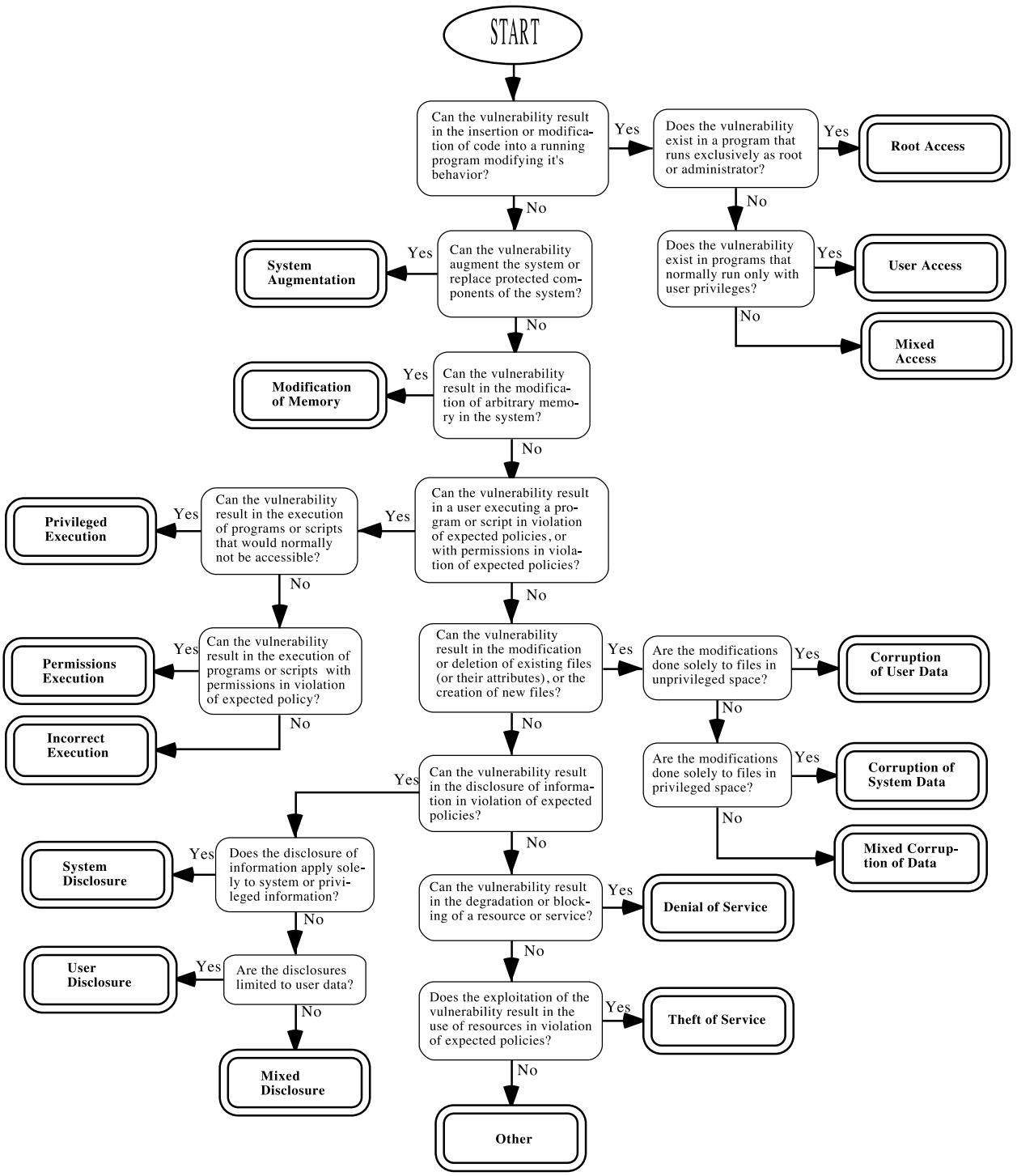


Figure C.2: Selection decision tree for the `direct_impact` classification.

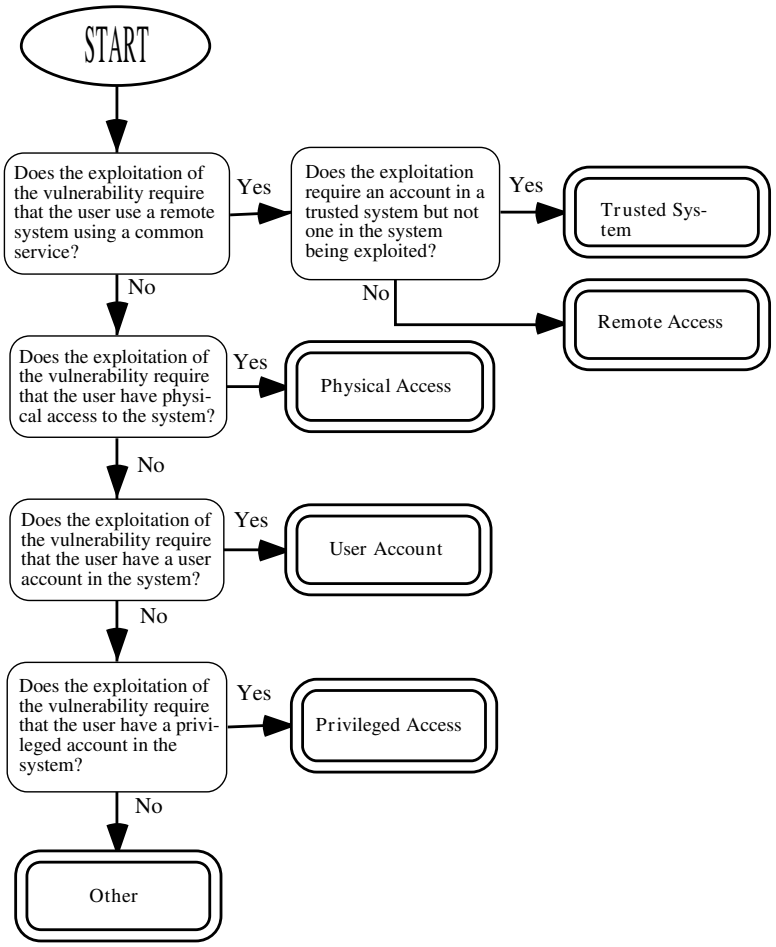


Figure C.3: Selection decision tree for the `access_required` classification.

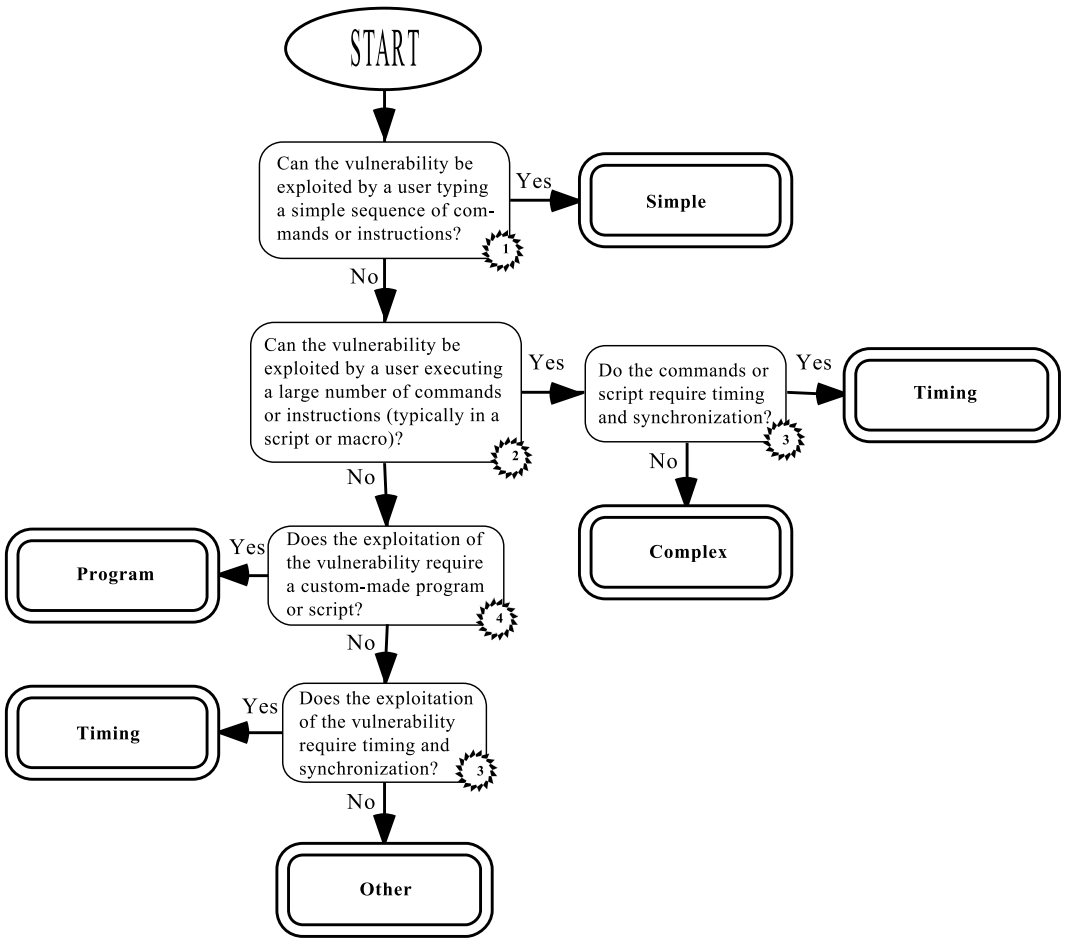


Figure C.4: Selection decision tree for the `complexity_of_exploit` classification.

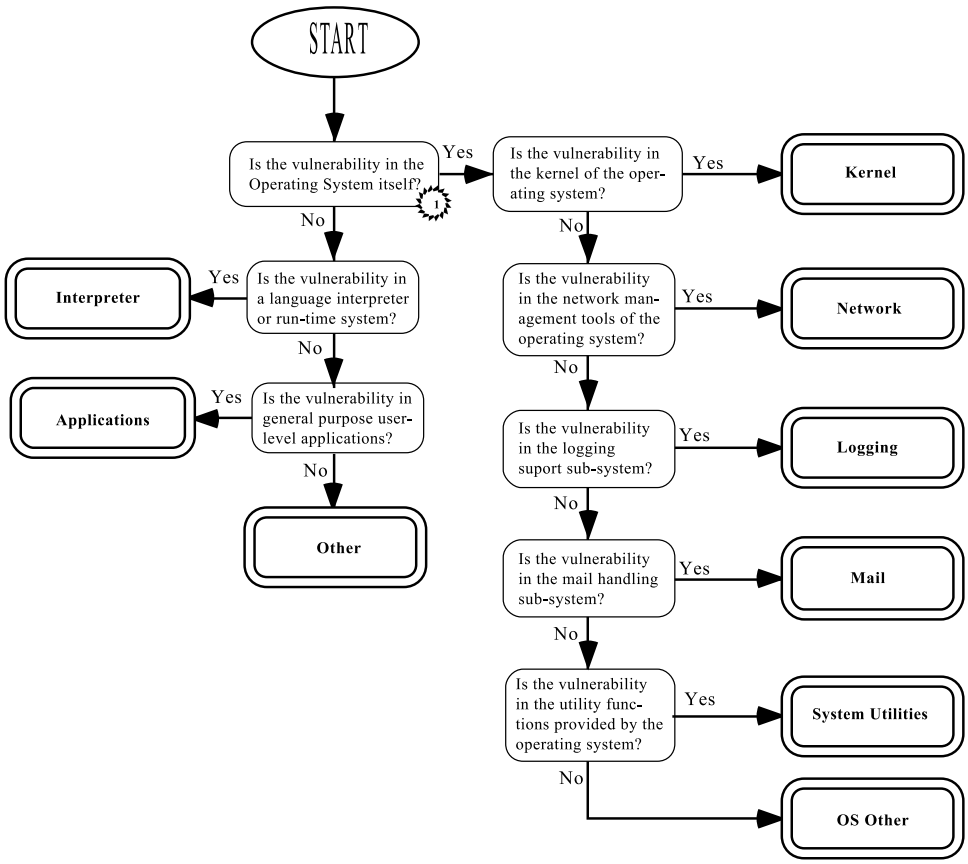


Figure C.5: Selection decision tree for the `category` classification.

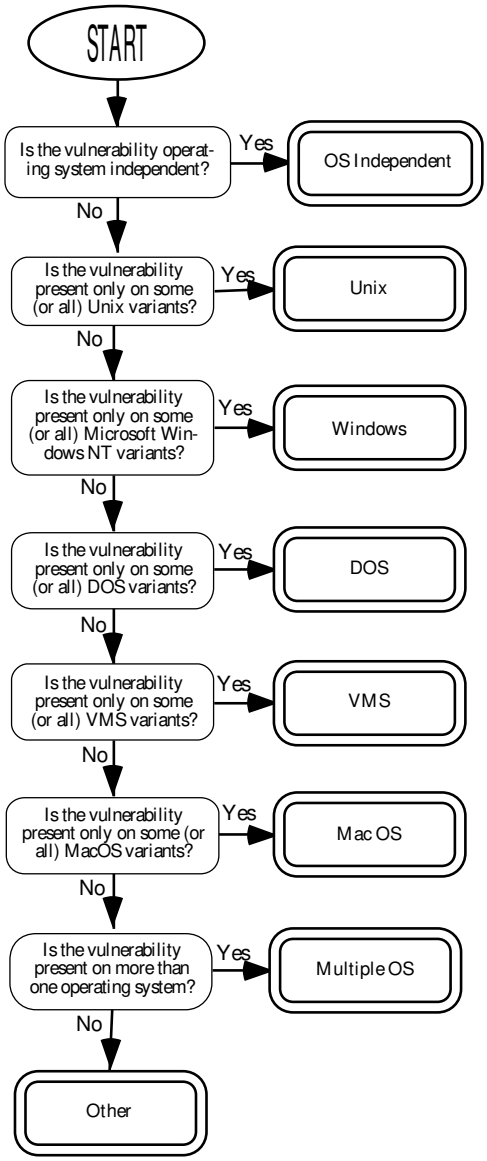


Figure C.6: Selection decision tree for the `os_type` classification.

VITA

VITA

Ivan Krsul was born on the 5th of October 1966 in La Paz, Bolivia. He received his masters degree in computer science from Purdue University in May 1994. He received his Bachelor of Science in computer engineering from The Catholic University of America, Washington D.C., in May 1989, and his high school degree from Colegio Saint Andrews in La Paz, Bolivia, in December 1984.