A Framework for Deriving Verification and Validation Strategies to Assess Software Security

By

Anil Bazaz

Dissertation submitted to the faculty of
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science

Approved:

James D. Arthur
(Chairman)

Richard E. Nance

Srinidhi Varadarajan

Joseph G. Tront

Randolph C. Marchany

April 14th, 2006
Blacksburg, VA

Keywords: Vulnerabilities, Verification, Validation, Software Security, Constraints, Assumptions

A Framework for Deriving Verification and Validation Strategies to Assess Software Security

By

Anil Bazaz

Abstract

In recent years, the number of exploits targeting software applications has increased dramatically. These exploits have caused substantial economic damages. Ensuring that software applications are not vulnerable to the exploits has, therefore, become a critical requirement. The last line of defense is to test before hand if a software application is vulnerable to exploits. One can accomplish this by testing for the *presence* of vulnerabilities.

This dissertation presents a framework for deriving verification and validation (V&V) strategies to assess the security of a software application by testing it for the presence of vulnerabilities. This framework can be used to assess the security of any software application that executes above the level of the operating system. It affords a novel approach, which consists of testing if the software application permits violation of constraints imposed by computer system resources or assumptions made about the usage of these resources. A vulnerability exists if a constraint or an assumption can be violated. Distinctively different from other approaches found in the literature, this approach simplifies the process of assessing the security of a software application.

The framework is composed of three components: (1) a taxonomy of vulnerabilities, which is an informative classification of vulnerabilities, where vulnerabilities are expressed in the form of violable constraints and assumptions; (2) an object model, which is a collection of potentially vulnerable process objects that can be present in a software application; and (3) a V&V strategies component, which combines information from the taxonomy and the object model; and provides approaches for testing software applications for the presence of vulnerabilities. This dissertation also presents a step-by-step process for using the framework to assess software security.

Problems worthy of attack prove their worth by fighting back.

Paul Erdos

We all agree that your theory is crazy, but is it crazy enough?

Niels Bohr

...one of the tests of a theory is that, once grasped, it appears self-evident.

Arthur Koestler

To:

My parents, without whom I could not have accomplished this

My brother Sandeep, who has always been there for me

My soon-to-be wife Shivani, who inspired me to complete this work

Acknowledgements

I would like to take this opportunity to thank the people who helped me bring this work to completion:

- My advisor Dr. James D. Arthur, who has been an advisor, a collaborator and a friend, and has guided me through the dissertation process.
- All my Ph.D. Committee – James D. Arthur, Richard E. Nance, Srinidhi Varadarajan, Joseph G. Tront and Randolph C. Marchany – for their valuable suggestions, support and encouragement.
- A special thanks to Randolph Marchany for spending countless hours talking to me about this work.
- The staff at the Department of Computer Science.
- Sara Thorne-Thomsen for her help with reviews of this dissertation.
- My friends who put up with me for all this time: Jon Bernard, Karthik Channakeshava, Bharath Ramesh, Joy Mukherjee, Vikas Agrawal, Rajneesh Mahajan, Arvind Sharma, Omprakash Seresta and Scott Preddy.
- My family for their support and caring: (Late) Raj Rani Bazaz, Arjun Nath Bazaz, Janki Nath Wali, Krishan and Santosh Bazaz, Sharawan and Anupama Bazaz, Veerji and Sweety Bazaz, Dilip and Sunita Bazaz, Surinder and Girja Kak, H. L. and Veena Kalu, Vishal and Manju Kak.
- Lastly, I would like to thank Dr. Richard E. Nance and the faculty and staff at the Systems Research Center (SRC) for supporting me while I was doing this work. Working with them has provided me with professional and life experience, caring and support, and I have learned much from them.

Contents

Abstract ... ii
Acknowledgements ... v
List of Figures ... x
List of Tables ... xii
1. Introduction ... 1
1.1 Problem Synthesis ... 3
1.2 Why Assessing Security Is Difficult ... 5
1.3 Solution Approach ... 6
1.4 Organization of the Dissertation ... 8
2. Framework Overview ... 9
2.1 Software Process ... 11
2.2 Related Work ... 12
2.2.1 Flaw Hypothesis Methodology ... 14
2.2.2 Hollingworth's Automated Tools ... 16
2.2.3 Gupta and Giligor's Formal Penetration Analysis ... 17
2.2.4 Standards and Criteria ... 19
2.2.5 System Security Engineering-Capability Maturity Model (SSE-CMM) ... 20
Summary ... 21
3. Taxonomy of Software Vulnerabilities ... 22
3.1 Theoretical Foundation ... 23
3.2 The Taxonomy ... 26

3.2.1 Main Memory ... 29
3.2.1.1 Dynamic Memory ... 30
3.2.1.2 Static Memory ... 33
3.2.2 Input/Output (I/O) ... 35
3.2.2.1 Filesystem ... 35
3.2.2.2 Network Interface ... 38
3.2.3 Cryptographic Resources ... 41
3.2.3.1 Randomness Resources ... 42
3.2.3.2 Cryptographic Algorithms and Protocols ... 45
3.2.4 Scope ... 49
3.3 Extending the Taxonomy ... 50
3.4 Verifying the Taxonomy ... 54
3.5 Characteristics of a Taxonomy ... 57
3.6 Related Work ... 60
3.6.1 Protection Analysis Project ... 61
3.6.2 Research in Secure Operating Systems (RISOS) Project ... 62
3.6.3 Newman and Parker’s Taxonomy of Misuse Techniques ... 63
3.6.4 VERDICT ... 64
3.6.5 Aslam’s Taxonomy ... 65
3.6.6 Krsul’s Extension of Aslam’s Taxonomy ... 66
Summary ... 67
4. Object Model ... 68
4.1 Dynamic Memory ... 70
4.2 Static Memory ... 71
4.3 Filesystem ... 72
4.4 Network Interface ... 72

4.5 Randomness Resources ... 73
4.6 Cryptographic Algorithms and Protocols ... 74
Summary ... 75
5. Verification and Validation Strategies ... 76
5.1 Dynamic Memory ... 79
5.2 Static Memory ... 82
5.3 Filesystem ... 84
5.4 Network Interface ... 87
5.5 Randomness Resources ... 89
5.6 Cryptographic Algorithms and Protocols ... 91
Summary ... 94
6. Security Assessment Process ... 95
6.1 Security Assessment Process ... 95
6.2 Example ... 97
Summary ... 102
7. Conclusions and Future Directions ... 103
7.1 Summary of Main Contributions ... 104
7.2 Future Work ... 106
7.2.1 Refining the Process of Using the Framework ... 106
7.2.2 Identification and Development of Tools ... 106
7.2.3 Adding New Vulnerabilities to the Taxonomy ... 106
7.2.4 Identifying Critical Constraints and Assumptions ... 107
7.2.5 Deriving Requirements for Developing Secure Software ... 107
Summary ... 107
Appendix A ... 109
Appendix B ... 110

References... 112
Vita... 121

List of Figures

Figure 2.1 Framework for deriving V&V strategies 10
Figure 2.2 Hierarchal view of computer system 12
Figure 3.1 High level view of software process executing on a computer system 24
Figure 3.2 First Level Categories of the taxonomy 26
Figure 3.3 Subcategories of main memory category. 29
Figure 3.4 Constraints/Assumptions associated with dynamic memory 32
Figure 3.5 Constraints/Assumptions associated with static memory............................... 34
Figure 3.6 Subcategories of I/O category. 35
Figure 3.7 Constraints/Assumptions associated with filesystem..................................... 37
Figure 3.8 Constraints/Assumptions associated with network interface 40
Figure 3.9 Subcategories of cryptographic resources category. 42
Figure 3.10 Constraints/Assumptions associated with randomness resources 45
Figure 3.11 Constraints/Assumptions associated with cryptographic algorithms & protocols.. 48
Figure 3.12 Distribution of advisories classified and not classified by the taxonomy 55
Figure 3.13 Distribution of classified vulnerabilities according to first level categories of the taxonomy.. 55
Figure 3.14 Distribution of classified vulnerabilities according to second level categories of the taxonomy 56
Figure 3.15 Protection analysis classification scheme... 61
Figure 3.16 RIOS taxonomy of system integrity flaws 62
Figure 3.17 Newman and Parker's taxonomy of computer misuse techniques................ 63
Figure 3.18 Aslam's taxonomy of security faults.. 65

Figure 3.19 Structure of Krul’s taxonomy 66
Figure 4.1 The object model 69
Figure 4.2 Objects associated with dynamic memory 70
Figure 4.3 Objects associated with static memory.. 71
Figure 4.4 Objects associated with filesystem 72
Figure 4.5 Objects associated with Network Interface. 72
Figure 4.6 Objects associated with randomness resources. 73
Figure 4.7 Objects associated with cryptographic algorithms and protocols 74
Figure 5.1 The V&V strategies component. 78

List of Tables

Table 5.1 Base strategies associated with dynamic memory. 79
Table 5.2 Base strategies associated with static memory. 83
Table 5.3 Base strategies associated with filesystem... 84
Table 5.4 Base strategies associated with network interface. 87
Table 5.5 Base strategies associated with randomness resources. 89
Table 5.6 Base strategies associated with cryptographic algorithms and protocols. 91
Table 6.1 Test strategies for password program example.. 100

Chapter 1

Introduction

The last decade has seen a dramatic growth in the use of the computer systems. In addition to being used for a wide variety of applications ranging from personal use to managing critical infrastructure, computers today are also more accessible. The advent of the Internet has dramatically increased interconnectivity among computers, making it possible to remotely access most of the computers connected to the Internet.

This widespread use and dependence makes computers an enticing target for attacks, while the increased accessibility makes them an easy target. Disruption/destruction of services they provide can lead to damage to both life and property or, at the very least, cause substantial annoyance to the users. The numerous attacks targeting computer systems highlight the fact that computers are both easy and enticing targets [BUGT, SEFO].

These attacks have caused damages worth millions of dollars (e.g., the denial of service attack on e-commerce websites in February, 2000) and much suffering. In 2001, an Australian hacked into the Maroochy Shire, Queensland waste management system and spilled millions of liters of sewage into parks and rivers. The computerized sewage system was not connected to the Internet and was accessible only through a modem. When the authorities caught the man responsible, it turned out that he worked for the company that had installed the computerized system [SIMT01]. This attack underlines the importance of security even for computer systems not accessible on the Internet.

Almost all of these attacks, which occur daily on numerous computer systems, target and exploit software applications running on them. Exploits against software applications are particularly damaging, because most of these can be automated relatively easily. That is, once a software application has been successfully exploited, a program can be written that automates the exploit process. This program is then used to attack other installations of the software application. The implications of such capabilities are that once an exploit is automated, little skill is required of other perpetrators who desire to effect similar software exploits. More specifically, anyone who has basic operational knowledge of a computer system can execute these automated programs and exploit the software application.

The exploits themselves have also evolved in complexity and sophistication. Some require multiple steps and involve a number of computer system components. Consider the example of the "broken" version of the password program in SunOS and HP/UX systems, a historic case of the "time of check time of use" flaw [BISH96a]. The complete exploit of the password program is complicated and is carried out by identifying and replacing a number of files being used by the program while it is in execution. The exploit involves multiple steps, which much be carried out in precise order and with precise timing.

The importance of the computer systems and the ease and frequency of attacks create an urgent need to ensure the security of software applications. An obvious solution is to develop software applications that are not vulnerable to exploits. However, because software developers cannot develop perfect applications, the last line of defense is to assess, after-the-fact, if a software application is susceptible to security exploits. Current approaches for assessing the security of software applications are variations of Flaw Hypothesis Methodology (FHM). FHM, first introduced by Weissman and Linde [WEIS73, WEIS95, LIND75], is a form of penetration testing, where a system is analyzed and probed for weaknesses. Although FHM provides a structured process for assessing security, it still relies heavily on the ability and past experience of the person evaluating the software application.

FHM was introduced in mid 1970's. Since then, however, research that takes holistic view towards assessing software security has been minimal. Current approaches for assessing security are being constrained by several limiting factors, e.g., application complexity, execution environment and the variations in hardware and operating systems. These factors necessitate a fresh look at the problem and development of new approaches for assessing if a software application is vulnerable to exploits.

1.1 Problem Synthesis

Assessing the extent to which a software application is vulnerable to exploits consists of either evaluating the process used to develop the application or testing if the application itself is secure. These two approaches complement each other. The research described herein takes the latter approach, that is, assessing the security of a software application by testing the application itself. This approach is more effective than evaluating the process, because an application developed using a process that includes security best practices is not necessarily immune to security exploits. However, evaluating an application for the presence of vulnerabilities can guarantee its immunity from the exploits that employ those vulnerabilities.

It is possible to test a software application for its susceptibility to exploits in two different ways:

1. *Threat based approaches:* These approaches consider threats against a software application. Threats are risks of security exploits. For example, threats exist against

software applications using the network interface, because someone can intercept data being sent and received through the network interface. Threat based approaches, therefore, use an external perspective, because threats are external to the software application. Assessing the security of an application consists of identifying threats and testing if they can be realized.

2. *Vulnerability based approaches:* These approaches consider vulnerabilities present in the software application. A vulnerability is defined as a state of the system. What differentiates a vulnerable state from any other state is the fact that it is possible to move to an incorrect system state from it [BISH96b]. In other words, a vulnerability is a defect which, when exploited, can produce undesirable or incorrect behavior [WHIT03]. Vulnerability based approaches, therefore, use an internal perspective to assessing security because vulnerabilities are internal to the software application. Software applications are attacked by exploiting vulnerabilities present in them. Therefore, assessing security can be achieved by identifying vulnerabilities present in the software application.

Before continuing, it is important to note the difference between a vulnerability and an exploit. An exploit consists of a vulnerability present in the software application *and* a method used to take advantage of that vulnerability. Thus, an exploit occurs when a method is applied to exercise the vulnerability. For example, the buffer overflow exploit consists of a vulnerability, which is an unbounded buffer; the method used to exercise that vulnerability is to store data larger than the size of the buffer in the unbounded buffer.

We take the vulnerability approach towards assessing security, because vulnerabilities are fundamental to exploiting a software application. That is, vulnerability based approaches address internal defects, and are therefore more effective than the threat based approaches that address external risks. Nonetheless, threat based approaches, in the form of a risk management component that prioritizes threats, can still be used to manage risks against a software application.

They can be used in combination with vulnerability based approaches to provide a more comprehensive solution for assessing the security of the software application.

1.2 Why Assessing Security Is Difficult

There are a number of issues that contribute to the difficulty of evaluating the security of a software application. A discussion of these issues follows:

1. **Software Complexity:** Modern software applications are large and complicated. It is common for an application to have hundreds of thousands of lines of code. Additionally, these applications use services provided by other applications, which results in a manifold increase in the overall complexity of the application. Identifying any single vulnerability in such complex applications is a formidable task. Moreover, the complexity makes it difficult to develop a systematic approach for assessing security of such applications.
2. **Vulnerabilities, numbers and complexity:** Because there are numerous known vulnerabilities, and a number of them can be present in a software application, it is impractical to list them all and test for their presence in a software application. Furthermore, some vulnerabilities are so complicated in nature that testing for their presence or absence becomes a monumental task.
3. **Dynamic nature of security:** Vulnerabilities and exploits are dynamic in nature with new variations of older instances being continuously discovered. Moreover, completely new vulnerabilities and exploits are constantly being discovered.

   Computer science is itself, a dynamic field with new development paradigms and new languages being developed constantly. These developments bring with them new vulnerabilities. Therefore, any approach for assessing security has to evolve with the dynamic nature of computer science and software security.

4. **Scope of security assessment:** Because no single security assessment approach can address all existing vulnerabilities, it is important to bound the set of vulnerabilities that a security assessment approach addresses. This is difficult to do however, because there is no clear and established definition of a vulnerability. Consequently, this ambiguity often leads to any error that has the slightest security implications as being labeled a vulnerability. This ambiguity also limits the ability to bound the set of vulnerabilities that a security assessment covers or should cover.

   Moreover, it is important to identify the class of software applications to which a particular security assessment approach can be applied. The wide variety of existing software applications and their diverse set of characteristics make this task extremely difficult.

1.3 Solution Approach

As a solution to the problem of assessing software security, this dissertation presents a framework for testing a software application for the presence of vulnerabilities. This framework provides a systematic approach for assessing software security. It is composed of three components:

- ***Taxonomy of Vulnerabilities*:** The taxonomy provides an informative categorization of existing vulnerabilities, which are in the form of violable constraints and assumptions. It is based on a theoretical model of computing that establishes the relationship between software applications and computer system resources, i.e., main memory, Input/Output (I/O), and cryptographic resources. This relationship captures the fact that software applications are exploited by violating constraints imposed by computer system resources or by violating assumptions made about the usage of these resources. These constraints and assumptions underlie the rationale as to why a software application is vulnerable to exploits. In other words, a vulnerability exists in the software application if it allows violation of a constraint or an assumption. The taxonomy classifies these constraints and assumptions.

The characterization of vulnerabilities as constraints and assumptions simplifies the process of identifying vulnerabilities present in a software application. That is, one can test if the software process permits the violation of a constraint or an assumption; a vulnerability exists if either can be violated.

Because constraints and assumptions are bound to resources, the taxonomy uses them as organizational categories. Hence, the three top level categories of the taxonomy are: (1) memory, (2) I/O, and (3) cryptographic resources. Each of these categories is divided into subcategories into which the constraints and assumptions are classified.

- ***Object Model*:** The object model is an organized collection of potentially vulnerable process objects. A process object is defined as an entity that holds data and has associated with it a set of operations that can be performed on it. Examples of process objects include files, directories, and buffers. The object model enumerates potentially vulnerable process objects present in a software application. The objects currently listed in the object model have been identified by analyzing the constraints and assumptions listed in the taxonomy. Furthermore, the object model uses a classification scheme similar to that found in the taxonomy to organize objects. This facilitates the association of process objects with constraints and assumptions.
- ***Verification and Validation (V&V) Strategies Component*:** The V&V strategies component provides base strategies for testing a software application for the presence of vulnerabilities. Each base strategy is composed of three elements: (1) potentially vulnerable process objects (identified in the object model), which serve as targets for the base strategy; (2) a constraint or an assumption from the taxonomy that serves as the goal of the base strategy – that is, the goal is to test if the software application

allows violation of a constraint or an assumption; and (3) a method for testing if the software process allows violation of the constraint or the assumption. The base strategies are used as blue prints to derive the detailed tests. As is the case with the object model, the V&V strategies component uses the classification scheme of the taxonomy to organize test strategies. In effect, this makes it possible to associate process objects and constraints and assumptions with base strategies.

The framework also details a process for assessing the security of a software application. The process provides a step-by-step procedure to test a software application for presence of vulnerabilities. The first step employs the object model for identifying potentially vulnerable objects present in the software application. The second step uses the taxonomy to identify the constraints and assumptions associated with those objects. Finally, the V&V base strategies are used to derive methods that test if the constraints and assumptions can, in fact, be violated.

1.4 Organization of the Dissertation

The remainder of this dissertation is organized as follows: Chapter 2 provides an overview of the framework and highlights existing approaches for assessing software security. Chapter 3 presents the taxonomy of vulnerabilities along with notable existing security taxonomies. Chapters 4 and 5 present the object model and the V&V strategies component, respectively. Chapter 6 presents the process of using the framework. Finally, Chapter 7 provides a conclusion and outlines future work.

Chapter 2

Framework Overview

In this chapter, we present an overview of the framework for assessing security of a software application. The framework presents a conceptual abstraction based on the premise that software applications are exploited by violating constraints imposed by computer system resources or by violating assumptions made about the usage of those resources. The framework uses this premise to develop a systematic approach to assess security of a software application by testing it for the presence of vulnerabilities. The framework is composed of three components:

- *The taxonomy of vulnerabilities:* The taxonomy is an informative classification of vulnerabilities, which are in the form of:
    - constraints imposed by computer system resources, and
    - assumptions made about the usage of these resources.

  These constraints and assumptions underlie the rational as to why a software application is vulnerable to exploits. In other words, a vulnerability exists in the software application if it allows the violation of a constraint or an assumption. Chapter 3 details the taxonomy of vulnerabilities.
- *The object model:* The object model is an organized collection of the process objects, such as files, directories, and buffers. It assists in the identification of potentially vulnerable process

objects that are present in a software application. These process objects are derived by analyzing the constraints and assumptions listed in the taxonomy. Chapter 4 presents the object model.

- *The Verification and Validation (V&V) strategies component:* The V&V strategies component provides base strategies to test the software application for the presence of vulnerabilities. These strategies include methods to test if a software application permits the violation of constraints and assumptions associated with the potentially vulnerable process objects present in a software application. The base strategies are used as blueprints to derive detailed test strategies. Chapter 5 presents the verification and validation strategies component.

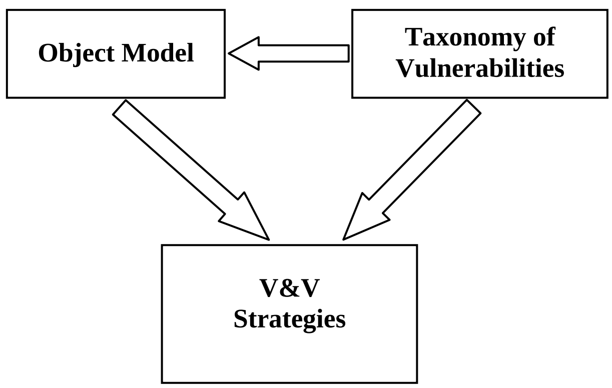


Figure 2.1 Framework for deriving V&V strategies

As illustrated in Figure 2.1, the three components of the framework are related to each other. The process objects in the object model are derived by analyzing constraints and assumptions listed in the taxonomy. These process objects also serve as targets for the V&V base strategies. Additionally, the V&V component uses constraints and assumptions defined in the taxonomy as goals of the base strategies.

The three components of the framework share a similar decomposition. In particular, the process objects in the object model, the constraints and assumptions in the taxonomy, and the base strategies in V&V strategies component are organized in a similar manner. Similar structures make it possible to draw definitive associations among process objects, constraints and assumptions, and base strategies, which, in turn, enables the definition of detailed tests for determining if a software application contains vulnerabilities.

This framework offers a number of advantages for assessing software security. In particular, it:

1. Offers a comprehensive solution for assessing security. As demonstrated in Section 2.1, the framework supports the security assessment of any software application that executes above the level of the operating system.
2. Provides a systematic approach for assessing software security.
3. Includes a taxonomy of vulnerabilities. The taxonomy is an informative classification of known vulnerabilities, characterized as violable constraints and assumptions.
4. Provides base test strategies from which evaluators derive detailed tests to determine if vulnerabilities are present in a software application.
5. Uses a novel approach for testing the security of a software application. That approach consists of evaluating the software application to determine if it permits the violation of a constraint or an assumption.

In addition to these advantages, the framework offers a number of other advantages that are discussed later in this dissertation.

The remainder of this chapter presents the definition of a "software process" and discusses existing approaches for assessing software security.

2.1 Software Process

For clarification purposes, and relative to this research, the definition of a "software application" is a necessary beginning point. This is accomplished by describing it in terms of its relationship with the other components of the computer system. Modern computer systems can be visualized as a hierarchy of components. Figure 2.2 presents a rendering of this hierarchy as a layered inverted pyramid. The pyramid is inverted to emphasize that the number and frequency of attacks increase as one moves from the bottom layer to the top layer. The hardware, or the physical devices that make up a computer system, is located at the bottom of the pyramid. This layer includes memory, hard disk, processor, buses, and so forth. The operating system, which manages the hardware devices and provides an interface to them, occupies the layer above the hardware. Software applications, which occupy the layer above the operating system, provide specific services to the computer system users and to other software applications. Examples of software applications include document editing, e-mail, and scientific applications.

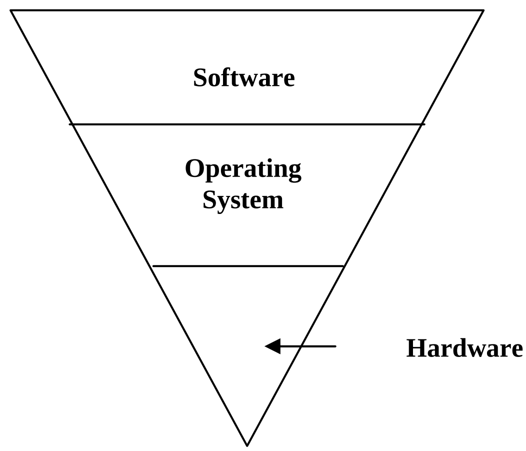


Figure 2.2 Hierarchal view of computer system

The three layers of the pyramid are tightly coupled; a typical layer utilizes the functionality provided by the lower layer, and provides services to the layer above it. Software applications are at the top of this pyramid and provide services to the end users and to other software applications. When executing, these applications can be viewed as software processes. The framework focuses on software processes that operate above the level of the operating system. Although processes utilize hardware resources, they do so through the interface provided by the operating system.

2.2 Related Work

A significant amount of research has been done on individual security vulnerabilities. Tools have been developed to safeguard against specific vulnerabilities. For example, StackGuard is a tool for avoiding buffer overflows [COWA98]. Tools have also been developed to test for the presence of groups of known vulnerabilities. For example, COPS [COPS] is a package tool that tests a UNIX system for the presence of known vulnerabilities such as improper file and directory permissions, setuid processes and poor passwords. Another similar security tool is ITS4 [ITS4], which scans C and C++ source code for known implementation level vulnerabilities such as buffer overflows. Other examples of similar tools include ISS [ISSC], Swatch [SWAT], and Tripwire [TRIP].

Even though these tools are helpful in assessing the security of a software application, they suffer from several shortcomings. The primary ones are:

1. Because these tools are specific to an operating system or a programming language or a set of software applications, they have limited applicability.
2. Most of these tools identify vulnerabilities by searching a software application for signatures of known vulnerabilities. However, because the implementation details of each vulnerability vary from one software application to another, and because pattern matching only allows searching for specific patterns, these tools can only test for the presence of a limited set of simple vulnerabilities, e.g., improper access permissions and unbounded buffers. This further reduces the applicability of these tools.

Nevertheless, despite their limited applicability, these tools are still useful, because they automate at least some aspects of assessing the security of a software system.

In addition to tools, a large amount of literature in the form of books, websites, mailing lists, magazine articles, and so forth is available on individual security flaws and how to avoid them. Nonetheless, research that takes comprehensive view of the problem has been sparse. Only a few approaches exist that take a holistic view of the problem of assessing the security of a software application. The remainder of this section outlines these approaches and discusses their advantages and disadvantages.

2.2.1 Flaw Hypothesis Methodology

Clark Weissman [WEIS73, WEIS95] at Systems Development Corporation developed the Flaw Hypothesis Methodology (FHM), one of the first approaches for testing an operating system for presence of security flaws. FHM consists of a method for testing an operating system using penetration testing, a form of stress testing, where a system is analyzed and probed to expose its weaknesses. Weissmans's FHM consists of four steps:

1. *Flaw generation:* This step begins with the study and analysis of the target system. After the study phase, the evaluators conduct a detailed system review, in which they hypothesize flaws. The evaluators probe critical areas of system design for security flaws; these design considerations become "plausible move generators." Weissman categorizes the top ten generators of flaws as: (1) past experience with flaws in other similar systems, (2) ambiguous, unclear architecture and design, (3) circumvention/bypass of "omniscient" security controls, (4) incomplete design of interfaces and implicit sharing, (5) deviations from the protection policy and model, (6) deviations from initial conditions and assumptions, (7) system anomalies and special precautions, (8) operational practices, prohibitions, and spoofs, (9) development environment, practices, and prohibitions, and (10) implementation errors.
2. *Flaw confirmation:* This step involves confirming the hypothesized flaws. Examining the object system documentation and code confirms the majority of the hypothesized flaws. Weissman lists three sub-steps within flaw confirmation: (1) flaw prioritization and assignment, which involves prioritizing the flaws and assigning them to different team members, (2) desk checking, which involves confirming the flaws by examining the object system's documentation, and (3) live testing, which focuses on designing test cases to confirm the flaws. Weissman notes that live testing is expensive and should be performed only as a last resort. Also the test cases should be designed to confirm the flaw and not to exploit it.

3. *Flaw generalization:* In this step, evaluators analyze each confirmed flaw to determine if the flaw is a member of a generic class of flaws. This generalization assists evaluators in finding similar flaws in other parts of the system and in other systems. Additionally, the generalization helps in gaining new insight on countermeasures that can be employed against the flaw.
4. *Flaw elimination:* The final step in Weissmans FHM is flaw elimination. This step is not present in Weissmans original report [WEIS73], but is included in a summary article [WEIS95] that he published in 1995. This step involves repairing the confirmed flaws. These repairs can range from simple implementation fixes to design countermeasures. Once repaired, the system is re-evaluated to confirm that the flaws have been fixed.

FHM is a generic methodology that can be applied to any software application, and has become the most widely used method for testing a software application for the presence of security flaws.

Linde, who worked with Weissman at Systems Development Corporation, also published a paper describing FHM [LIND75]. Linde offers a slightly different version of FHM in that he adds a new step at the beginning and leaves out the last step (flaw elimination). This new first step in Linde's FHM is:

1. *Knowledge of control structure*. In this step, system evaluators gain familiarity with the operating system by studying its manuals, analyzing the filesystem, analyzing I/O, and so forth.

Although Weissman does not explicitly include this step in his FHM, he has a background stage, which is analogous to this step, and which he divides into four steps: (1) establishing testing goals, which consists of establishing explicit goals for the FHM, (2) defining the object system to be tested, which means identifying whether the system is a stand alone or a networked system, and isolating it from the other production systems, (3) posturing as a penetrator, which consists of figuring out whether the system is to be tested as a white box or as a black box, and (4) fixing penetration analysis resources, which establishes the upper bounds on the penetration testing based on the available resources.

Although FHM has become a leading method to testing for the presence of security flaws, it does suffer from some drawbacks. First, FHM does not describe a methodology or a theory for hypothesizing security flaws, nor does it include an inventory or a taxonomy of security flaws that can be used to identify potential flaws in a software application. Therefore, the evaluators are responsible for analyzing the system and identifying potential security flaws. Secondly, since FHM does not describe a method to actually test a software application for presence of security flaws, the evaluators must determine their own ways of confirming a flaw. This is accomplished either by studying the system documentation or by defining live tests that expose the flaws. Consequently, as both drawbacks indicate, FHM relies heavily on the evaluator’s ability, past experience and familiarity with similar systems. Although FHM by itself does provide some structure to the process of finding security flaws, it is the evaluators who identify potential security flaws and define ways of confirming their presence.

2.2.2 Hollingworth’s Automated Tools

Hollingworth *et al.* [HOLL74] attempted to change the status quo in penetration testing by advocating the use of automated tools for identifying security flaws. Based on their past experience, they described the desirable functional capabilities for these tools:

1. *Controlled program execution:* The tools should allow the evaluators to introduce run time break points and redirect the control flow. Although such capabilities are found in debuggers, they need to be enhanced so that they can be applied to large applications, e.g., operating systems.
2. *Control/Data Flow Mapping:* Given a system module, the tools should have the capability for the automatic development of control flow and parameter usage maps, which are useful in manual analysis of the system modules.

3. *Automated Module Exercising:* Given a system module, the tools should be able to automatically test all control paths under parameter conditions, thus freeing the evaluators from the monotonous task of manually exercising the control paths in a system module.
4. *Heuristic Module Analysis:* The tools should be able to produce representations of security relevant activities of the system modules. That is, the tools should be able to extract and represent security relevant information while ignoring information not relevant for security analysis.

Although Hollingworth *et al.* do not propose a new approach for identifying security flaws, they were among the first to suggest automation of at least some parts of the process.

2.2.3 Gupta and Giligor's Formal Penetration Analysis

In 1991, Gupta and Gilgor [GUPT91] described a formalized approach to penetration analysis. They based the approach on the hypothesis that *system flaws that cause a large class of penetration patterns can be identified in system source code as incorrect/absent condition checks or integrated flows that violate the intentions of the system designers*. They argue that a majority of penetration scenarios occur due to a small number of generic flaws. Furthermore, they contend that these generic flaws arise from the violation of one or more of the following properties:

1. *System isolation:* This property ensures that system boundaries, which are in the form of user interfaces, are protected from external users. Three aspects need to be addressed to guarantee that the system is isolated: (1) parameter checking at system interface, (2) user/system address space separation, and (3) system call selection and transfer of control.
2. *System noncircumventability:* This property refers to the guarantee that the system mediates all object references. In other words, there should be no way to bypass the access check mechanisms of the system.

3. *Consistency of system global variables and objects:* This property requires that system invariant assertions hold over global variables, objects and internal functions of the system. These assertions might require that a global variable not be alterable by unprivileged users or that global tables do not overflow.

4. *Timing consistency of condition checks:* This property ensures that the validity of conditions is not lost between the time the conditions are validated and an action that depends on these conditions is performed.

5. *Elimination of undesirable system and user dependencies:* This property ensures that no undesirable system and user dependencies exist within the system. An example of an undesirable system-user dependency is when an unprivileged user is able to lock or crash the system.

Gupta and Giligor have created a formal model of penetration analysis based on these properties. Its principal components are model interpretation constants. They are sets of conditions that need to be checked before an object is altered or viewed, before any system critical function is invoked, and at every system entry point. These interpretation constants are used to generate rules for state transitions. Any state transition must adhere to these rules; a flaw exists if it does not. Gupta and Giligor use this model as a foundation to develop an approach for penetration analysis, which consists of: (1) generating model interpretation constants, (2) using a tool to generate integrated flows (an integrated flow is information flow, function calls and dependencies, and condition checks along an execution path), and (3) applying the model rules to each such integrated flow. Gupta and Giligor have also developed a tool to automate penetration analysis based on their model.

Although Gupta and Giligor provide a promising approach for penetration analysis, their approach has weaknesses. The primary weakness stems from the fact that the model can only handle security flaws that arise due to violation of the conditions listed in the model invariants. Furthermore, the penetration analyst populates the conditions in the model invariants. Hence, as is the case with the FHM, such an approach relies heavily on the penetration analyst's ability, past experience and familiarity with the system.

2.2.4 Standards and Criteria

Standards and criteria have also been developed for evaluating security. For example, Common Criteria (CC) [CCIT] is the result of the combined effort of North American and several European governments to establish criteria for security evaluation in the information technology domain. CC provides requirements for Information Technology (IT) security in the form of categories of functional requirements. It also provides categories of assurance requirements that enable the evaluators to determine if the claimed security measures are effective and are implemented correctly. These assurance requirements are used to construct a set of seven assurance levels, from EAL 1 to EAL 7, with an increase in assurance as one moves to higher levels. In order to use CC to evaluate a product, developers must produce a construct called Security Target (ST), which is a collection of a security threats to the product, security requirements, and summary specification of security functions and assurance measures. The security requirements for a product are drawn from categories of the functional requirements provided to the evaluators. A product is evaluated against its ST on one of the seven assurance levels. Evaluation is carried out using Common Evaluation Methodology (CEM) [CEVM], which describes the minimum actions an evaluator must perform in order to conduct a CC evaluation.

It is important to note that CC is fundamentally different from the framework described in this dissertation in that CC focuses on the functional testing of the security features of a product, whereas the proposed framework focuses on identifying vulnerabilities in a product. Nevertheless, CC represents an important step in both creating and evaluating secure systems. However, some basic disadvantages impair CC. Since CC addresses a wide range of products, it is specified in general terms and is subject to a variety of interpretations that determine the level of security evaluation. This results in a lack of precision. CC also carries with it significant bureaucratic baggage, which casts doubts on its ability to evolve with the increasing number and complexity of threats [TRUS99]. Other similar approaches, such as Trusted Computer System Evaluation Criteria (TCSEC) [TCSE] and Information Technology Security Evaluation Criteria (ITSEC) [ITSE], suffer similar disadvantages. Additionally, the imposition of standards and criteria for the engineering of the software has not proven to be significantly effective to date. Pfleeger *et al.* [PFLE94] identify 250 standards for the engineering of software and conclude that these are mostly ineffective.

2.2.5 System Security Engineering-Capability Maturity Model (SSE-CMM)

All of the above approaches evaluate the product to assess its security. Alternatively, one can assess security by evaluating the process used to develop the product. One such approach being used widely is SSE-CMM [SSEC], a process reference model for evaluating and improving security engineering practices in an organization. It describes essential characteristics of an organization's security engineering activities, and which must exist to ensure good security engineering. SSE-CMM has two dimensions, domain and capability. The domain dimension consists of all practices that describe security engineering. The capability dimension represents practices that indicate process management and institutionalization capability, and which should be performed as a part of the domain practices. Based on the practices in the two dimensions, SSE-CMM provides five capability levels, Level 1 to Level 5, which represent increasing organizational capabilities in terms of security engineering. SSE-CMM also provides an appraisal method to evaluate the security engineering practices of an organization and define improvements in them. Using this appraisal method, organizations are evaluated and assigned one of the five capability levels.

The framework presented in this dissertation differs from approaches like the SSE-CMM; the proposed framework focuses on evaluating the product itself, while approaches like SSE-CMM focus on evaluating the development process of the product. Although process-based approaches are essential to evaluating security, they are not a replacement for product-based evaluation approaches – a product developed by an organization having a high SSE-CMM level is not necessarily secure. Therefore, to evaluate the security of a software process, product-based evaluation approaches have to be identified. Our proposed framework offers one such approach.

Summary

This chapter has outlined a proposed framework for assessing the security of a software process by testing it for the presence of vulnerabilities. The framework is composed of three components: (1) the object model, (2) the taxonomy of vulnerabilities, and (3) the V&V strategies component. The framework simplifies the testing of a software process for presence of vulnerabilities. The approach involves determining if the software process allows the violation of a constraint or an assumption; a vulnerability exists if either can be violated. Furthermore, the framework provides a comprehensive solution for assessing security, because it can be applied to assess security of any software process executing above the level of the operating system. In addition to providing an overview of the framework, this chapter also discusses existing approaches for assessing software security.

Chapter 3

Taxonomy of Software Vulnerabilities

Taxonomy, as defined by the *Merriam-Webster's Dictionary* [MWEB93], is the "study of general principles of scientific classification." A taxonomy embodies information about the objects it classifies, their characteristics, and their relationships to each other. In this way, it lays the foundation upon which a science is further developed.

A taxonomy methodically and uniquely classifies objects into categories. An object is first classified into a category, then reclassified into one of the subcategories, and so on, until no further classification is possible. Hence, a taxonomy can be depicted as a tree structure with the nodes denoting categories and subcategories, and the leaves denoting the objects. Classifying an object can be viewed as traversing the tree while selecting the nodes to which the object belongs. At every node, a decision must be made to select one of the child nodes to which the object belongs.

This chapter presents a taxonomy of vulnerabilities that provides an informative classification of vulnerabilities. The taxonomy is grounded in a theoretical model of computing, which establishes pair-wise relationships between vulnerabilities, software applications and computer system resources. This relationship captures the notion that software applications are exploited by violating (a) constraints assumed by computer system resources or (b) assumptions made about the usage of these resources. This taxonomy classifies these constraints and assumptions.

The theoretical model of computing, presented in Section 3.1, also reflects decomposition scheme similar to that used by the taxonomy, which consists of using computer system resources as categories. The top-level categories of both the theoretical model and the taxonomy are: (1) main memory, (2) Input/Output (I/O), and (3) cryptographic resources. Each of these categories is further divided into subcategories in which the constraints and assumptions are classified. Section 3.2 details the categories and subcategories of the taxonomy and lists the constraints and assumptions associated with each subcategory.

Because it is important to keep the taxonomy updated, Section 3.3 of this chapter details a process for adding new constraints and assumptions to the taxonomy. Section 3.4 presents the results of a preliminary study classifying vulnerabilities derived from CERT advisories issued from January 2004 to June 2004; Section 3.5 discusses the desired characteristics of vulnerability taxonomies; and lastly, Section 3.6 discusses related work pertaining to existing taxonomies.

3.1 Theoretical Foundation

This section presents a theoretical model of computing, which serves as a foundation for the taxonomy of vulnerabilities. The model establishes the relationship among vulnerabilities, computer system resources and the software process. This relationship, in turn, allows characterization of vulnerabilities in terms of constraints imposed by the computer system resources and assumptions made about the usage of these resources.

The model takes a high-level view of a software process executing on the computer system (Figure 3.1). The process can be visualized as a black box entity that takes input, performs some operations, and produces an output. However, it does not execute in a vacuum. To perform its requisite operations, it requires resources[1] such as (a) memory to store data, instructions and other execution specific parameters, (b) input/output (I/O) to receive and store input, store output, present output, as well as (c) cryptographic resources[2] to store secrets, and ensure data integrity. A software process, through the operating system interface, uses one or more of these resources to carry out its requisite functions.

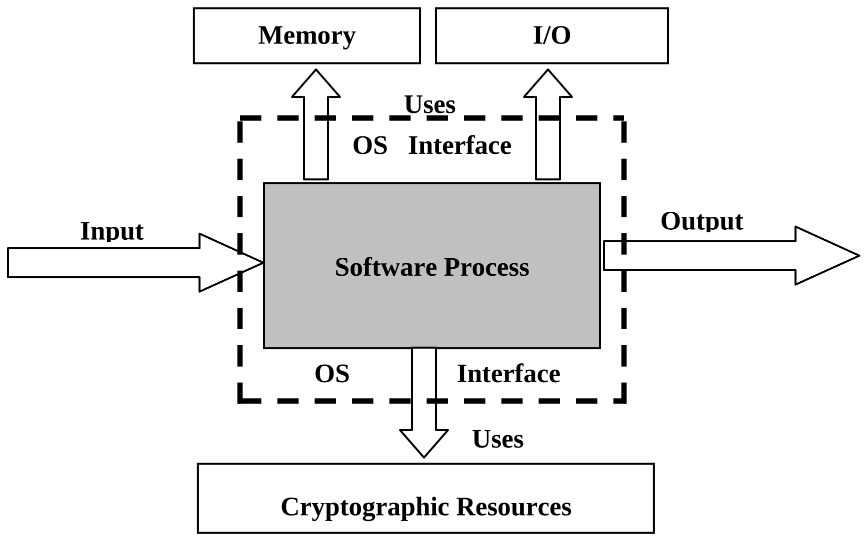


Figure 3.1 High level view of software process executing on a computer system

A software process, however, utilizes resources according to certain rules and limits, which are defined by the resources themselves or by the operating system that provides an interface to them. These rules and the limits manifest themselves in the form of constraints that a software process must adhere to in order to function correctly. For example, the system memory assumes a constraint that limits the total amount of memory available to a software process. In addition to such constraints, the software process itself makes assumptions about the resources it uses; for example, it assumes that it will be provided with the requisite amount of memory required for its

---

[1] Please note that there exist other resources that a software process uses, such as the processor. However, the theoretical model only covers resources whose usage leads to vulnerabilities.
[2] Please note that cryptographic algorithms and protocols are considered as resources. It is the author's contention that these algorithms and protocols should be treated as a resource and should not, under any circumstances, be developed by software developers.

operation. The process has to ensure that the assumptions it is making about usage of the resources are valid, and hold under current computational demands.

These constraints and assumptions give rise to vulnerabilities. It is the author's contention that a vulnerable state arises when a software process fails to enforce a constraint assumed by a resource and/or makes an incorrect assumption about its usage. In general, the resources and/or the underlying operating system enforce most of the constraints. For those that are not enforced at the resource or OS level, it is assumed that the software process will enforce or adhere to them. An enforcement or adherence failure creates the possibility that constraints can be violated. Additionally, the software process is expected to only make the assumptions that hold true under all circumstances.

At this point it is important to note is the difference between a vulnerability and an exploit. A vulnerability is present whenever constraints assumed by the resources or the assumptions made about their usage or both can be violated. An exploit is realized when these constraints or assumptions are actually violated. To further illustrate the concept, consider the example of buffer overflow. The system state can be described as a buffer that is allocated on the program stack with no bounds checking being performed on it. The program stack (part of main memory) assumes the following constraint on the software process: *Data accepted as input by the process and assigned to a buffer must occupy and modify only specific locations allocated to the buffer.* Neither the hardware nor the operating system enforce this constraint; in effect the software process is expected to adhere to it. However, the process does not enforce this constraint. Instead, it makes the assumption that *the data being accepted by the process and stored in the buffer will always be smaller than or equal to the size of the buffer.* Here, both the constraint assumed by the system memory and the assumption made by the software can be violated. These violable constraints and the assumptions are the reason that this system state is vulnerable. This state leads to the undesired and incorrect functionality when the process accepts and stores data that is larger than the buffer, thereby overwriting data that lie beyond the bounds of the buffer.

3.2 The Taxonomy

This section presents the taxonomy of vulnerabilities. The taxonomy provides a classification of vulnerabilities, which are in the form of violable constraints and assumptions. We identify the constraints and assumptions currently classified by the taxonomy by analyzing vulnerabilities, which in turn, are identified by analyzing security exploits. We have obtained security exploits from various sources including, but not limited to, books [HOWA02, VIEG01], mailing lists [BUGT], and websites [SEFO, PHRA].

The theoretical model presented in the previous section presupposes that constraints and assumptions are bound to the resources, an insight used to derive the classification scheme embodied in the taxonomy. This scheme uses resources as categories of the taxonomy. Effectively, the three top-level categories of the taxonomy are (1) Main Memory, (2) Input/Output, and (3) Cryptographic Resources (Figure 3.2).

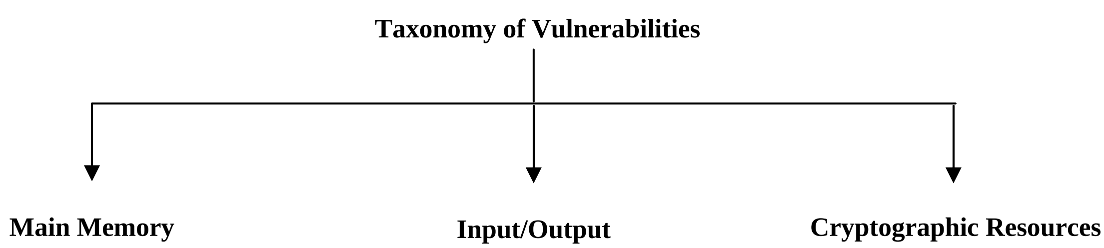


Figure 3.2 First Level Categories of the taxonomy

Each top-level category of the taxonomy is divided into additional subcategories, which themselves are resources, but represent different components of each top-level category. Each of the components has its own distinct set of characteristics. Additionally, the software process utilizes each for a different purpose. Therefore, each component has different constraints and assumptions associated with it, which necessitates different subcategories for the different components. Hence, main memory is divided into static memory and dynamic memory, I/O is partitioned into network interface and filesystem, and cryptographic resources into randomness resources and cryptographic algorithms and protocols. Appendix A presents the complete taxonomy with all of its categories and subcategories.

The constraints and assumptions are classified into one of the subcategories based on the resource with which they are associated. For example, an assumption such as *the software process will be provided with the required dynamic memory necessary for its execution* is categorized under dynamic memory, as that resource is the one in question.

It is important to note that the taxonomy views resources (its categories and subcategories) from the same perspective as that of the software process. That is, the taxonomy is not concerned with the physical or the operating system view of resources, but instead, how the software process is using those resources, and how the attendant constraints and assumptions apply to the software process.

Another important point to note is that although the taxonomy classifies both constraints and assumptions, it is of no consequence to distinguish between them. This is because the focus of the taxonomy is on vulnerabilities, and not on whether a particular statement being classified is a constraint imposed by a resource or an assumption made by the software process. Furthermore, distinguishing between a constraint and an assumption is difficult and, more often than not, a matter of point of view. For example, once the taxonomy has classified the statement, *data accepted as input by the process and assigned to the buffer must occupy and modify only specific locations allocated to the buffer*, it can be viewed as a constraint imposed by dynamic memory (a resource) as well as an assumption made by the process while accepting data. Also adding to the complications is the fact that assumptions are usually made about constraints. Hence, differentiating between constraints and assumptions provides no benefits and introduces unnecessary complexity.

The taxonomy affords a number of advantages in terms of testing a software process for presence of vulnerabilities. We discuss the primary ones below.

1. The taxonomy provides an inventory of vulnerabilities, where vulnerabilities are expressed in the form of violable constraints and assumptions. This characterization simplifies the task of identifying vulnerabilities present in a software process. One can test if the software process permits violation of a constraint or an assumption; a vulnerability exists if either can be violated.
2. As we demonstrate in Section 3.3, the process of classifying an exploit involves breaking it into multiple vulnerabilities, breaking each vulnerability into multiple constraints and assumptions, and finally classifying each of these constraints and assumptions. Hence, within the taxonomy a single exploit can engender multiple constraints and assumptions. This implies that a single exploit can be visualized as a *chain* of violable constraints and assumptions. If a single link in the chain is broken, the exploit cannot be completed. In other words, if a single constraint or assumption cannot be violated, the exploit will stop. This visualization implies that testing a software process against a carefully crafted subset of constraints and assumptions ensures its immunity against a large number of exploits.
3. The taxonomy requires minimal updates to keep it current. Again, this is because within the taxonomy, an exploit is represented by multiple constraints and assumptions. Hence, classifying new exploits that are variations of old exploits requires minimal or no updates, because new variations either violate all or most of the same constraints and assumptions used in the original exploits and have already been classified.
4. The taxonomy does not exhibit ambiguity. That is, a single constraint or an assumption can be classified only in a single category. This is a direct result of using resources to classify constraints and assumptions. Since constraints are imposed by resources, and assumptions are made about their usage, a single constraint or assumption can only be associated with a single resource, which implies that it can only be classified in a single category.

The remainder of this section is organized as follows: Subsections 3.2.1 through 3.2.3 describe each category of the taxonomy, its subcategories, and present the constraints and assumptions associated with each subcategory. Furthermore, for each subcategory, an example of a vulnerability resulting from violation of an associated constraint or an assumption is also provided; lastly, Subsection 3.2.4 discusses the scope of the taxonomy.

3.2.1 Main Memory

The main memory, physically, refers to memory chips (DRAM, SRAM, and so forth) that are present in a computer system. Conceptually, it refers to the storage space that the software process uses to store input, generated output, instructions, execution-specific parameters, and other objects necessary for the execution of the process. Each process executing on a computer system is allocated its own share of memory. Hence, each process has its own view of memory that is independent of the physical view. The underlying operating system supplies it with this view.

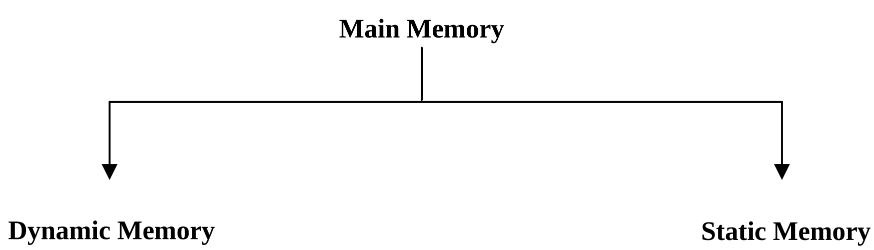


Figure 3.3 Subcategories of main memory category.

Main memory is divided into two subcategories, static memory and dynamic memory (Figure 3.3). Subsections 3.2.1.1 and 3.2.1.2 present the dynamic memory and static memory subcategories, list the constraints and assumptions associated with them, and present an example of the violation of a constraint or an assumption for each subcategory.

3.2.1.1 Dynamic Memory

Dynamic memory is the part of main memory whose size changes as the process executes. It consists of two components, the program stack and the heap, both of which store variables while the process is in execution. The program stack, or execution stack, is a contiguous block of memory used by the operating system and the process. It stores process data such as local variables, execution-specific parameters, and return addresses. The heap, on the other hand, is blocks of memory used to store dynamically allocated variables. For example, in C language the heap stores variables that are allocated using the *malloc( )* call, while in C++ it stores variables that are allocated using the *new* operator.

Since the nature of the variables being stored on the program stack and the heap is different, it is plausible to think that dynamic memory should be further divided into two subcategories, the program stack and the heap. In fact, not dividing it into subcategories might seem to be counter intuitive, because the exploits resulting from vulnerabilities on the program stack and heap have different impacts. However, the nature of the variables does not affect how the software process uses these resources. It uses these resources to store variables. This similar use implies that the constraints and assumptions associated with the program stack and heap are similar. Therefore, there is no need to divide the dynamic memory into more subcategories.

Figure 3.4 presents the constraints and assumptions associated with the dynamic memory, most of which are applicable to data held by variables stored on it. Failure to comply with any of the identified constraints and assumptions gives rise to a vulnerable condition.

Dynamic Memory

1. ***Data accepted as input by the process and assigned to a buffer must occupy and modify only specific locations allocated to the buffer.***

   This constraint is violated if the software process allows data larger than the size of the buffer to be written to the buffer. Since the data is larger than the buffer, it will overwrite memory that lies beyond the bounds of the buffer.

2. ***The process will not interpret data present on the dynamic memory as executable code.***

   This assumption is violated if the process is made to interpret data present on the dynamic memory as executable code, which, in turn, can be accomplished by changing process variables it holds, such as, return addresses, exception pointers.

3. ***Environment variables being used by the process have expected format and values.***

   This is a violable assumption, because a hostile entity can change the environment variables before the process begins execution. These variables are provided by the operating system and define the behavior of the process. A violation can occur if the process uses these variables and makes assumptions regarding their format and values.

4. ***The process will be provided with the dynamic memory that it requests.***

   This is a violable assumption, because the amount of dynamic memory available to a process is limited and depends on the total amount of memory available on the computer system and the number of running processes. A violation can occur if the process assumes that it has access to an unlimited amount of memory.

5. ***Data present on the dynamic memory cannot be observed while the process is in execution.***

   This is a violable assumption, because a hostile entity can run the process in a controlled environment and observe the contents of the dynamic memory, including any privileged data it holds.

6. ***Data owned by the process and stored on the dynamic memory cannot be accessed after the process frees the memory.***

   This is a violable assumption, because the memory being used by the process is not erased after the process frees it. Hence, another process, if allocated the same physical memory, can access the data left over by the previous process.

7. ***A pointer variable being used by the process references a legal memory location.***

   This is a violable assumption, because a pointer variable can point to any memory location, including memory locations outside the process address space. Furthermore, it can reference wrong variables, thereby creating illegal memory references.

Dynamic Memory (Cont.)

8. ***A memory pointer returned by the underlying operating system does not point to zero bytes of memory.***

   This is a violable assumption, because some operating system's can provide the process with a pointer that points to zero bytes of memory. Using this pointer will cause illegal memory references or overwriting of memory locations being used by other variables.

9. ***A pointer variable being used by the process cannot reference itself.***

   This assumption is violated if a pointer variable references itself. The consequences of this violation vary from garbage value being written to the memory location to the process going into an infinite loop.

10. ***Data accepted by the process must not be interpreted as a format string by the I/O routines.***

    This constraint is violated if a process accepts input and interprets it as a format string. A violation will at the very least reveal the contents of the process stack. Additionally, a hostile entity can provide the process a specially crafted format string that allows it to write data to the process stack.

11. ***The value of an integer variable/expression (signed & unsigned) accepted/calculated by the process cannot be greater (less) than the maximum (minimum) value that can be stored in the integer variable.***

    This is a violable assumption, because the maximum (minimum) value of an integer variable is determined by the amount of storage space provided to it by the underlying operating system. If the process tries to store a value that requires more storage space than is allocated to an integer, then the higher order bits of this value are dropped, resulting in a wrong value being stored.

12. ***An integer variable/expression used by the process as the index to a buffer must only hold values that allow it access to the memory locations assigned to the buffer.***

    This constraint is violated if the process does not restrict the value of the integer variable being used as an index to the array. A hostile entity can provide the process with any integer value to use as an index, which, in turn, gives it access to memory locations beyond the bounds of the array.

13. ***An integer variable/expression used by the process to indicate length/quantity of any object must not hold negative values.***

    This constraint is violated if the process uses an integer value to indicate length/quantity of an object and does not restrict the value to only positive values. A hostile entity can use a negative value to indicate the length of the object, thereby creating an error condition.

Figure 3.4 Constraints/Assumptions associated with dynamic memory

An Example

The constraint, *an integer variable/expression used by the process as the index to a buffer must only hold values that allow it access to the memory locations assigned to the buffer* (Figure 3.4, 12), is used here to provide a detailed example of a vulnerability. The vulnerable state occurs when a software process accepts from a user: two integer values – a position to store data and the value of the data itself. The software process stores the data in an integer array allocated on the program stack and uses the position value as an index to the array. If we assume that the process does not (or fails to) restrict the values of the index, then the user is free to enter any integer value he/she desires.

A vulnerability results from a software process allowing any value as the index for the array. Users can exploit this vulnerability by providing the software process with an index value greater than the size of the array, which permits them to write to a memory location beyond the array. Because users also provide data to be placed at the designated index value, they can write any value to any memory location that lies beyond the bounds of the array. Thus, users can overwrite the return value on the program stack with the value of their choice, which gives them the ability to execute any code they choose with the privileges of the software process. Thus, by failing to ensure that the constraint is not violated, the software process creates a vulnerable system state.

3.2.1.2 Static Memory

The second subcategory of the main memory is static memory. As the name suggests, the size of the static memory is fixed before the process begins execution and does not change as the execution proceeds. There are two major components of static memory, the data segment and the block storage segment (BSS). The data segment stores initialized global (static) variables; the BSS stores un-initialized global (static) variables. The software process uses the data segment and BSS in a similar way, i.e., to store data whose size is fixed before execution starts. Figure 3.5

presents the constraints and assumptions associated with static memory. Again, failure to comply with the identified constraints and assumptions gives rise to potential vulnerable states.

Static Memory

1. ***Data accepted as input by the process and assigned to a buffer must occupy and modify only specific locations allocated to buffer on the static memory.***

   This constraint is violated if the software process allows data larger than the size of the buffer to be written to the buffer. Because the data is larger than the buffer, it will overwrite memory that lies beyond the bounds of the buffer.

2. ***Data held on the static memory cannot be observed while the process is in execution.***

   This is a violable assumption, because a hostile entity can run the process in a controlled environment and observe the contents of the static memory, thereby gaining access to any privileged data held in static memory.

Figure 3.5 Constraints/Assumptions associated with static memory

An Example

The constraint, *data accepted as input by the process and assigned to a buffer must occupy and modify only specific locations allocated to buffer on the static memory* (Figure 3.5, 1), is used to provide a detailed example of a vulnerability. The vulnerable state occurs when a software process accepts a string as input from the users and copies it into a fixed length buffer without performing any bounds checking. The software process declares this buffer along with several other variables as global variables, thereby implying that all these variables and the buffer reside on the static memory.

A vulnerability results because the software process does not perform bounds checking before copying data into the buffer. Users can exploit this vulnerability by providing the software process with a string that is larger than the size of the buffer. Because, the software process performs no bounds checking, the string will overwrite the variables that lie beyond the bounds of the buffer. Depending on the variables being overwritten and the data with which they are being overwritten, the consequences of this attack can vary from the process producing wrong output to the abnormal termination of the process. Thus, by failing to enforce this constraint, the software process creates a vulnerable system state.

3.2.2 Input/Output (I/O)

The second category of the taxonomy is I/O. The software process uses I/O resources to receive input and to communicate output. Physically, I/O devices correspond to components like the mouse, keyboard, monitor, hard disks and network cards. Conceptually, I/O corresponds to filesystem, network interface, standard input interface, and standard output interface.

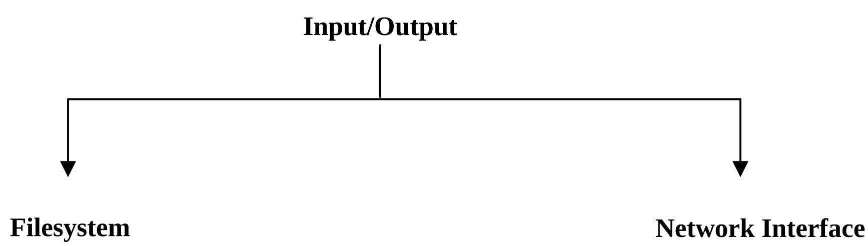


Figure 3.6 Subcategories of I/O category.

I/O, as a category of this taxonomy, is divided into two subcategories: filesystem and network interface. Subsections 3.2.2.1 and 3.2.2.2 present the filesystem and network interface subcategories, list the constraints and assumptions associated with them, and present an example of the violation of a constraint or an assumption for each subcategory.

3.2.2.1 Filesystem

The first subcategory of I/O is the filesystem, which is characterized by an encoding of the addressing scheme, boot block, super block, inode structure, storage blocks, and so forth. Different filesystem types use different encoding schemes. For example, ext3, FAT, and NTFS all use different encoding schemes. However, a software process, while using the filesystem as a resource, does not perceive it as an encoding scheme; instead the process views the filesystem as a hierarchy of connected directories containing metadata about files and the files themselves.

Filesystem

1. ***Access permissions assigned to newly created files/directories are such that only the required principals have access to them.***

   This constraint is violated if a software process creates new files and directories without assigning them proper access permissions, which, in turn, allows a hostile entity to access them. A violation occurs when a hostile entity actually reads or modifies these files and directories.

2. ***Access permissions of the files/directories being used by the process are such that only the required principals have access to them.***

   This is a violable constraint, because if a software process uses file/directories, already present on the filesystem, which do not have proper access permissions, then there exists a probability that a hostile entity has already read or modified these files. A violation occurs when the process uses the files/directories after the entity has read or modified them.

3. ***A file being created by the process does not have the same name as an already existing file.***

   This is a violable assumption, because a hostile entity can create a file with the same name as a file being created by the software process and place it in the same directory where the software process was going to place its file. Depending on the underlying operating system, the consequences of this attack vary from the process using the file placed by the hostile entity to the process terminating execution.

4. ***A filename (including path) being used by the process is not a link that points to another file for which the user, executing the process, does not have the required access permissions.***

   This is a violable assumption, because a hostile entity can provide the software process with a file that is a link to another file. If the process has more privileges than the entity, then the entity can point the link to a file to which it does not have access, thereby resulting in the entity gaining improper access to the file pointed to by the link.

5. ***A file created/populated by a principal other than the process and being used by the process will have expected format and data.***

   This is a violable assumption, because a hostile entity can provide the software process with specially crafted files containing corrupt data. A violation occurs when the software process uses these corrupt files.

Filesystem (Cont.)

6. ***A file being used by the process cannot be observed/ modified/replaced while the process is in execution.***

   This constraint is violated if the software process does not lock the files that it is using. A violation occurs when a hostile entity can read, modify or replace these files while the process is in execution.

7. ***A file/directory being used by the process and stored on the file-system (information used by the process over multiple runs) cannot be observed/modified/replaced in-between these runs.***

   This is a violable assumption, because a hostile entity can observe, modify or replace files created by the software process and stored on the filesystem after the process stops execution and before the next run.

8. ***Data held by files owned/used by the process must not be accessible after the process deletes them.***

   This is a violable constraint, because files stored on the filesystem are not erased after the software process deletes them. The operating system simply deletes their name from the list of existing files, but anyone with proper access permissions can access this data by directly accessing the physical storage.

9. ***The process must be provided with the filesystem space that it requests.***

   This is a violable constraint, because filesystem space is a limited resource and the process will not always be provided with the filesystem space that it requires.

10. ***A file having proprietary or obscure format cannot be understood or modified.***

    This is a violable assumption, because proprietary or obscure file formats are not enough to keep the contents of a file secret. Tools and techniques a hostile entity can use to reverse engineer these formats to reveal the contents of the files exist.

Figure 3.7 Constraints/Assumptions associated with filesystem

Figure 3.7 presents constraints and assumptions associated with the filesystem subcategory. These constraints and assumptions focus directly on files and directories. Historically, software processes have been particularly vulnerable to exploits involving files/directories because the filesystem, itself, does not enforce the identified constraints. By failing to address these omissions, and by making yielding assumptions regarding filesystem usage, the software process creates states wherein constraints and assumptions can be violated. This, of course, creates vulnerabilities and, in turn, permits exploits.

An Example

The assumption, *a filename (including path) being used by the process is not a link that points to another file for which the user, executing the process, does not have the required access permissions* (Figure 3.7, 4), is used to provide a detailed example of a vulnerability. The vulnerable system state occurs when a software process executes with setuid permissions. In other words, although a user is executing the process, it has effective root user privileges, which give the process access to all resources of the computer system. The software process, in course of its execution, asks the user to provide it with the name of a file to which it has to write data.

A vulnerability results from the software process not checking if the user executing the process has access permission to the file pointed to by the symbolic link. A user can exploit this vulnerability by providing the software process with the name of a symbolic link which points to another file, to which he/she does not have the required access permissions. If, for instance, the symbolic link points to the system password file, the software process will follow the link, open the system password file, and write data to it. Depending on the data the process is writing, the consequences of this exploit vary from trashing of system password file to the user gaining root privileges. Therefore, by not taking this assumption into account, the software process enables this assumption to be violated, which creates the vulnerability. A detailed explanation of this exploit is described later in this dissertation.

3.2.2.2 Network Interface

The second subcategory of I/O is the network interface, which physically corresponds to the network interface card. This card enables a computer system to talk to other computer systems on the network. A software process uses this card to communicate over the network, but it does not directly interact with the physical interface. Instead it uses ports to send and receive data. A port is an end of a logical connection, which a software process uses to communicate over the network. Ports are numbered from 1 to 65535, with different ports being used by different applications for different services. A software process can attach itself to any open port and use it to send and receive data. In this way, the port serves as an interface for the software process. Thus, from the perspective of the software process, the network interface is simply a resource that can be used to send and receive data.

Figure 3.8 presents constraints and assumptions associated with the network interface. Typically neither the network interface, the operating system, nor the software process enforce the identified constraints and assumptions. More specifically, the software process makes decisions based on the data received from the network interface, with little consideration as to whether or not the constraints and assumptions are being upheld, which, in turn, creates vulnerable system states.

Network Interface

1. ***The data received by the software process through the network interface is neither read nor modified by anyone other than the intended recipient.***
   This is a violable assumption, because a hostile entity can intercept, read or modify, and resend data that has been sent to the software process by sitting between the sender and the software process. A violation occurs when a hostile entity reads or modifies and resends data that has been sent to the software process.

2. ***The data received by the software process through the network interface is from a legitimate client or peer or server and has expected format and length.***
   This is a violable assumption, because any entity can send data using the network interface if it knows the IP address of the host machine and the port number being used by the software process. A violation occurs when the software process uses corrupt data sent by a hostile entity.

Network Interface (Cont.)

3. ***The data sent by the software process via the network interface will not be read/modified before it reaches its destination.***
   This is a violable assumption, because a hostile entity can intercept, read or modify, and resend data being sent by the software process by sitting between the software process and receiver of data. A violation occurs when a hostile entity reads or modifies and resends data being sent by the software process.

4. ***The software process will be able to utilize the network interface to send and receive data.***
   This is a violable assumption, because no guarantee that the software process will always be provided with access to the network interface exists. A violation occurs if the software process does not take this assumption into account and is not able to use the network interface.

5. ***The byte order of numerical data accepted from the network interface is same as that of the host machine.***
   This is a violable assumption, because the format of numerical data of a host machine and the network can be different and the software process has to change the format before using it. Making this assumption results in wrong numerical values being used by the software process.

Figure 3.8 Constraints/Assumptions associated with network interface

An Example

The assumption, *the data received by the software process through the network interface is from a legitimate client or peer or server and has expected format and length* (Figure 3.8, 2), is used to provide a detailed example of a vulnerability. The vulnerable system state occurs when a server process accepts input from various client processes running on other computer systems, which it then copies into a fixed length buffer for further processing. The server process expects the input to have an agreed upon format and maximum size.

A vulnerability results when the server process assumes that the data accepted from the network interface conforms to a predefined structure. In particular, a user can send the server process data that is larger than the size of the fixed length buffer, into which this data is being copied. This results in memory beyond the bounds of the buffer being overwritten with the users data. Depending on the data supplied and the location of the fixed length buffer, the consequences of this exploit can vary from the server process crashing, to the user gaining control of the server by performing a buffer overflow exploit.

3.2.3 Cryptographic Resources

The third category of the taxonomy is cryptographic resources, which refers to the algorithms and protocols contributed by the discipline of cryptography. The field of cryptography is extensive[3] and has been in existence for many years [SING00]. In computer systems, cryptography provides the capability for securing data and resources in the areas of confidentiality, authentication, integrity, and nonrepudiation [VIEG01]. Confidentiality means that only authorized entities are able to understand data. Authentication focuses on ensuring that only authorized entities are able to access data/resources or to supply data. Integrity means that only authorized entities should be able to modify data. Nonrepudiation, which concerns the communication of messages, means that the sender of a message should be able to substantiate that the receiver received the message; and conversely, the receiver should be able to substantiate that the sender actually sent the message.

The category of cryptographic resources differs from the previous two categories of this taxonomy. The previous categories have physical devices associated with them and a software process uses these devices as resources; cryptographic resources, more often than not, are software components themselves, used by other software processes. However, cryptographic resources has been included as a category of this taxonomy because they are, in their own right, resources used by the software process. Developing or even implementing cryptographic algorithms and protocols is a very difficult task. The author shares the opinion with other security experts [VIEG01] that no software developer should even try to engage in this activity. In fact,

---

[3] This by no means is a complete description of cryptography. For a detailed treatment of the subject please refer to a book. There are several good books on the subject, e.g., [SCHN96].

many well-know exploits have occurred when software applications have used “home brewed” cryptographic algorithms and protocols [NETS99].

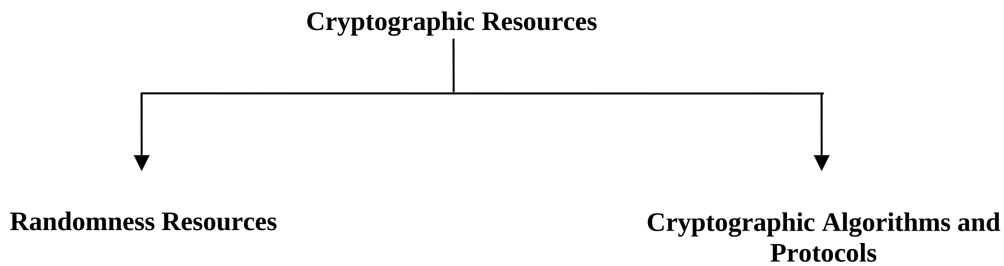


Figure 3.9 Subcategories of cryptographic resources category.

The two subcategories of the cryptographic resources category are randomness resources and cryptographic algorithms and protocols (Figure 3.9). Subsections 3.2.3.1 and 3.2.3.2 present the randomness resources and cryptographic algorithms and protocols subcategory, list the constraints and assumptions associated with them, and provide an example of the violation of a constraint or an assumption for each subcategory.

3.2.3.1 Randomness Resources

Randomness, the first subcategory of cryptographic resources, refers to the generation and use of random numbers, which are a series of numbers whose values are uniformly distributed over a set. Ideally, the probability of predicting the next number in the series is zero. Many algorithms have been developed for generating random numbers. These algorithms, called Pseudo-Random Number Generators (PRNGs), accept input, called a seed, and use it to generate a series of random numbers. The series generated depends on the seed provided to the PRNG. To generate an unpredictable series of numbers, an unpredictable seed must be used. Thus, the problem shifts its focus from the generation of random numbers to the generation of a seed that is truly random or close to truly random. The seed for PRNGs is generated using events that occur in a computer system. Although computers are deterministic machines, certain events occurring in a computer system, including the press of a key on the keyboard, mouse movements, CPU scheduling and difference between CPU timer and interrupt timer are difficult to guess. These events are used to generate unpredictable seeds.

Because PRNGs are algorithms running on deterministic machines, they do not generate true random numbers. Therefore, a measure of randomness called entropy exists. In information theory, entropy is defined as the amount of information present in a data set. For example, flipping a coin results in a head or a tail. One bit can represent this information. Thus, flipping a coin has 1 bit of entropy. A direct relationship exists between entropy and randomness, and is embodied by an alternative definition of true random numbers. This definition states that a series of numbers is truly random if it cannot be compressed. In other words, information present in this series cannot be expressed in a shorter form. Thus, a series of true random numbers has information equal to the number of bits in that series. This, in turn, implies that the series has entropy equal to the number of bits present in it. This relationship allows one to use entropy as a measure of how random a particular set of numbers is.

The generation and use of random numbers is one of the least understood aspects in the design and implementation of a software system. Consequently, software processes tend not to consider the constraints associated with randomness and make assumptions that can be violated. The constraints and assumptions become particularly critical when random numbers are used for cryptographic purposes. Such vulnerabilities usually render the cryptographic algorithms and protocols useless. Figure 3.10 presents constraints and assumptions associated with randomness. Most of these address the use of random numbers. Because the taxonomy considers random numbers a resource that a software process uses, it is assumed that the software process is using standard random number generation techniques.

Randomness Resources

1. ***The series of random data being produced by the PRNG is unpredictable (assuming unpredictable seed).***

   This is a violable assumption, because there exist PRNGs that produce predictable random data series. These PRNGs produce data that is random in the sense that each number has an equal probability of being the next number in the series, but it is computationally feasible to predict the next number in the series. Hence, the random data produced by these generators cannot be used for cryptographic purposes.

2. ***The seed being used by the PRNG is unpredictable.***

   This assumption is violated if the seed being used by the PRNG is predictable. Random data being produced by the PRNG is dependent upon the seed provided as input to the PRNG. Thus, using an unpredictable seed is a key requirement for producing unpredictable random data series. A predictable seed would result in a predictable random data series, which cannot be used for cryptographic purposes.

3. ***The process will have easy access to entropic data on a computer system.***

   This is a violable assumption, because computers are deterministic machines. Therefore, it is very difficult to have to access entropic data on a computer system.

4. ***The process will be able to accurately estimate entropy of a data set.***

   This is a violable assumption, because currently available approaches for estimating entropy provide only a coarse approximation of the value of entropy for a data set. Hence, a process, even while employing these approaches, should be conservative in estimating entropy, because estimating a higher value of entropy than is actually present leads to a false sense of security.

6. ***User selected passwords/keys will have a sufficient amount of entropy.***

   This is a violable assumption, because user selected passwords/keys do not have sufficient amount of entropy and typically are highly predictable. Hence, they should not be used in cryptographic algorithms or protocols that require highly entropic keys/passwords.

6. ***If two different seeds are provided to the PRNG, it is computationally infeasible to produce the same series of data both times.***

   This is a violable assumption because of the structure of the data series some PRNGs produce, which can be visualized as a series of numbers on the circumference of a circle. The seed, given as input to these PRNGs, selects a point on this circle from where the PRNG starts to output random numbers. It is possible, especially if the circle is small, for two different seeds to select the same point on the circle (modulo arithmetic) and produce the same random data series. These PRNGs are not suitable for cryptographic purposes and can be identified from current cryptographic literature.

Randomness Resources (Cont.)

7. ***Given that the PRNG is continuously producing random data, it is computationally infeasible to produce the same sequence of random data after some time.***
   This is a violable assumption, because it is possible that PRNGs, which adhere to the circular structure visualized in item 6 and produce random data continuously, will repeat the data series after some time.

Figure 3.10 Constraints/Assumptions associated with randomness resources

An Example

The assumption, *the seed being used by the PRNG is unpredictable* (Figure 3.10, 2), is used here to provide a detailed example of a vulnerability. The vulnerable software process is an online poker game. The process uses a popular and readily available PRNG to generate a series of random numbers, which it uses to shuffle cards for the poker game. The software process uses the current system time, which is an easily predictable number, as a seed for the PRNG [VIEG01].

A vulnerability results from the software process using a predictable number to seed the PRNG. A user can use the same PRNG that the software process is using and provide it with the seed being used by the software process. This provides him or her with access to the random data series being used by the software process. Because, the user has access to the random data series being used to shuffle the cards, he or she can predict the order of the cards in the poker deck and, as a result, have complete control of the poker game. Thus, the software process makes this violable assumption, thereby creating a vulnerable system state.

3.2.3.2 Cryptographic Algorithms and Protocols

The second subcategory of cryptographic resources, cryptographic algorithms and protocols, refers to algorithms and protocols that provide to the software process the services of confidentiality, integrity, authentication, and nonrepudiation. This subcategory includes:

- encryption algorithms, such as DES, RC4, and RSA, to ensure confidentiality of data,

- authentication protocols, such as Kerberos, to ensure that only authorized entities have access to data and resources,
- cryptographic checksums and hashing algorithms, such as MD5, SHA1, to ensure data integrity, and
- protocols and algorithms, such as digital signatures and PKI, that promote nonrepudiation.

The taxonomy treats cryptographic algorithms and protocols as resources, just as it does randomness. Like the generation and use of random numbers, the use of these algorithms and protocols is complex. Figure 3.11 presents constraints and assumptions associated with cryptographic algorithms and protocols. Note that they do not address specific algorithms and protocols; instead, they focus on their usage by the software process. Several of the constraints and assumptions refer to the length of a key or a hash as being sufficient. Sufficient length is used instead of actual numerical values, because the required length for keys and hashes changes with the algorithm/protocol and with time. Therefore, to ensure the longevity of constraints and assumptions, the term *sufficient length* is used instead of actual numerical values.

Cryptographic Algorithms and Protocols

1. ***Random data being used by the cryptographic algorithm/protocol is unpredictable.***
   This is a violable assumption, because there exist PRNGs that produce predictable random data series. Cryptographic algorithms and protocols require a random data series to exhibit two properties: (1) the data series is statistically random. That is, each number in the set of numbers has an equal probability of being the next number in the random data series; (2) the next number in the series is unpredictable. There are PRNGs that produce random data that is statistically random but predictable. These PRNGs cannot be used as a source of random data for cryptographic purposes.

Cryptographic Algorithms and Protocols (Cont.)

2. ***The length of the key being used by the cryptographic algorithm or protocol is sufficient.***

   This assumption is violated if the minimum length of the key, being used for a cryptographic algorithm or protocol, is less than the current standard. Most cryptographic algorithms and protocols use keys for a variety of purposes, such as keeping secrets and restricting access. The length of these keys is a critical factor in security of these algorithms or protocols. Typically, if a small key is used, the algorithm or protocol can be compromised easily. The length of the key required for keeping the algorithm or protocol secure changes with time and can be found in the current cryptographic literature. Using a key that is smaller than the current standard puts the algorithm or protocol at risk of being compromised.

3. ***The hashing algorithm will not produce same hash for two different inputs.***

   This is a violable assumption, because there is a possibility that a hashing algorithm produces same hash for two different input texts. These algorithms are considered as compromised and cannot be used for cryptographic purposes. Information about compromised hashing algorithms is available in current cryptographic literature.

4. ***The process cannot use encryption to ensure data integrity.***

   This is a violable constraint, because encryption only ensures data confidentiality by changing plain text into ciphertext. No one can make sense of ciphertext without changing it back to plain text. But anyone with access to ciphertext can change it even if it does not make sense. Hence, encryption just ensures data confidentiality and not data integrity.

5. ***The process cannot use a key more than once for a stream cipher.***

   This is a violable constraint, because using a key more than once for a stream cipher can compromise the ciphertext. Stream ciphers use a key to produce a stream of random data, which they XOR with the plain text to produce ciphertext. If the process uses the same key more than once, then the resultant random data series is same, which, in turn, implies that multiple instances of plain text data will be XORed with the same random data series. There are known crypto-analytical techniques for compromising ciphertext’s produced by XORing multiple instances of plain texts with same random data series. Hence, a process should never use same key more than once for stream ciphers.

Cryptographic Algorithms and Protocols (Cont.)

6. ***The process cannot use one time pads to encrypt a large quantity of data.***
   This is a violable constraint, because one time pads require truly random or close to truly random data that has bit length equal to that of the plain text. It is very difficult to have access to large quantity of high quality random data. Hence, one time pads should not be used to encrypt large quantity of data.

7. ***The process cannot use keys that are self reported by a client or a server.***
   This is a violable constraint, because a hostile entity can masquerade as a client or a server and send its own keys to the process. Hence, any key that is self reported by a client or server cannot be trusted without some form of validation.

8. ***The process cannot use obfuscation instead of encryption to ensure confidentiality.***
   This is a violable constraint, because there exist tools and techniques that we can use to reverse engineer obfuscated data to reveal any privileged data. Obfuscation is the process of changing data so as to make it difficult to perceive or understand. In this way, it can be visualized as a much weaker form of encryption, which can be compromised easily. Thus, a process should not use obfuscation to ensure data confidentiality.

9. ***The process cannot store keys/passwords in clear text.***
   This constraint is violated if a process stores keys/passwords in plain text. Keys/ passwords are very important for the security of cryptographic algorithms and protocols. Compromise of a key or a password essentially means complete compromise of the cryptographic algorithm or protocol. Hence, they should never be stored in plaintext.

Figure 3.11 Constraints/Assumptions associated with cryptographic algorithms & protocols

An Example

The constraint, *the process cannot use encryption to ensure data integrity* (Figure 3.11, 4), is used here to provide a detailed example of a vulnerability. The vulnerable state occurs when a software process, running on a computer system in a bank, sends financial transactions to a central server. Although the process encrypts all transactions using a secret key, the format of the transactions follows an industry standard consisting of date, time, type and amount that is publicly available [SCHN96].

A vulnerability results from the software process using standard encryption details to ensure both data confidentiality and integrity. A user can exploit this vulnerability by intercepting the transactions being sent to the server. Because the user does not have access to the key being used to encrypt the transactions, she/he cannot decrypt the transactions. However, because she/he is familiar with the format of the transactions, she/he can change the time in encrypted transactions and then resend the transactions. The consequences of the attack can vary from multiple deposits for a single account to complete destruction of the transaction data stored at the central server. Therefore, by failing to consider this constraint, the software process creates a vulnerable system state.

3.2.4 Scope

The taxonomy is applicable to any software application executing above the level of the operating system. However, the level of abstraction of constraints and assumptions can present itself as a problem when applying the taxonomy to different applications. This problem stems from the broad range of applications available today, and include personal e-mail and document processing, web servers and applications controlling the national power infrastructure. The expectation that constraints and assumptions be applicable to all of these applications introduces the question of whether the constraints and assumptions and, in turn, the taxonomy should be general and apply to all applications, or be specific and apply to a more restricted set of applications. Restricting applicability undermines the scope of the taxonomy. Because of the wide range of applications available today we have elected to keep the taxonomy generally applicable. This implies that the users of the taxonomy may have to adapt some *constraints* and *assumptions* to nuances of the software application. To facilitate this adaptation, a conscious effort has been made to balance the level of abstraction of constraints and assumptions so that they can be applied to a wide variety of applications.

Scope definition is further exacerbated from the lack of a clear and agreed upon definition of a vulnerability. Currently, anything that has the slightest security implications is labeled as a vulnerability. Krsul [KRSU98] lists a number of definitions of a vulnerability that occur within just the current academic literature. This vagueness creates confusion, and makes it difficult to uniquely classify vulnerabilities within a taxonomy. The theoretical model of computing presented in Section 3.1, however, assists in the vulnerability classification process by establishing the relationship between the vulnerabilities, software process and the computer system resources. This relationship implies that the taxonomy classifies only vulnerabilities that can be characterized as constraints assumed by computer system resources or as assumptions made about the usage of these resources.

3.3 Extending the Taxonomy

This section outlines the process for adding new constraints and assumptions to the taxonomy of vulnerabilities. The objective of defining the process is to help ensure that the taxonomy remains relevant and does not loose its usefulness over time. This need is underscored by the dynamism present in the field of computer security, and the continuing evolution of new software development paradigms and languages. The process is also important, because it provides a more detailed view of the composition of the taxonomy and its usage.

Adding constraints and assumptions to the taxonomy is a multi step process. Below, we present a step-by-step account of the process:

Step1: Identifying vulnerabilities

The process starts with a security exploit that is analyzed to identify the vulnerabilities that it is utilizing. The objective of the analysis is to separate the method of exploitation from the vulnerabilities. This enables identification and definition of the vulnerable system state. Again, it is important to note that a single exploit can utilize multiple vulnerabilities. For example, a successful buffer overflow exploit uses a minimum of two vulnerabilities: a buffer whose bounds are not enforced, and the software process interpreting instructions held in dynamic memory as data.

Step 2: Identifying resources utilized by vulnerabilities

Once the identification of a security exploit's vulnerabilities is complete, the next step is to identify the resources associated with the vulnerabilities. Initially, a vulnerability is analyzed to identify a high-level candidate resource. The taxonomy is then used to refine that resource classification (resources represent the categories and subcategories of the taxonomy). This step is repeated for all identified vulnerabilities.

Step 3: Identifying constraints and assumptions

With both the vulnerabilities and associated resources identified, the third step is to identify the appropriate constraints and assumptions. The taxonomy provides a candidate set of potential constraints and assumptions for the resources in question. To identify appropriate constraints and assumptions, select a vulnerability and ask the following two questions: W*hat constraint assumed by the resource is being violated?* and W*hat assumption made by the software process regarding the usage of that resource is being violated?* Answers to either (or both) of the above questions provide the candidate constraint or assumption (or both) whose violation is the source of the vulnerability. This step is repeated for all vulnerabilities identified in Step 1.

Step 4: Adding constraints and assumptions to the taxonomy

If the identified constraints or assumptions do not appear under the appropriate resources, then they are added to the taxonomy.

The following example illustrates the process of identifying and classifying constraints and assumptions. The security exploit being analyzed is the "time of check time of use" exploit. A popular version of this exploit involves a software process running on the UNIX operating system with setuid permissions. In other words, a user is executing the process, but it has effective root user privileges, which gives the process access to all resources of the computer system. The software process, in course of its execution, asks the user to provide it with the name of a file to which it has to write data. The process has to access the file and ensure that the user running the process has permissions to read and write to it. To execute these two steps, the process first checks if the user has the required permissions by using the *access* system call, and then opens the file for writing, if and only if, the user has the required permissions. The issue here is the time gap between these two steps, because the user can exploit the presence of this time gap to gain access to any file on the computer system. A step-by-step procedure for executing this attack follows:

1. The user first provides the process with the name of a symbolic link that points to a file. The user has the required permissions to access both the symbolic link and the file.
2. The process successfully checks that the user has access permissions to this file.
3. Before the process can open the file, the user changes the symbolic link to point to another file to which he or she does not have access permissions.
4. The process, which has root privileges and can access any file, opens the file pointed to by the symbolic link and writes to it.

Because the time gap for changing the symbolic link is very small, it is difficult to change the link manually. Therefore, to exploit this vulnerability, the user uses a script to replace the file within the required time gap. The script runs in parallel with the software process and continuously tries to replace the file.

The seriousness of such a compromise is realized when the user makes the process write to the system password file. Depending on the data the software process is writing, the consequences of this attack can vary from trashing of the password file to the user gaining root access of the system.

The process of classifying the “time of check time of use” exploit is provided next.

1. The process starts with analyzing the exploit to separate the method of exploitation from the vulnerability being exploited. The method of exploitation in this case consists of using the time gap between checking permissions and accessing the file to replace the file. The vulnerability is the system state that affords this functionality. This system state can be defined as the software process accepting a file name from the user and using that file name first to check access permissions and then to access the file.
2. The next step is to identify the resource associated with the vulnerable system state. In this case, it is the filesystem, which in terms of the structure of our taxonomy, is a subcategory that falls under the category of I/O.
3. The next step in the process is to derive the constraints and assumptions that are the source of this vulnerability. This involves analyzing the relationship between the vulnerability and the resource to identify the conditions the exploit is violating. In this case, the taxonomy is first searched for a constraint like: *a file being used by the process cannot be replaced while the process is in execution*.
4. If the violated constraint is not found in the taxonomy, an appropriately worded one (like the above) is added to the taxonomy under the filesystem subcategory.

The process of adding constraints and assumptions to the taxonomy consists of analyzing an exploit to identify one or more vulnerabilities, and deriving the appropriate constraints and assumptions from each vulnerability. For this reason, a number of constraints and assumptions, more often than not, represent a security exploit. This underscores the fact that a security exploit often enlists a chain of violations of constraints and assumptions. If a single link in this chain is broken, the exploit will not work. In other words, the exploit will stop if one constraint or assumption among the exploit's list of these cannot be violated. Although it is possible that the exploit will not be rendered completely unsuccessful by preventing the violation of a single constraint or assumption, the damage will be restricted to that associated with the violated constraints and assumptions. Visualizing exploits as chains of violated constraints and assumptions implies that preventing violation of a carefully crafted subset of constraints and assumptions ensures that the software process is secure against a large number of exploits.

3.4 Verifying the Taxonomy

It is important that we verify the taxonomy of vulnerabilities in terms of its utility and validity. One approach towards verifying the taxonomy is to compare it with existing taxonomies. However, because our taxonomy is distinctively different, a direct comparison with the existing taxonomies is inappropriate. The categories and subcategories, the distinction drawn between vulnerabilities and exploits, and the inclusion of constraints and assumptions amplify these differences. Hence, we elected not to pursue this approach for verifying our taxonomy.

An alternative approach to verifying the taxonomy is to apply it to existing real world vulnerabilities. To attest to its effectiveness, this section presents a taxonomic classification of vulnerabilities obtained from advisories issued by CERT [CERT] from January 2004 to June 2004. CERT was chosen as the source of vulnerabilities, because it has been fairly consistent and complete in issuing security advisories. The advisories themselves consist of a description of the application being exploited, a description of exploit, and the impact of the exploit, as well as links to more information about the exploit.

A total of 175 advisories were collected for classification by the taxonomy. Classification included identifying vulnerabilities and constraints and assumptions using the process outlined in the previous section. In addition to the information provided in the advisories, additional information was collected by pursuing links provided in these advisories. Nonetheless, several of the reported exploits could not be analyzed because of missing information, or because the exploits targeted operating system or hardware vulnerabilities.

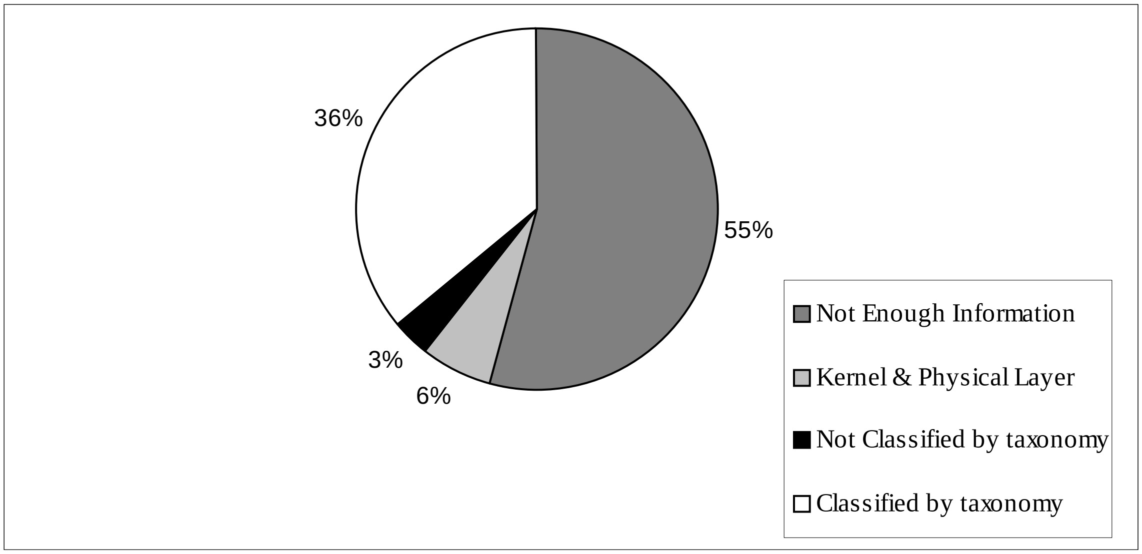


Figure 3.12 Distribution of advisories classified and not classified by the taxonomy

Figure 3.12 presents the results of classifying the 175 CERT *advisories.* 95 (55%) advisories could not be classified due to lack of required information; 11 (6%) were not classified because they were kernel or hardware related advisories. We were able to classify 63 of the remaining 69 advisories.

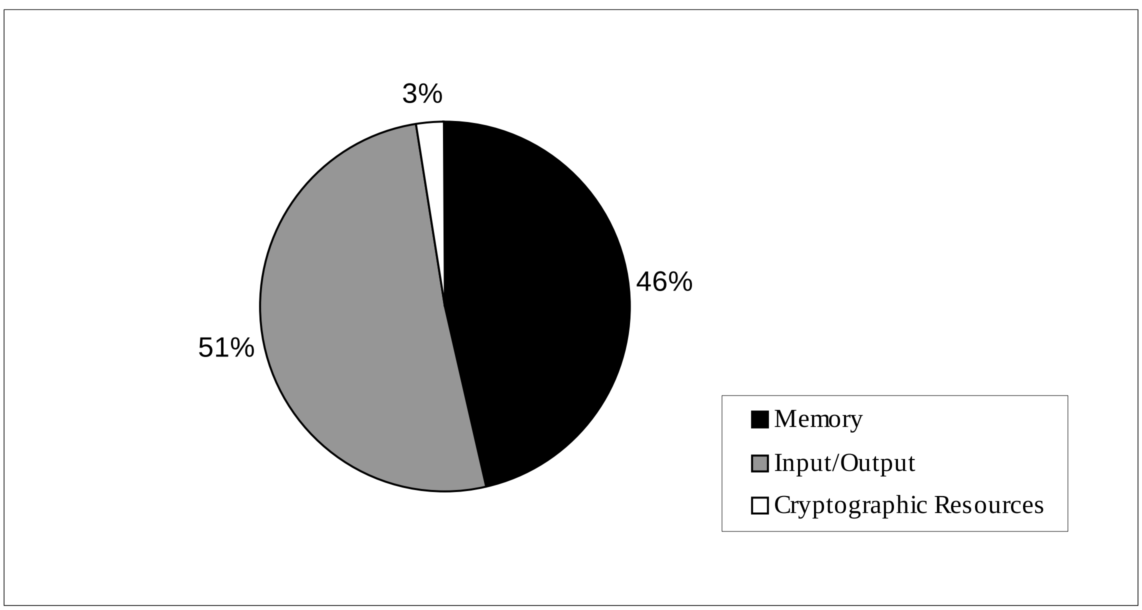


Figure 3.13 Distribution of classified vulnerabilities according to first level categories of the taxonomy

Figure 3.13 shows the distribution of classified *vulnerabilities*[4] across first level categories of the taxonomy. As expected, a majority of the vulnerabilities are classified under main memory (53) and input/output (58) categories, whereas only three are classified under cryptographic resources. Although it is well known that finding and exploiting vulnerabilities of cryptographic resources is difficult, the number of vulnerabilities classified under this category is still surprisingly small. This finding can be attributed to an accepted use of standardized cryptographic resources, and to the fact that agencies are reluctant to publicly report these vulnerabilities (because they are both dangerous and embarrassing).

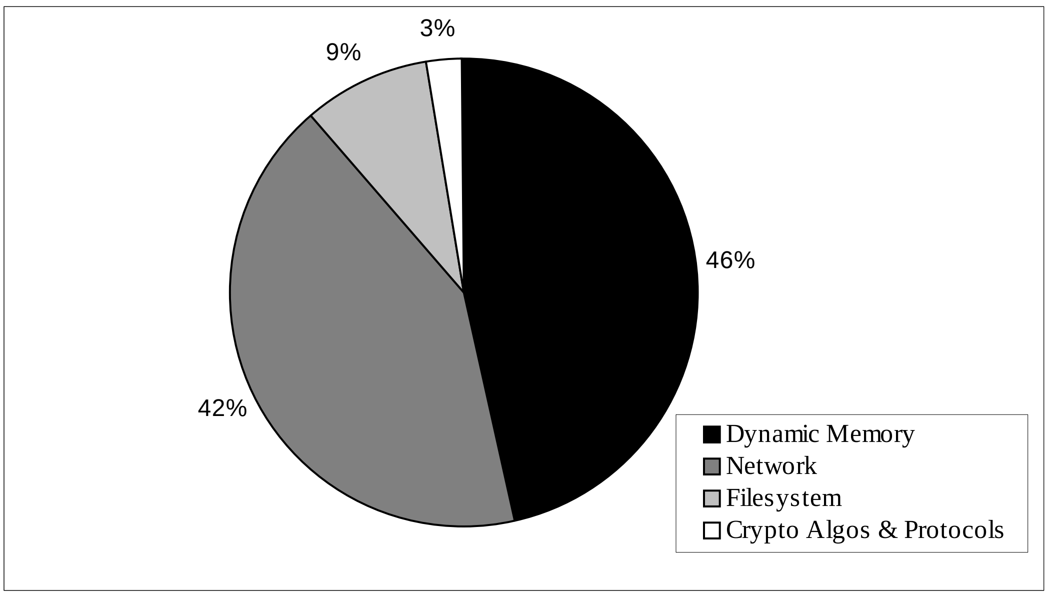


Figure 3.14 Distribution of classified vulnerabilities according to second level categories of the taxonomy

Figure 3.14 shows the distribution of vulnerabilities across second level categories of the taxonomy. Two categories, static memory and randomness resources are absent. The narrow scope of the data that static memory holds (static initialized and un-initialized global data) makes it difficult to exploit static memory vulnerabilities, and also limits the impact of their exploitation. Randomness resources are used primarily for cryptographic purposes and are also difficult to

[4] Singe advisory usually describes a single exploit. But a single exploit can consist of a number of vulnerabilities. Therefore, the number of vulnerabilities classified will differ from the total number of advisories.

exploit. However, because these factors only explain a *reduction* in the number of exploits using static memory and randomness, we were surprised that *no* exploits pertaining to static memory and randomness were reported.

Figure 3.14 indicates that a majority of vulnerabilities are classified, as expected, under the categories of dynamic memory and network interface. This stems from the fact that they are critical and heavily used resources, and have a significant number of violable constraints and assumptions associated with them. Unexpectedly, however a large number of these vulnerabilities are buffer and heap overflows. It seems that even in 2004, buffer and heap overflows were the biggest security problem, which is surprising in light of the increased publicity they have received over the years.

The data and analysis provided above illustrate both the power and utility of the taxonomy. If one considers those advisories that provided sufficient information, we were able to classify 63 out of 69 (91%) exploits (or 111 vulnerabilities) using the taxonomy of vulnerabilities. The remaining 6 advisories contained "vulnerabilities" that in actuality were errors that should never have made it to the software release. Even labeling these errors as vulnerabilities is questionable.

3.5 Characteristics of a Taxonomy

A taxonomy is not just a grouping of objects into classes. It should have certain characteristics to be considered a taxonomy. Howard [HOWA97], Lindqvist and Jonsson, [LIND97], Krsul [KRSU98], and Amoroso [AMOR94a] have all elaborated on the characteristics of security taxonomies. Lough [LOUG01] has complied a list of these characteristics. The majority of the characteristics these authors have identified are straightforward. For example, a security taxonomy should be accepted, unambiguous, appropriate and comprehensible, and employ terminology consistent with that used in the security field. Most taxonomies, including the one presented in this chapter, conform to these characteristics.

However, some of the more intricate and vital characteristics these authors have identified are missing in many of the reported taxonomies. Bishop [BISH96b] highlights two of these characteristics: (1) the categories of a taxonomy should be mutually exclusive and each object should be able to be uniquely classified into its own category, and (2) there should be a well-defined procedure for classifying objects into categories. Both of these characteristics are important, because they remove ambiguity and allow new objects to be classified, thereby affording continued usage and evolution of the taxonomy. The taxonomy presented in this chapter reflects both of these characteristics.

The authors noted above also identify several characteristics to which a vulnerability taxonomy need not or cannot conform. One is completeness. Because security is a dynamic field with new vulnerabilities being discovered continuously, and because new software paradigms are constantly emerging, new security vulnerabilities are always surfacing. Hence, no taxonomy can claim to classify all vulnerabilities. Although the taxonomy presented in this chapter categorizes a large number of vulnerabilities identified from various sources, there most probably exits constraints and assumptions that have not been listed in the taxonomy.

Objectivity is offered as yet another characteristic that is difficult to embody in a vulnerability taxonomy. Krsul [KRSU98] identifies this characteristic, which applies to the objects being classified, and defines it as, “the property (being used to classify the object) must be identified from the object known and not from the subject known.” In the case of vulnerability taxonomies, the above characteristic implies that one should not require a detailed knowledge of the fields of software systems and software security to identify the property used to classify vulnerabilities. Identifying properties of vulnerabilities, however, is still a difficult task because software systems themselves are complicated, and vulnerabilities residing within them are often subtle and intricately enmeshed within the software system. Consequently, most, if not all, vulnerability taxonomies lack the objectivity characteristic. The taxonomy presented in this chapter is no exception. Appropriately classifying vulnerabilities does require knowledge of the fields of software systems and software security.

In addition to the above characteristics, it is the author's opinion that other critical characteristics need to be linked to vulnerability taxonomies. They are discussed below.

- ***Goal of the taxonomy:*** A taxonomy should serve some purpose. For example, one goal might be to foster an understanding of the objects being classified, their characteristics, and their relationships. The purpose of the taxonomy presented in this chapter is to aid in the development of V&V strategies to assess software security. It conveys an ordered classification of vulnerabilities that provides a foundation upon which a systematic approach can be developed to assess software security.
- ***Domain of the taxonomy:*** A taxonomy should describe the domain to which it is applicable. That is, it should clearly specify the objects and the scope of the objects that it is classifying. The taxonomy presented in this chapter classifies security vulnerabilities by characterizing them as violable constraints and assumptions. Section 3.1 defines vulnerabilities and presents the theoretical model that permits their characterization as constraints and assumptions; Section 3.2.4 specifies the vulnerabilities classified by the taxonomy.
- ***Theoretical basis for the taxonomy:*** A taxonomy should be grounded in a theoretical foundation. The theoretical foundation should encompass the taxonomy and support a logical explanation of the classification scheme it uses. Section 3.1 presents the theoretical model of computing that serves as the foundation for both the taxonomy presented in this chapter and its classification scheme.
- ***Updates to the taxonomy:*** A taxonomy should be defined and structured in a manner that supports updates. This is necessary because over time new exploits will be developed that

incorporate yet to be identified vulnerabilities. Section 3.3 demonstrates that the taxonomy presented in this chapter permits updates to keep it current.

3.6 Related Work

Significant research has been directed towards creating security taxonomies. A large number of them focus on classifying different facets of a security exploit. In other words, some classify vulnerabilities, some methods of attack, and some security exploits. Additionally, they apply to different components of a computer system. For example, some apply to operating systems, some to application software, some to protocols, and some to any software system. This section presents a review of several notable taxonomies that represent important steps in attempts to understand and structure the area of software security. Appendix B presents a summery table of taxonomies not reviewed in this section.

3.6.1 Protection Analysis Project

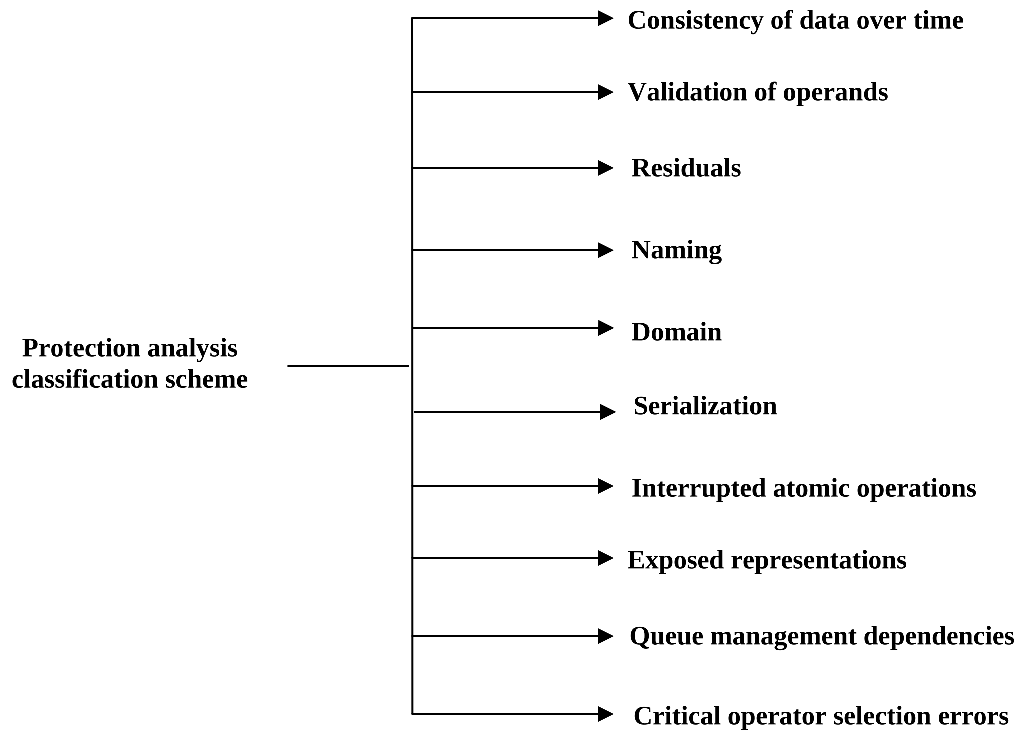


Figure 3.15 Protection analysis classification scheme

Bisbey and Hollingworth [BISB78] developed a classification scheme for *security errors* as a part of the Protection Analysis Project. Presented in the final report of the project, this scheme classifies operating system security errors into ten categories: consistency of data over time, validation of operands, residuals, naming, domain, serialization, interrupted atomic operations, exposed representations, queue management dependencies, and critical operator selection errors. However, the report does not elaborate on the definition of a security error. That is, it is not clear if a security error is a vulnerability, an attack methodology, or a combination of both. Furthermore, Bisbey and Hollingworth admit that some of the above categories can be included as subcategories of other categories; this introduces ambiguity when attempting to classify security errors. The report also lacks a process for classifying new errors. The classification scheme proposed by Bisbey and Hollingworth, however, does represent one of the first attempts at understanding security errors by grouping together errors with similar properties.

3.6.2 Research in Secure Operating Systems (RISOS) Project

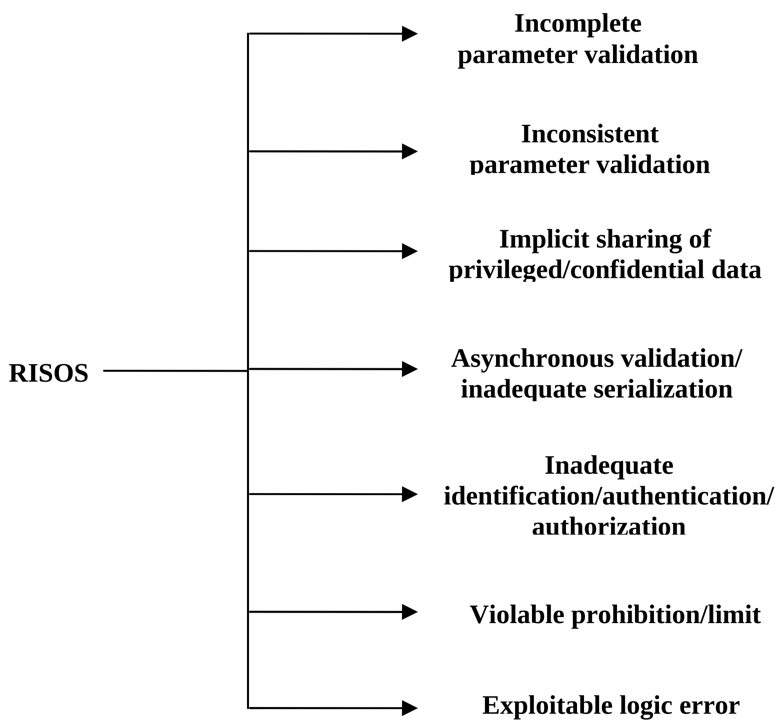


Figure 3.16 RIOS taxonomy of system integrity flaws

At Lawrence Livermore laboratory, Abbot *et al.* [ABBO76] developed a taxonomy as a part of the Research In Secure Operating Systems (RISOS) project. This taxonomy categorizes *operating system integrity flaws* into seven categories: incomplete parameter validation, inconsistent parameter validation, implicit sharing of privileged/confidential data, asynchronous validation / inadequate serialization, inadequate identification/authentication/authorization, violable prohibition/limit, and exploitable logic error. The authors define an operating system integrity flaw as something that causes the operating system to execute in a less reliable and less secure mode. Their definition, however, is somewhat amorphous and can be interpreted to include almost any error as an integrity flaw. As in the case of the Protection Analysis Project, the authors approach lacks a process for classifying integrity flaws. Furthermore, Bishop also states that the taxonomy is ambiguous, because a single flaw can be categorized into multiple categories [BISH96b].

3.6.3 Newman and Parker's Taxonomy of Misuse Techniques

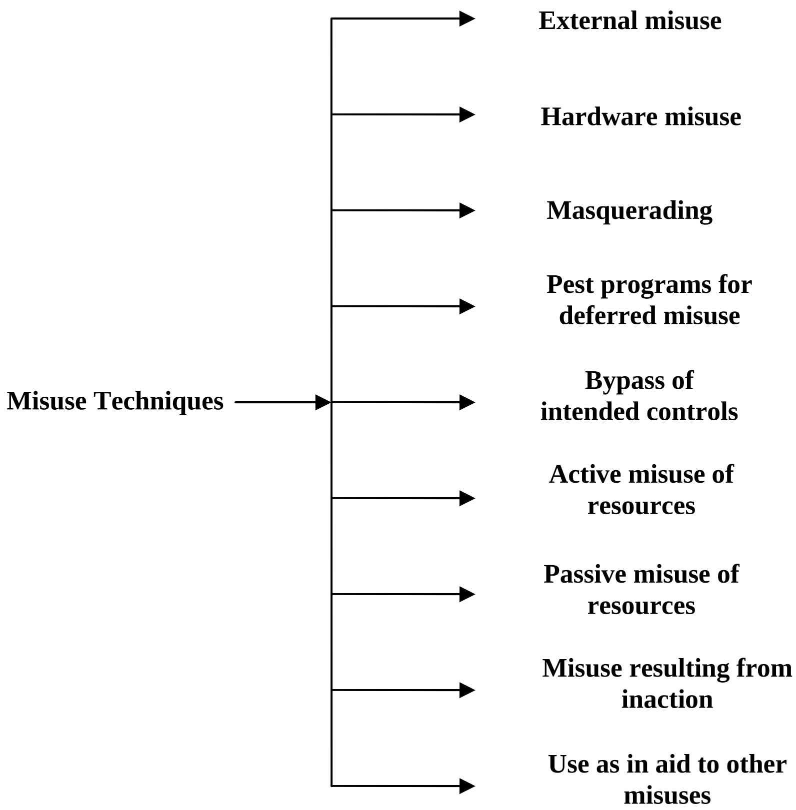


Figure 3.17 Newman and Parker's taxonomy of computer misuse techniques

In the late 1980s, Newman and Parker [NEUM89] published a taxonomy of *computer misuse techniques* in which they define nine distinct categories: external misuse, hardware misuse, masquerading, pest programs for deferred misuse, bypass of intended controls, active misuse of resources, passive misuse of resources, misuse resulting from inaction, and use as in aid to other misuses. Lindqvist and Jonsson [LIND97] refine that taxonomy by dividing three of the nine categories into subcategories. This extended taxonomy offers a fairly exhaustive classification of methods of attack. However, Lindqvist and Jonsson acknowledge that classifying a particular attack into a category is still often a matter of interpretation. Additionally, neither the original nor the extended taxonomy provide a process for classifying new attack methods. The taxonomy presented in this chapter classifies vulnerabilities (as opposed to computer misuse techniques), and as such, is fundamentally different from both the original and extended taxonomies.

3.6.4 VERDICT

In 2001, Lough [LOUG01] presented VERDICT, an acronym for Validation Exposure Randomness Deallocation Improper Conditions Taxonomy. Validation, exposure, randomness, and deallocation constitute the four categories of his taxonomy. This taxonomy differs from the others presented in this section in that it classifies the *causes of security errors*. However, Lough does not elaborate on what constitutes the cause of a security error or the relationship between an exploit, a security error, and the cause of a security error; nor does he present a process for deriving the cause of a security error from an exploit and then classifying it. The importance of his work, however, is that his taxonomy focuses on the causes of security errors rather than vulnerabilities.

3.6.5 Aslam's Taxonomy

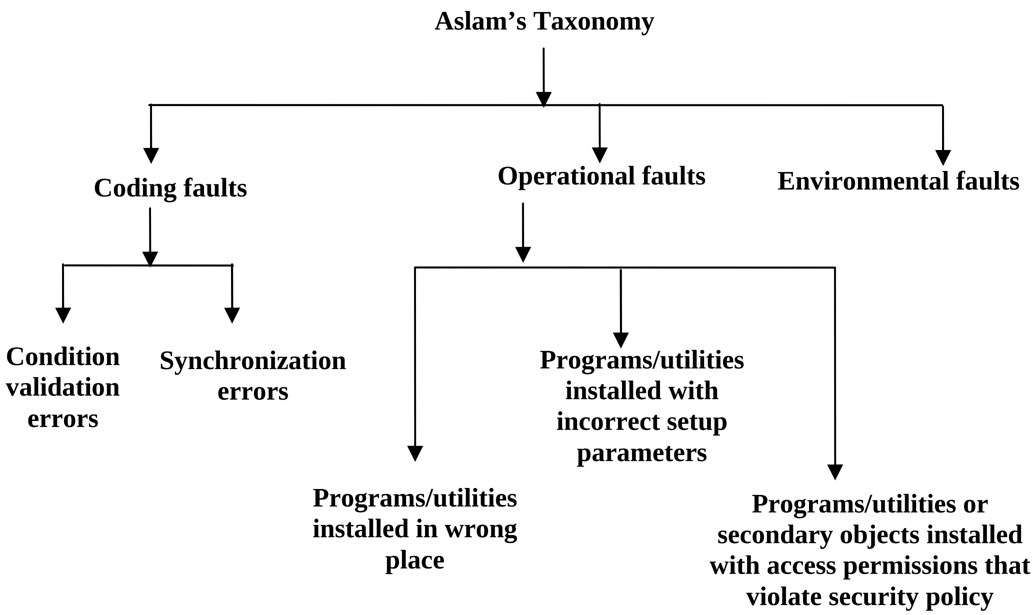


Figure 3.18 Aslam's taxonomy of security faults

Aslam [TAIM95] presented a taxonomy of *security faults* in 1995. This taxonomy, which was developed to organize information being stored in a vulnerability database, consists of three top-level categories: operational faults (configuration errors), coding faults, and environment faults. Aslam divides the coding faults category into condition validation errors and synchronization errors subcategories, which he then further divides. He further divides the operational faults category into three subcategories: (1) programs/utilities installed in the wrong place, (2) programs/utilities installed with incorrect setup parameters, and (3) programs/utilities or secondary objects installed with access permissions that violate security policy. Aslam also presents the selection criteria that are to be used when classifying vulnerabilities. However, Bishop [BISH96b] shows that this taxonomy is also ambiguous, because it allows a single vulnerability to be classified into multiple categories.

3.6.6 Krsul's Extension of Aslam's Taxonomy

Environmental faults

↓

Environmental objects

↓

Attributes

↓

Assumptions

Figure 3.19 Structure of Krul's taxonomy

Krsul [KRSU98] extends Aslam's work by further decomposing the environmental faults category. The top-level categories of his taxonomy consist of environmental objects, each of which is defined as an entity that contains or receives information, and which has a unique name and set of operations that can be performed on it. For example, "running program" is an object. Associated with an object are attributes. Those attributes are defined as data components of the object. For example, an environmental variable is an attribute of the object: running program. The attributes form the next level categories in the taxonomy and can be further refined as required. Each of the attributes has assumptions associated with it that the programmers make. Krsul asserts that these assumptions are responsible for environmental vulnerabilities.

Although Krsul's taxonomy is the most detailed of those included in this section, it still has some shortcomings. First, it is difficult to distinguish between objects and their attributes because of the latitude in interpretation permitted by the taxonomy. For example, an environment variable can be considered as an attribute of the executing program (an object), or it can be considered as an object by itself – this is a source of ambiguity in the taxonomy. Furthermore, Krsul fails to elaborate on how assumptions lead to vulnerabilities. Nonetheless, the significance of Krsul's taxonomy is that it provides substantial insight into what constitutes vulnerabilities by focusing attention on assumptions that are responsible for them.

Summary

This chapter presents a taxonomy of vulnerabilities. The vulnerabilities classified by the taxonomy pertain to any software process executing above the level of the operating system. Those vulnerabilities are characterized in the form of violable constraints and assumptions. It is the constraints and assumptions that are exploited to compromise software processes. The taxonomy is grounded in a theoretical model of computing. The model provides not only the foundation for characterizing vulnerabilities, but also supports a taxonomic, resource-based classification scheme. At the top level, the taxonomy has three categories: main memory, input/output, and cryptographic resources. These three categories are further divided into subcategories, with the constraints and assumptions being the last set of entries. This chapter also presents the process of classifying newly identified vulnerabilities. The utility and power of the taxonomy is confirmed through the classification of vulnerabilities present in advisories issued by CERT from January to June 2004. Lastly, this chapter presents a review of some of the more notable taxonomies defined prior to this research effort.

Chapter 4

Object Model

The object model is an organized collection of potentially vulnerable process objects that can be present in a software process. A process object is defined as an entity that satisfies the following two conditions: (1) it holds data, and (2) there exists a set of operations, such as create, remove, open, close, read, write, and so forth, that can be performed on it. Examples of process objects include files, directories and buffers. The object model permits the identification of potentially vulnerable process objects present in a software process.

Figure 4.1 illustrates the structure of the object model. As is apparent, the object model borrows its classification scheme from the taxonomy of vulnerabilities. The same classification scheme makes it possible to draw associations between constraints and assumptions in the taxonomy and the process objects in the object model. The three top-level categories of the object model are: (1) main memory, (2) I/O, and (3) cryptographic resources. Each of these categories is divided into subcategories. The main memory is divided into dynamic memory and static memory; I/O is divided into network interface and filesystem; and the cryptographic resources category is divided into randomness resources and cryptographic algorithms and protocols. Associated with each subcategory is a list of potentially vulnerable process objects.

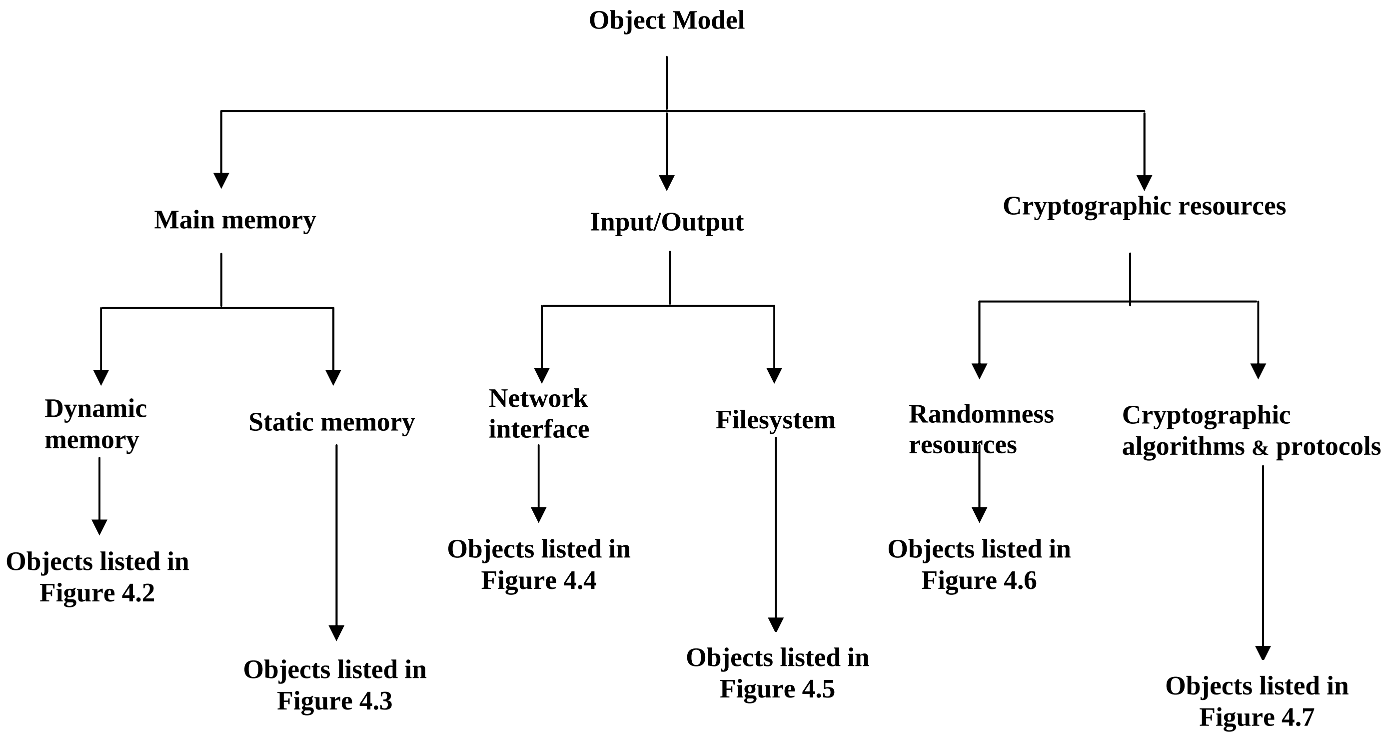


Figure 4.1 The object model

The process objects currently listed in the object model are derived from the constraints and assumptions listed in the taxonomy. Each constraint and assumption is analyzed to identify the object or objects to which it applies. This analysis requires the identification of the entity that fits the definition of a process object, and on which a constraint is applied or about whose usage the software process is making the assumption. To further illustrate the concept, consider the example of a constraint classified in the dynamic memory subcategory of the taxonomy: *Data accepted as input by the process and assigned to a buffer must occupy and modify only specific locations allocated to the buffer.* An analysis of the constraint reveals that the entity in question is a buffer – because the constraint is applied to it, and it fits the definition of the process object.

The object model structures and simplifies testing a software process for the presence of vulnerabilities by dividing the testing process into multiple, less complex steps. Therefore, instead of one complicated step of identifying the constraints and assumptions that apply to the software process, there are two simpler steps: (1) using the object model to identify potentially vulnerable process objects present in the software process, and (2) using the taxonomy to identify violable constraints and assumptions associated with these objects.

The remainder of this chapter presents the process objects associated with each subcategory of the object model. Because the subcategories themselves are same as those of the taxonomy of vulnerabilities, please refer to Chapter 3 for a more comprehensive description of each subcategory.

4.1 Dynamic Memory

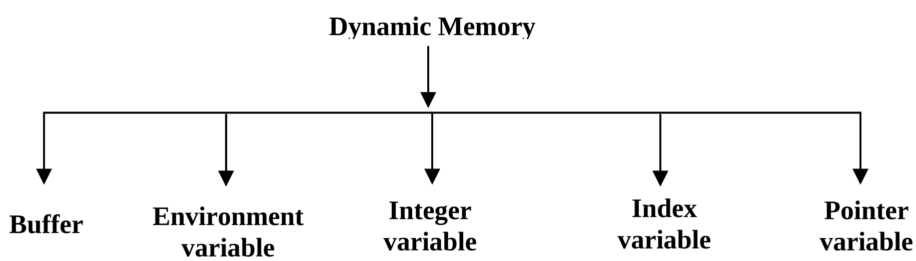


Figure 4.2 Objects associated with dynamic memory

The dynamic memory includes five process objects (Figure 4.2), which are given below along with a description of each:

- *Buffer:* A buffer refers to a contiguous block of memory, whose length varies from 1 to n, where n is a positive integer. For example, an array is a buffer.
- *Environment variable:* An environment variable is a dynamic value that affects how a running process behaves. There are a number of such variables with different names and values [WIKI]. For example, PATH is an environment variable provided by the UNIX operating system.
- *Integer variable:* An integer variable refers to a variable that takes only integral values (positive, negative and zero). For example, in the C language variables declared as *int* are integer variables.

- *Index variable:* An index variable refers to a variable that is used to identify the position of elements in a buffer.
- *Pointer variable:* A pointer variable refers to a variable whose value points to another variable stored elsewhere in the computer memory [WIKI]. For example, in the C language variables declared as *int* * are pointer variables that point to an integer variable.

4.2 Static Memory

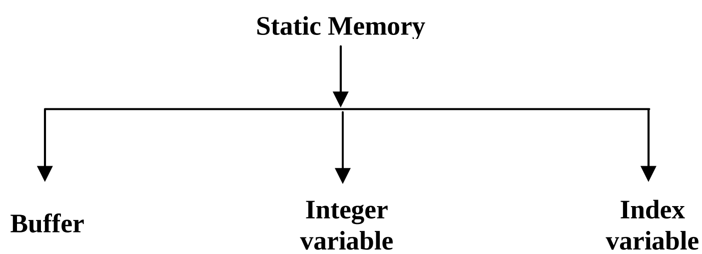


Figure 4.3 Objects associated with static memory

The static memory includes three process objects (Figure 4.3): buffer, integer variable, and index variable. All of these objects have the same definitions as their counterparts in the dynamic memory category. The difference between the objects in the static memory and dynamic memory lies in the usage of each. A software process uses dynamic memory to store variables and other process related data while it is in execution, whereas it uses the static memory to store initialized and un-initialized global variables whose size is fixed before the process begins execution and does not change as execution proceeds.

4.3 Filesystem

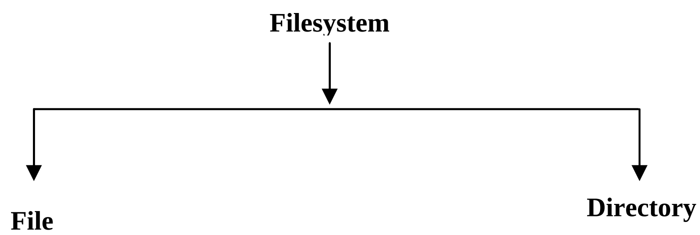


Figure 4.4 Objects associated with filesystem

Filesystem includes two process objects (Figure 4.4), which are given below along with their descriptions:

- *File:* A file is a stream of bits stored as a single unit on the filesystem [WIKI]. Typically, a name is associated with a file so that it can be referenced to at a later time.
- *Directory:* A directory is an entity that contains a group of files and other directories. It is used to organize files by keeping related files together [WIKI].

4.4 Network Interface

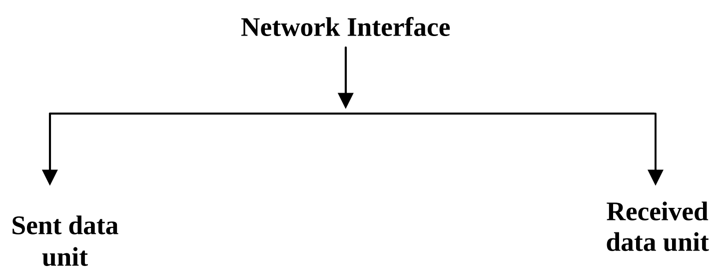


Figure 4.5 Objects associated with Network Interface.

Network interface includes two process objects (Figure 4.5); their descriptions are provided below:

- *Sent data unit:* A sent data unit refers to the unit of information sent by the software process using the network interface. It is important to note that the sent data unit does not refer to a data packet. The software process is not aware of how data is being sent through the network; it simply uses system calls to send information through the network interface. A sent data unit refers to this information.

- *Received data unit:* A received data unit refers to the unit of information received by the software process from the network interface. Again, a received data unit does not imply data packets received through the network. It denotes the information that the software process receives using the network interface.

4.5 Randomness Resources

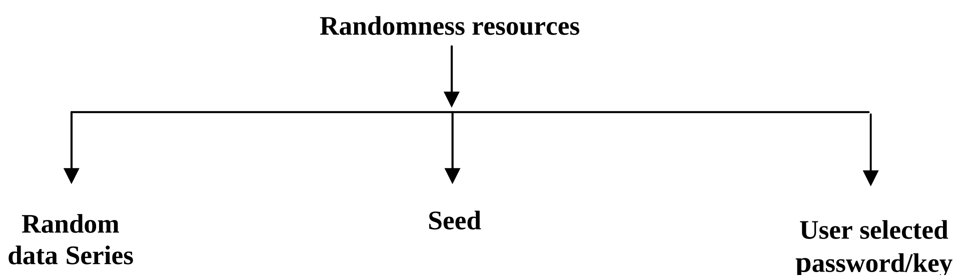


Figure 4.6 Objects associated with randomness resources.

Randomness resources include three process objects (Figure 4.6), which are given below along with a description of each:

- *Random data series:* A random data series is a series of numbers that conforms to the following two conditions: (1) numbers in the series are uniformly distributed over a set of numbers and (2) the probability of predicting the next number in the series is zero.
- *Seed:* The seed refers to a number used as input to the Pseudo Random Number Generator (PRNG). The seed itself is generated through a random process.
- *User selected password/key:* A user selected password or key consists of secret data used to control the operation of a number of cryptographic algorithms and protocols. The password or key is generally used in authentication protocols to prove the identity of an entity. In other words, an entity proves its identity to the authentication

protocol by providing it the secret password or key. If the authentication protocol successfully matches the password or key with its stored password or key, the entity is granted access to the services or resources being guarded by the protocol. Ideally, a user selected password or key should be random.

4.6 Cryptographic Algorithms and Protocols

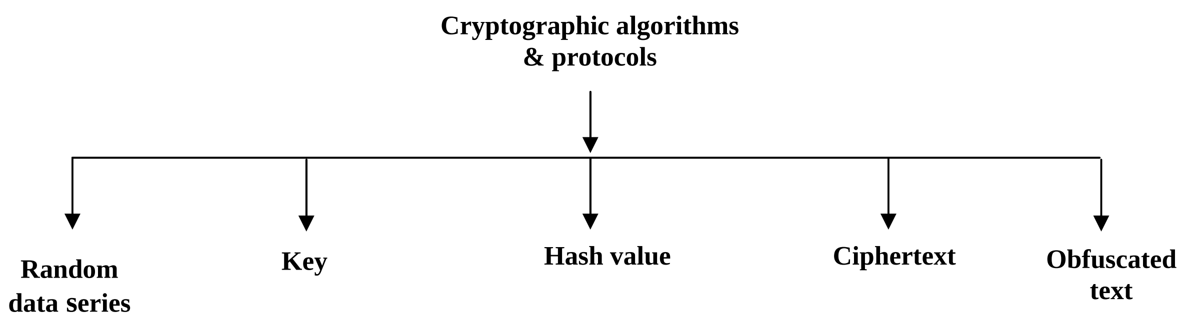


Figure 4.7 Objects associated with cryptographic algorithms and protocols

Cryptographic algorithms and protocols include five process objects (Figure 3.10):

- *Random data series:* A random data series is a series of numbers that conforms to the following two conditions: (1) numbers in the series are uniformly distributed over a set of numbers and (2) the probability of predicting the next number in the series is zero.
- *Key:* A key is secret data used to control the operation of a number of cryptographic algorithms and protocols. For example, in encryption a key is used to convert plaintext to ciphertext and vice versa. Other cryptographic algorithms and protocols, such as digital signatures, key hash functions and authentication protocols [WIKI] also use keys.
- *Hash value:* Hash value refers to the output of a hash function. A hash function is one that converts input from a (typically) large domain into an output in (typically) a smaller range. Examples of hash functions include algorithms such as MD5 and

SHA1 [WIKI]. Hash values are used in a number of security applications, such as digital signature, certificates and so forth.

- *Ciphertext:* Ciphertext refers to the output of the process of encryption. Encryption is defined as a process of obscuring information and meaning. There exist a number of encryption algorithms such as DES, RSA and AES. Typically, an encryption algorithm takes plain text and encrypts it using a key to produce ciphertext. The ciphertext appears meaningless without converting it back to plain text, which requires access to the key. Encryption is used to ensure data confidentiality.
- *Obfuscated text:* Obfuscated text refers to the text that has been changed to make it difficult to perceive or understand by employing such techniques as using obscure formats, rearranging data and scrambling the symbols.

Summary

This chapter presents the object model, which is an organized collection of potentially vulnerable process objects that can be present in a software process. The object model borrows its classification scheme from the taxonomy of vulnerabilities. Thus, it has the same categories and subcategories as the taxonomy; the process objects are classified into these subcategories. Furthermore, the same classification scheme makes it possible to relate process objects in the object model to the constraints and assumptions in the taxonomy. The objects currently listed in the object model are derived by analyzing constraints and assumptions listed in the taxonomy.

Chapter 5

Verification and Validation Strategies

This chapter presents the third and final component of the framework, Verification and Validation (V&V) strategies. The V&V strategies component combines the information from the object model and the taxonomy, and provides base strategies to test a software process for the presence of vulnerabilities. The base strategies focus on verification or validation activities that provide a guided approach to test if a constraint or an assumption associated with a process object can be violated. Each base strategy is composed of three elements:

- *Target:* The target specifies the process object or objects present in the software process to which the base strategy is to be applied. Herein, the object model provides the process objects that serve as targets for V&V.
- *Goal:* The goal specifies the objective of the base strategy. It is associated with the target of the base strategy. The taxonomy provides the constraints and assumptions, which serve as goals of the base strategies. That is, the goal of a base strategy is to test if the software process permits violation of the identified constraint or assumption.
- *Method:* The method specifies the approach to achieve the goal associated with the target of the base strategy. In other words, the method specifies the approach to test if a

constraint or an assumption associated with a process object can be violated. In particular, the method specifies the approach for exposing the vulnerability, not exploiting it.

The base strategy is generally applicable and can be applied to any software process executing above the level of the operating system. These base strategies are used to derive V&V test strategies that are specific to the software process being tested. Deriving a test strategy from a base strategy involves refining and adapting the base strategy for the specific software process. These test strategies are implemented by deriving test cases, which are used to test if a software process is vulnerable. This chapter provides several examples of deriving test strategies from base strategies and implementing these test strategies by deriving test cases.

We envision base strategies that provide two levels of assessment, depending on the amount of information available about the development of the software process:

1. *Validate Only:* This level of assessment applies when evaluators only have access to the process executable. That is, evaluators are only familiar with the process' functionality; they do not have access to any process documentation. Therefore, in this case they can only use base strategies that can be applied through the execution of the software process.
2. *End game V&V:* This level of assessment applies when evaluators have access to both the process executable and its documentation. Process documentation includes artifacts, such as source code and design documentation. Clearly, evaluators can use base strategies that require execution of the software process. Additionally, they can also employ base strategies that take advantage of information in development artifacts.

The level of assessment to be used for the software process is situational and dependent on the following factors: (1) the software process in question, (2) available development documentation, and (3) availability of tools that can assist in the execution of base strategies. For example, using commonly available tools such as filemon [FMON], most of the base strategies associated with the filesystem subcategory can be applied to almost any software process just by executing the process. However, it is difficult to apply base strategies associated with the cryptographic algorithms and protocols subcategory without access to the development documentation.

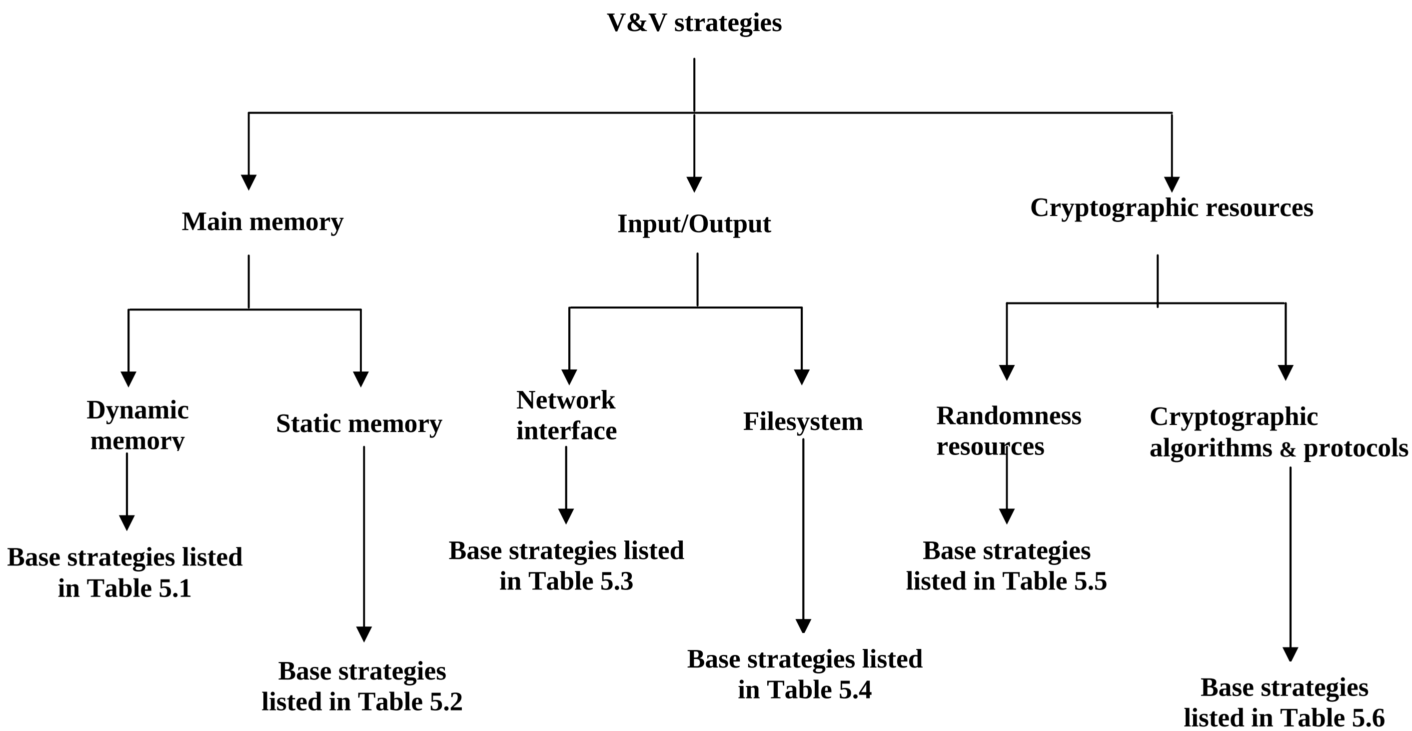


Figure 5.1 The V&V strategies component.

Figure 5.1 illustrates the structure of the V&V strategies component. As is apparent, this component borrows its classification scheme from the taxonomy. Hence, the three top-level categories are: (1) main memory, (2) I/O, and (3) cryptographic resources. Each of these categories is divided into subcategories. The main memory is divided into dynamic memory and static memory; I/O is divided into network interface and filesystem; and cryptographic resources is divided into randomness resources and cryptographic algorithms and protocols. The base strategies are classified within these subcategories.

As is the case with the object model, the V&V strategies component shares its classification scheme with the taxonomy, which makes it possible to draw associations between base strategies, process objects, and constraints and assumptions. Furthermore, there exists a one to one relationship between the base strategies and the constraints and assumptions. In other words, there is a base strategy associated with each constraint and assumption listed in the taxonomy. Moreover, the base strategies are listed according to their corresponding constraint or assumption in the taxonomy. For example, the first base strategy in the dynamic memory subcategory corresponds to the first constraint or assumption in the dynamic memory subcategory of the taxonomy, the second strategy to second constraint or assumption, and so forth.

The remainder of this chapter presents the base strategies associated the subcategories of the V&V strategies component. Because the subcategories themselves are same as those of the taxonomy of vulnerabilities, please refer to Chapter 3 for a description of each subcategory. In addition to the base strategies, for each subcategory a detailed example of using a base strategy to derive a test strategy is provided.

5.1 Dynamic Memory

This section presents the base strategies associated with the dynamic memory in Table 5.1 and provides an example of deriving a test strategy from a base strategy.

Table 5.1 Base strategies associated with dynamic memory.

No.	Base strategy
1.	**Goal:** To test if the software process allows violation of the constraint: d*ata accepted as input by the process and assigned to a buffer must occupy and modify only specific locations allocated to the buffer.* **Target:** Buffer **Method:** Attempt to store data larger than the size of the buffer into the fixed length buffer. The constraint is considered violated if the process does not restrict the size of data and copies it into the buffer.
2.	**Goal:** To test if the software process allows violation of the assumption: *the process will not interpret data present on the dynamic memory as executable code.* **Target:** Buffer **Method:** Attempt to overwrite process variables, such as return addresses and exception pointers. Because these variables are responsible for redirecting the instruction pointer to the appropriate

	instruction, evaluators can overwrite them to point to an address of their choice. The assumption is considered violated if evaluators are able to redirect the instruction pointer to an address of their choice. Typically, an unbounded buffer on the program stack is used to overwrite these variables.
3.	**Goal:** To test if the software process allows violation of the assumption: *environment variables being used by the process have expected format and values.* **Target:** Environment variables **Method:** Change the format and values of environment variables before the process begins execution. The assumption is considered violated if the process uses these modified environment variables.
4.	**Goal:** To test if the software process allows violation of the assumption: *the process will be provided with the dynamic memory that it requests.* **Target:** All data stored by the process on the dynamic memory **Method:** Restrict the amount of memory available to the software process by running it in a controlled environment using tools such as holodeck [WHIT03]. These tools allow control of the amount of memory available to the software process. The assumption is considered violated if the process terminates abnormally or hangs indefinitely.
5.	**Goal:** To test if the software process allows violation of the assumption: *data present on the dynamic memory cannot be observed while the process is in execution.* **Target:** All data stored by the process on the dynamic memory **Method:** Execute the software process in a controlled environment, such as a debugger, which allows evaluators to view the contents of the dynamic memory. The assumption is considered violated if evaluators can access any privileged data that the process has stored in the dynamic memory.
6.	**Goal:** To test if the software process allows violation of the assumption: d*ata owned by the process and stored on the dynamic memory cannot be accessed after the process frees the memory.* **Target:** All data stored by the process on the dynamic memory **Method:** Attempt to read the contents of the memory allocated to the software process after it terminates. Since the memory being used by the process is not erased after the process frees it, evaluators can directly access the physical memory and attempt to read data left over by the process. The assumption is considered violated if evaluators can read data left over by the software process.
7.	**Goal:** To test if the software process allows violation of the assumption: *a pointer variable being used by the process references a legal memory location.* **Target:** Pointer variable **Method:** Attempt to change the memory location to which the pointer points. The assumption is considered violated if evaluators are able to modify the value of the pointer variable to refer to memory

	locations outside the process space or to wrong variables.
8.	**Goal:** To test if the software process allows violation of the assumption: *a memory pointer returned by the underlying operating system does not point to zero bytes of memory.* **Target:** Pointer variable **Method:** Request zero bytes of memory from the operating system. Some operating systems do return pointers that point to zero bytes of memory. The assumption is considered violated if the process uses this pointer to access memory.
9.	**Goal:** To test if the software process allows violation of the assumption: *a pointer variable being used by the process cannot reference itself.* **Target:** Pointer variable **Method:** Attempt to change a pointer to point to itself. The assumption is considered violated if evaluators are successful in redirecting the pointer to point to itself.
10.	**Goal:** To test if the software process allows violation of the constraint: *data accepted by the process must not be interpreted as a format string by the I/O routines.* **Target:** Buffer **Method:** Provide the software process with a format string as input. The constraint is considered violated if the process accepts the string and outputs the contents of the program stack.
11.	**Goal:** To test if the software process allows violation of the assumption: *the value of an integer variable/expression (signed & unsigned) accepted/calculated by the process cannot be greater (less) than the maximum (minimum) value that can be stored in the integer variable.* **Target:** Integer variable **Method:** Store values that are larger (smaller) than the maximum (minimum) value that can be stored in an integer variable. The assumption is considered violated if the process attempts to store these values, and in doing so, stores overflow (underflow) values, which are different from the intended values.
12.	**Goal:** To test if the software process allows violation of the constraint: *an integer variable/expression used by the process as the index to a buffer must only hold values that allow it access to the memory locations assigned to the buffer.* **Target:** Index variable **Method:** Use index values that are larger than the size of the buffer. The constraint is considered violated if the process uses these index values.
13.	**Goal:** To test if the software process allows violation of the constraint: *an integer variable/expression used by the process to indicate length/quantity of any object must not hold negative values.* **Target:** Integer variable

	Method: Use negative values to indicate the length of the objects. The constraint is considered violated if the process uses these negative values.

An Example

Base strategy 12 in Table 5.1 is used to provide a detailed example of deriving a test strategy to determine if a software process allows the violation of a constraint associated with the dynamic memory. The system state consists of a software process that accepts from a user two integer values: a position to store data and the value of the data itself. The software process stores the data in an integer array allocated on the program stack and uses the position value as an index to the array.

The information provided by the base strategy is refined and adapted to derive a test strategy to assess if the software process is vulnerable. For this specific software process, the test strategy involves executing the software process and providing it index values that range from one more than the size of the buffer to several times the size of the buffer and the data values to be placed in the buffer. If the software process stores and uses any of these index values, the constraint is considered violated and the process vulnerable. This test strategy is implemented by deriving multiple test cases, where for each test the index value is varied according to the range specified in the test strategy.

5.2 Static Memory

Table 5.2 presents the base strategies associated with the static memory. An example of deriving a test strategy from a base strategy follows.

Table 5.2 Base strategies associated with static memory.

No.	Base strategies
1.	**Goal:** To test if the software process allows violation of the constraint: d*ata accepted as input by the process and assigned to a buffer must occupy and modify only specific locations allocated to the buffer on the static memory.* **Target:** Buffer **Method:** Attempt to store data larger than the size of the buffer into a fixed length buffer. The constraint is considered violated if the process does not restrict the size of the data and copies it into the buffer.
2.	**Goal:** To test if the software process allows violation of the assumption: *data held on the static memory cannot be observed while the process is in execution.* **Target:** All data stored by the process on the static memory **Method:** Execute the software process in a controlled environment, such as a debugger, which allows evaluators to view the contents of the static memory. The assumption is considered violated if evaluators can access any privileged data that the process has stored in the static memory.

An Example

Base strategy 1 in Table 5.2 to used to provide a detailed example of deriving a test strategy to determine if a software process allows violation of an assumption associated with the static memory. The system state consists of a software process that accepts a string as input from the users and copies it into a fixed length buffer. The software process declares this buffer along with several other variables as global variables, thereby implying that all of these variables and the buffer reside in the static memory.

The information provided by the base strategy is refined and adapted to derive a test strategy to assess if the software process is vulnerable. For this specific software process, the test strategy involves executing the software process and providing to it input strings whose sizes range from one more than the size of the buffer to several times the size of the buffer. If the software process stores any of these strings (in total) in its buffer, the assumption is considered violated and the process vulnerable. This test strategy is implemented by deriving multiple test cases, where for each test case the length of the string is varied according to the range specified in the test strategy.

5.3 Filesystem

This section presents the base strategies associated with the filesystem. Those strategies are listed in Table 5.3. An example of deriving a test strategy from a base strategy follows.

Table 5.3 Base strategies associated with filesystem.

No.	Base strategies
1.	**Goal:** To test if the software process allows violation of the constraint: *access permissions assigned to newly created files/directories are such that only the required principals have access to them.* **Target:** File, directory **Method:** Analyze the access permissions of all newly created files or directories. This constraint is considered violated if any principal other than the required principals has access to these files or directories.
2.	**Goal:** To test if the software process allows violation of the constraint: *access permissions of the files/directories being used by the process are such that only the required principals have access to them.* **Target:** File, directory **Method:** Analyze access permissions of all files and directories being used by the software process. This constraint is considered violated if any principal other than the required principals has access to these files and directories.
3.	**Goal:** To test if the software process allows violation of the assumption: *a file being created by the process does not have the same name as an already existing file.* **Target:** File **Method:** Create a file that has the same name as a file being created by the software process and place it in the same directory where the software process is going to place its file. The assumption is considered violated if the software process uses the file or terminates abnormally.
4.	**Goal:** To test if the software process allows violation of the assumption: *a filename (including path) being used by the process is not a link that points to another file for which the user, executing the process, does not have the required access permissions.* **Target:** File **Method:** Provide to the process a file that is a link to another file for which the evaluator, executing the process, does not have the required access permissions. The assumption is considered violated if the software process accepts this file, follows the link, and uses the file pointed to by the

	link.
5.	**Goal:** To test if the software process allows violation of the assumption: *a file created/populated by a principal other than the process and being used by the process will have expected format and data.* **Target:** File **Method:** Provide to the software process files containing corrupt data. The assumption is considered violated if the process accepts and uses these files.
6.	**Goal:** To test if the software process allows violation of the constraint: *a file being used by the process cannot be observed /modified/replaced while the process is in execution.* **Target:** File **Method:** Execute the software process in a controlled environment, such as holodeck [WHIT03]. Use this environment to identify, observe, modify, and replace the files that the process is using. This constraint is considered violated if the process allows evaluators to observe the file contents or uses the modified files or uses the replaced files or all three.
7.	**Goal:** To test if the software process allows violation of the assumption: a *file/directory being used by the process and stored on the filesystem (information used by the process over multiple runs) cannot be observed/modified/replaced in-between these runs.* **Target:** File, directory **Method:** Identify persistent files (files permanently stored on the file system) being used by the software process with the aid of such tools as filemon and holodeck. Attempt to observe and modify the contents of these files while the process is not in execution. This assumption is considered violated if evaluators can access privileged data that the process has stored in these files or if the process uses the modified files or both.
8.	**Goal:** To test if the software process allows violation of the constraint: d*ata held by files owned/used by the process must not be accessible after the process deletes them.* **Target:** File **Method:** Attempt to access data held in files used by the software process after the process has deleted them by directly accessing the physical storage. Since the operating system does not erase data after deleting the files, evaluators can access this data by directly reading the physical storage used by the filesystem. This constraint is considered violated if evaluators can access data held by these files.
9.	**Goal:** To test if the software process allows violation of the constraint: *the process must be provided with the filesystem space that it requests.*

	Target: All data stored by the process on the filesystem. **Method:** Restrict the amount of filesystem space available to the software process by running it in a controlled environment using tools, such as holodeck [WHIT03]. These tools allow control of the amount of filesystem space available to the software process. This constraint is considered violated if the process displays abnormal behavior, such as terminating without any warning or hanging indefinitely.
10.	**Goal:** To test if the software process allows violation of the assumption: *a file having proprietary or obscure format cannot be understood or modified.* **Target:** File **Method:** Attempt to reverse engineer and modify the file that has obscure or proprietary format. The constraint is considered violated if evaluators gain access to any privileged data held in the file or if the process uses the modified file or both.

An example

Base strategy 4 in Table 5.3 is used here to provide a detailed example of deriving a test strategy to determine if a software process allows violation of an assumption associated with the filesystem. The system state consists of a software process executing with setuid permissions. In other words, although a user is executing the process, it has effective root user privileges, which enables the process to access any resource of the computer system. The software process, in course of its execution, asks the user to provide it with the name of a file to which it has to write data.

The information provided by the base strategy is refined and adapted to derive a test strategy to assess if the software process is vulnerable. For this specific process, the test strategy involves executing the software process and providing it with a filename that is a link to another file for which the evaluators executing the process do not have the required access permissions. The type of the link is varied according to the underlying operating system. For example, on UNIX based operating systems, two types of links are used, hard links and soft links. Furthermore, the access permissions of the file pointed to by the link are also varied from the evaluators having partial access permissions to having no access permissions. If the process accepts the filename, follows the link, and writes data to the file pointed to by the link, the assumption is considered violated and the process vulnerable. This test strategy is implemented by deriving multiple test cases. Each test case varies the type of the link and access permissions of the file pointed to by the link.

5.4 Network Interface

This section presents the base strategies associated with the network interface (shown in Table 5.4). It also provides an example of deriving a test strategy from a network interface base strategy.

Table 5.4 Base strategies associated with network interface.

No.	Base strategies
1.	**Goal:** To test if the software process allows violation of the assumption: *the data received by the software process through the network interface is neither read nor modified by anyone other than the intended recipient.* **Target:** Received data unit **Method:** Intercept data being sent to the software process and attempt to read and modify it. The assumption is considered violated if evaluators are able to understand the received data, or if the process uses the modified data.
2.	**Goal:** To test if the software process allows violation of the assumption: *the data received by the software process through the network interface is from a legitimate client or peer or server and has expected format and length.* **Target:** Received data unit **Method:** Masquerade as a legitimate client, peer or server and send corrupt data to the software process. The assumption is considered violated if the process accepts and uses this data.
3.	**Goal:** To test if the software process allows violation of the assumption: *the data sent by the software process via the network interface will not be read/modified before it reaches its destination.* **Target:** Sent data unit **Method:** Intercept data being sent by the software process and attempt to read and modify it. The assumption is considered violated if evaluators are able to understand this data or if the receiving

	process uses the modified data or both.
4.	**Goal:** To test if the software process allows violation of the assumption: *the software process will be able to utilize the network interface to send and receive data.* **Target:** All data sent and received by the process using the network interface **Method:** Restrict the network access of the software process by running it in a controlled environment using tools such as holodeck [WHIT03], which allow control of the availability of the network interface. This assumption is considered violated if the process displays abnormal behavior, such as terminating without any warning or hanging indefinitely.
5.	**Goal:** To test if the software process allows violation of the assumption: *the byte order of numerical data accepted from the network interface is same as that of the host machine.* **Target:** Received data unit **Method:** Send to the software process numerical data that is in the network format. The assumption is considered violated if the process accepts this data and does not change its format (in case the host and network formats are different), thereby storing wrong numerical values.

An Example

Base strategy 2 in Table 5.4 is used to provide a detailed example of deriving a test strategy to determine if a software process allows violation of an assumption associated with the network interface. The system state consists of a server process that accepts input from various client processes running on other computer systems, which it copies into a fixed length buffer for further processing.

The information provided by the base strategy is refined and adapted to derive a test strategy to assess if the software process is vulnerable. For this specific process, the test strategy involves executing the server process, masquerading as a client, and providing to the server process data that ranges from one more than the size of the buffer to several times the size of the buffer. If the server process accepts the data and stores it in its buffer, the assumption is considered violated and the server process vulnerable. It is important to note that the focus of this test strategy is on the size of the data being accepted by the server process. Evaluators can also create test strategies that focus on other aspects of the base strategy, e.g., masquerading as a client or format of the data. This test strategy is implemented by deriving multiple test cases where each test case involves varying the length of the data according to the range specified in the test strategy.

5.5 Randomness Resources

This subsection presents base strategies associated with the randomness resources and an example of deriving a test strategy from a base strategy. Those base strategies are provided in Table 5.5.

Table 5.5 Base strategies associated with randomness resources.

No.	Base strategies
1.	**Goal:** To test if the software process allows violation of the assumption: *the series of random data being produced by the PRNG is unpredictable (assume that seed is unpredictable).* **Target:** Random data series **Method:** Identify the PRNG that is being used by the software process. The assumption is considered violated if the process is using a PRNG that produces predictable random data series. PRNGs that produce predictable random data series can be found in the current cryptographic literature.
2.	**Goal:** To test if the software process allows violation of the assumption: *the seed being used by the PRNG is unpredictable.* **Target:** Seed **Method:** Identify the seed that the software process provides as input to the PRNG. The assumption is considered violated if the software process uses a predictable seed, such as system time or process pid.
3.	**Goal:** To test if the software process allows violation of the assumption: *the process will have easy access to entropic data on a computer system.* **Target:** Random data series **Method:** Identify the method that the software process uses to gather entropic data on the computer system. The assumption is considered violated if the process uses predictable computer system events such as system time, number of running processes or process pid to gather entropic data.
4.	**Goal:** To test if the software process allows violation of the assumption: *the process will be able to accurately estimate the entropy of a data set.* **Target:** Random data series **Method:** Identify the method the software process uses to estimate the entropy of a data set. The

	assumption is considered violated if the process is not conservative in estimating the entropy of the data set. For example, the assumption is considered violated if the process assumes the entropy of a data set to be equal to the bit length of the data set.
5.	**Goal:** To test if the software process allows violation of the assumption: *user selected passwords/keys will have a sufficient amount of entropy.* **Target:** User selected passwords/keys **Method:** Identify the cryptographic algorithms and protocols being used by the software process that require highly entropic passwords or keys or both. The assumption is considered violated if the process is using user selected passwords or keys or both for these algorithms and protocols.
6.	**Goal:** To test if the software process allows violation of the assumption: *if two different seeds are provided to the PRNG, it is computationally infeasible to produce the same series of data both times.* **Target:** Random data series **Method:** Identify the PRNG that is being used by the software process. The assumption is considered violated if it is computationally feasible for the PRNG to produce the same random data series given two distinct seeds. Such PRNGs can be identified from the current cryptographic literature.
7.	**Goal:** To test if the software process allows violation of the assumption: *given that the PRNG is continuously producing random data, it is computationally infeasible to produce the same sequence of random data after some time.* **Target:** Random data series **Method:** Identify the PRNG that is being used by the software process. The assumption is considered violated if it is computationally feasible for the PRNG to produce the same random data series after a reasonable amount of time. Such PRNGs can be identified from the current cryptographic literature.

An Example

Base strategy 2 in Table 5.5 is used to provide a detailed example of deriving a test strategy to determine if a software process allows violation of an assumption associated with the randomness resources. The process in question is an online poker game. It uses a popular and readily available PRNG to generate a series of random numbers to shuffle cards for the poker game [VIEG01].

The information provided by the base strategy is refined and adapted to derive a test strategy to assess if the software process is vulnerable. For this specific process, the test strategy consists of studying the process documentation to identify the seed, and analyzing it to determine if it can be predicted. If the process uses a predictable number such as, system time or process pid as a seed, the assumption is considered violated and the process vulnerable. This test strategy is implemented by deriving a test case in which the evaluators first identify the seed by studying available process documentation, such as its source code or its design documentation, and then analyzing the seed to determine if it can be predicted.

5.6 Cryptographic Algorithms and Protocols

Table 5.6 lists the base strategies associated with cryptographic algorithms and protocols. An example of deriving a test strategy from one of the base strategies follows.

Table 5.6 Base strategies associated with cryptographic algorithms and protocols.

No.	Base strategies
1.	**Goal:** To test if the software process allows violation of the assumption: *random data being used by the cryptographic algorithm/protocol is unpredictable.* **Target:** Random data series **Method:** Identify the PRNG that is being used by the software process. The assumption is considered violated if the process is using a PRNG that produces predictable random data series, such as *rand()* system call. These PRNGs produce predictable series of numbers even if the seed provided as input is unpredictable. Such PRNGs can be found in the current cryptographic literature.
2.	**Goal:** To test if the software process allows violation of the assumption: *the length of the key being used by the cryptographic algorithm or protocol is sufficient.* **Target:** Key **Method:** Identify the minimum key length that the software process uses for the cryptographic algorithm or protocol. This assumption is considered violated if this minimum key length is less than the current standard. The minimum length of the key required for keeping the algorithm or protocol secure changes with time and can be found in the current cryptographic literature.

3.	**Goal:** To test if the software process allows violation of the assumption: *the hashing algorithm will not produce same hash for two different inputs.* **Target:** Hash value **Method:** Identify the hashing algorithm that the software process uses. There exist hashing algorithms that produce the same hash for two different input texts. These algorithms are considered compromised and cannot be used for cryptographic purposes. The assumption is considered violated if the software process uses one of the compromised hashing algorithms. Information regarding compromised hashing algorithms can be found in current cryptographic literature.
4.	**Goal:** To test if the software process allows violation of the constraint: *the process cannot use encryption to ensure data integrity.* **Target:** Ciphertext **Method:** Determine how the process is using encryption. This constraint is considered violated if the software process uses encryption to ensure data integrity. In other words, the constraint is considered violated if the process is not able to detect if the ciphertext has been modified or not.
5.	**Goal:** To test if the software process allows violation of the constraint: *the process cannot use a key more than once for a stream cipher.* **Target:** Key **Method:** Assess if the process uses the same key more than once for a stream cipher. The constraint is considered violated if the same key is used more than once.
6.	**Goal:** To test if the software process allows violation of the constraint: *the process cannot use one time pads to encrypt a large quantity of data.* **Target:** Ciphertext **Method:** Assess the use of one time pads by the software process. The constraint is considered violated if the process uses one time pads to encrypt large quantities of data. Clearly, what constitutes “a large quantity of data” is a subjective term. As used here, its value depends on the availability of high quality random data to the software process. Given that a software process is using a readily available random number generator, 1MB or more of data is considered a large quantity of data.
7.	**Goal:** To test if the software process allows violation of the constraint: *the process cannot use keys that are self reported by a client or a server.* **Target:** Key **Method:** Assess how the software process acquires keys from its clients, its peer processes or

	server processes. This constraint is considered violated if the process, without performing any validation, trusts keys that are self reported by the clients, peer processes or server processes.
8.	**Goal:** To test if the software process allows violation of the constraint: *the process cannot use obfuscation instead of encryption to ensure confidentiality.* **Target:** Obfuscated text **Method:** Assess to what extent the software process relies on obfuscation for keeping secrets. Obfuscation is the process of changing data to make it difficult to perceive or understand and includes techniques, such as using obscure formats and rearranging data. The constraint is considered violated if the process uses obfuscation to keep secrets.
9.	**Goal:** To test if the software process allows violation of the constraint: *the process cannot store keys/passwords in clear text.* **Target:** Key **Method:** Assess how the software process stores keys or passwords. The constraint is considered violated if the process stores keys or passwords in clear text.

An Example

Base strategy 4 in Table 5.6 is used to provide a detailed example of deriving a test strategy to determine if a software process allows violation of a constraint associated with cryptographic algorithms and protocols. The process in question runs on a computer system in a bank and regularly sends financial transactions to a central server. Although the process encrypts all transactions using a secret key, the format of the transactions follows a publicly available industry standard that includes the following fields: date, time, type and amount.

The information provided by the base strategy is refined and adapted to derive a test strategy to assess if the software process is vulnerable. For this specific process, the test strategy involves executing the software process, intercepting the encrypted transaction, modifying all combinations of individual fields, and then resending the encrypted transaction. Since the evaluators do not have access to the encryption key, they cannot decrypt the intercepted transaction. However, because they are familiar with the format of the transaction, they can directly modify the encrypted transaction. If the central server accepts any one of these modified transactions and uses it, the constraint is considered violated and the process vulnerable. This test strategy is implemented by deriving multiple test cases, where for each test case the individual fields in the transaction are modified according to the combinations specified by the test strategy.

Summary

This chapter presents the V&V strategies component, which provides base strategies to test a software process for the presence of vulnerabilities. A base strategy is a verification or a validation approach that provides a guided approach to test if the software process permits violation of a constraint or an assumption. The base strategies are used to derive test strategies, which again are V&V related, but are specific to the software process being tested. The V&V strategies component borrows its classification scheme from the taxonomy of vulnerabilities. Therefore, it has the same categories and subcategories as the taxonomy and the object model. The same classification scheme facilitates identifying relationships between the process objects in the object model, the constraints and assumptions in the taxonomy, and the base strategies in V&V strategies component.

Chapter 6

Security Assessment Process

This chapter describes the process of using the framework to assess security. This security assessment process provides a step-by-step procedure for testing a software process for the presence of vulnerabilities. Because the assessment process is generally applicable, it can be used to assess the security of any software process executing above the level of the operating system. The assessment process is applicable for situations where only the functionality of the software process is known, as well as the case where access to the complete development documentation is provided.

The remainder of this chapter presents the security assessment process and provides a detailed example of using it to test a software process for the presence of vulnerabilities.

6.1 Security Assessment Process

The security assessment process is a multi step procedure. A step-by-step account of the assessment process is provided below.

Step 1: Identifying resources

The security assessment process starts with identifying resources that the software process is using. This step requires knowledge of the structure of the framework components, and more specifically their subcategories, because they represent the resources a software process uses. After identifying the resources, the target resource is selected as a prelude to assessment. Targeting a single resource at a time focuses attention on a single subcategory and adds structure to the security assessment process.

Step 2: Identifying process objects

The second step is to identify potentially vulnerable process objects present in the software process. The object model facilitates their identification. The analysis is focuses by targeting only process objects present in the subcategory selected in the previous step. In this second step, the selected process objects are only those that are present in both the object model subcategory and in the software process. If available, the source code of the software process and/or its development documentation is analyzed to identify these objects. In the event that the source code or development documentation is unavailable, tools such as debuggers, filemon [FMON], regmon [RMON] and holodek [WHIT03] can be employed to identify these objects.

Step 3: Identifying constraints and assumptions

The third step is to identify relevant constraints or assumptions (or both) associated with the process objects identified in the previous step. Again, the analysis is focused by targeting constraints and assumptions listed in the subcategory identified in step 1. Constraints and assumptions in the selected subcategory are analyzed to identify the ones that are associated with the identified process objects, and which are relevant to the software process being tested.

Step 4: Identifying V&V strategies

The last step is to derive strategies for testing the software process for the presence of vulnerabilities. Again, the analysis is focused by targeting the base strategies present in the subcategory selected in step 1. Hence, in this step one selects the base strategies whose goal is violation of one of the identified constraints and assumptions and whose target includes one or more of the identified process objects. Because the base strategies are designed to be generally applicable, each base strategy can lead to the derivation of one or more test strategies. Deriving test strategies consists of refining and applying a base strategy to a particular software process. Similarly, refining each test strategy leads to one or more test cases for assessing if the software process allows the violation of a constraint or an assumption. In effect, test cases are instantiations of a test strategy, where each individual test case is realized by changing the values of the variable parameters present in the test strategy. Steps 2 through 4 are repeated for each resource identified in step 1.

6.2 Example

This section presents an example of using the security assessment process to assess security of a software process by testing it for the presence of vulnerabilities. The process in question is the "broken" version of the password program on SunOS and HP/UX. In these operating systems, the user information and his/her password are stored in the .rhosts file in the user's home directory. The password program executes with setuid root permissions. In other words, a user is executing the process, but it has effective root user privileges, which gives the process potential access to all of the resources of the computer system. The program, in course of its execution, asks the user to enter the new password for his/her account. It then modifies the password file to reflect the changes made by the user. This is a historic case of the *time of check time of use* exploit described in [BISH96a].

Below we describe the step by step execution of the password program [VIEGA01]:

1. The program opens and reads the password file, retrieves information about the user executing the program, and then closes the password file.
2. It then creates and opens a temporary file "ptmp" in the same directory as the password file.
3. It then opens the password file and copies its unchanged contents and the contents modified by the user (changed password) to the temporary file.

4. Finally, it closes the password file and renames the temporary file as the password file.

A user attacks this program by exploiting the time gap between the program checking access permissions of the user executing the program and actually making decisions based on those access permissions. A step by step procedure for executing the attack follows:

1. The user first provides the program with the name of a symbolic link that points to his/her own password file. Because the user has the required permissions for both the symbolic link and the file, the program opens the password file.
2. In step2, the program creates and opens the tmp file in the target user's password directory. Before the program starts the execution of step 2, however, the user changes the link to point to the password file of the target user. Because the program has setuid root permissions, it has access permissions for the target user's password file.
3. In step 3, the program copies the modified contents of the user's password file to the tmp file. Again, before the program starts the execution of this step, the user changes the link to point to his/her own password file. This is necessary because the program checks the password file for an entry with the user's id.
4. Finally, the user again changes the link back to the target user's password file. The program renames the tmp file in the target user's directory to the password file, thereby overwriting the target user's old password file.

This exploit gives the user the ability to log into the target user's account, thereby giving him/her complete control of the target user's domain. We now use the security assessment process outlined in previous section to analyze and test the password program for presence of vulnerabilities.

Step 1: Identifying resources

The first step is identifying resources being used by the password program. An analysis of the program's execution reveals that the resource in question is the filesystem. The filesystem, in terms of the structure of the framework components, is a subcategory of the I/O category.

Step 2: Identifying objects

The second step is to identify potentially vulnerable process objects being used by the password program. The security assessment process targets the filesystem subcategory of the object model. Process objects that are present both in the filesystem subcategory of the object model and in the password program are identified. Again, an analysis of the program's execution reveals that the process objects in question are files and directories. Specifically, the potentially vulnerable process objects present in the password program are: (1) the old password file, (2) ptmp, (3) the new password file, and (4) the password directory.

Step 3: Identifying constraints and assumptions

The third step is to identify relevant constrains or assumptions associated with the objects identified in the previous step. This is accomplished by analyzing all constraints and assumptions listed in the filesystem subcategory to identify those associated with files and directories. The constraints and assumptions are narrowed to:

- *Access permissions assigned to newly created files/directories are such that only the required principals have access to them.*
- *Access permissions of the files/directories being used by the process are such that only the required principals have access to them.*
- *A filename (including path) being used by the process is not a link that points to another file for which the user executing the process does not have the required access permissions.*
- *Files being used by the process cannot be observed/modified/replaced while the process is in execution.*

Step 4: Identifying V&V strategies

The fourth step is to select the base strategies whose goal is the violation of one of the identified constraints and assumptions and whose target includes a file or a directory. The selection process is facilitated by targeting the filesystem subcategory of the V&V strategies component. Table 6.1 lists the selected base strategies. Each of these base strategies is refined relative to the password program to derive one or more test strategies to test the password program for the presence of vulnerabilities. Table 6.1 also presents these test strategies (they follow the base strategy from which they are derived).

Table 6.1 Test strategies for password program example.

No.	**Base strategies**
1.	**Goal:** To test if the software process allows violation of the constraint: *access permissions assigned to newly created files/directories are such that only the required principals have access to them.* **Target:** File, directory **Method:** Access permissions of all newly created files or directories are analyzed. This constraint is considered violated if any principal other than the required principals has access to these files or directories.
	Test strategy #1: The password program is executed and tools such as filemon and holodec are used to test if only the password program has access permissions to the ptmp file. **Test strategy #2**: The password program is executed to test if only the principal executing the program has access permission to new password file (renamed ptmp file).
2.	**Goal:** To test if the software process allows violation of the constraint: *access permissions of the files/directories being used by the process are such that only the required principals have access to them.* **Target:** File, directory **Method:** Access permissions for all files and directories being used by the software process are analyzed. The constraint is considered violated if any principal other than the required principals has access to these files and directories.
	Test strategy #3: The access permissions for the old password file is examined to determine if only the principal executing the program has access permissions to it. **Test strategy #4**: The access permissions for the password directory is examined to determine if only the principal executing the program has access permissions to the password directory.

3.	**Goal:** To test if the software process allows violation of the assumption: *a filename (including path) being used by the process is not a link that points to another file for which the user executing the process does not have the required access permissions.* **Target:** File **Method:** The devised test is to determine if the software process allows violation of this assumption. This is accomplished by providing to the process a file that is a link to another file for which the user executing the process does not have the required access permissions. The assumption is considered violated if the software process accepts this file, follows the link, and uses the file pointed to by the link.
	Test strategy #5: We execute the password process and provide as input to it a filename that is a link to another file for which we do not have the required access permissions. Two types of links are used, hard links and soft links. We also vary access permissions of the file pointed to by the link, from partial access to no access.
4.	**Goal:** To test if the software process allows violation of the constraint: *files being used by the process cannot be observed/modified/replaced while the process is in execution.* **Target:** File **Method:** The software process is executed in a controlled environment such as holodeck [WHIT03]. We use this environment to identify, observe, modify, and replace the files that the process is using. This constraint is considered violated if the process allows us to observe file contents or uses the modified files or uses replaced files or all three.
	Test strategy #6: Execute the password program in a controlled environment such as holodeck and test if temporary file, ptmp, can be observed/modified while the process is in execution. **Test strategy #7**: Execute the password program in a controlled environment such as holodeck and test if the file provided as input can be replaced while the process is in execution.

Deriving test cases from the identified base strategies and executing them reveals vulnerabilities that form the building blocks of the *time of check time of use* flaw. In particular, the test cases reveal that the input file can be replaced while the password program is in execution – a serious security flaw. Additionally, the test cases reveal if the password program properly checks access permissions before opening a file pointed to by a link. Another serious flaw that is discovered is that the contents of the temporary file and its location can be observed while the program is in execution. Furthermore, the test cases reveal any discrepancy in the access permissions of the files and the directory being used by the program.

Summary

This chapter presents the process of using the security assessment framework. The process provides a systematic and structured approach for identifying vulnerabilities present in a software process. Additionally, the chapter also presents a detailed example of using the security assessment process to test a software process for the presence of vulnerabilities.

Chapter 7

Conclusions and Future Directions

This dissertation presents a framework for deriving verification and validation (V&V) strategies to assess software security. The framework is composed of three components: (1) the object model, (2) the taxonomy of vulnerabilities, and (3) the V&V strategies component. The object model provides an organized collection of potentially vulnerable process objects. The taxonomy of vulnerabilities provides an informative classification of vulnerabilities, wherein vulnerabilities are characterized as constraints imposed by computer system resources and assumptions made about the usage of these resources. The V&V strategies component combines the information from the object model and the taxonomy, and provides the base strategies to test a software process for the presence of vulnerabilities. These base strategies are used as a foundation for deriving detailed test strategies. This dissertation also defines a process for using the framework to assess security. The process provides a systematic and standardized approach to test a software process for the presence of vulnerabilities.

The framework offers a number of advantages for assessing software security. The primary advantage is that it enunciates a novel approach, which simplifies the testing of a software process for the presence of vulnerabilities. A vulnerability exists if the software process permits the violation of a constraint or an assumption.

Additionally, the framework includes an inventory of existing vulnerabilities, presented in the form of a taxonomy. Therefore, evaluators assessing the security of a software process using the framework have a predefined set of potential vulnerabilities from which to choose – a choice guided by the taxonomy. Furthermore, the framework provides base strategies, which the evaluators can use to test a software process for presence of vulnerabilities. In short, the framework reduces dependence on an evaluator's own abilities and past experience when assessing a process security.

The framework provides a comprehensive solution for assessing security; it can be applied to assess the security of any software process executing above the level of the operating system. Although, the taxonomy categorizes a large number of vulnerabilities taken from such sources as security texts [VIEG01, HOWA02], security lists [BUGT], academic and non-academic articles, and CERT advisories [CERT], we recognize that the encompassing framework is still limited by the vulnerabilities categorized by the taxonomy. The process defined in Section 3.3, extending the taxonomy, attempts to address this concern.

The remainder of this chapter presents the main contributions of the research described in this dissertation and outlines future work.

7.1 Summary of Main Contributions

The research presented in this dissertation contributes to the field of software security. A summary of its main contributions are listed below:

- A theoretical model of computing that establishes the relationship between a software process, vulnerabilities and computer system resources. The model supports the characterization of vulnerabilities as constraints imposed by computer system resources and assumptions made about the usage of these resources.
- A taxonomy of vulnerabilities that reflects an informative classification of known vulnerabilities, which are in the form of constraints and assumptions.

- A process for extending the taxonomy of vulnerabilities that specifies a step-by-step approach for identifying new constraints and assumptions and adding them to the taxonomy.
- A preliminary study attesting to the utility of the taxonomy. The study classifies vulnerabilities present in advisories issued by CERT [CERT] from Jan 2004 to June 2004. The study also helped in evolving the taxonomy by identifying new constraints and assumptions.
- An object model that represents an organized collection of potentially vulnerable process objects that can be present and exploited in a software process. The process objects in the object model are derived by analyzing constraints and assumptions listed in the taxonomy of vulnerabilities.
- A V&V strategies component, which provides base strategies to test a software process for the presence of vulnerabilities. The objects from the object model serve as targets for the base strategies, and the constraints and assumptions serve as goals of these base strategies. Moreover, each base strategy includes a method for testing if a software process allows a violation of a constraint or an assumption.
- A framework that provides a comprehensive approach for assessing the security of a software process. The framework can be used to assess the security of any software process that executes above the level of the operating system.
- A process for using the framework to assess the security of a software process. The process provides a systematic and ordered approach for using the framework to test a software process for the presence of vulnerabilities.
- Basis for a common template for reporting exploits and the causative vulnerabilities.

7.2 Future Work

This dissertation presents a framework for assessing the security of a software process by testing it for the presence of vulnerabilities. The framework serves as a foundation upon which one can explore complementary areas of research. This section presents those areas as future work.

7.2.1 Refining the Process of Using the Framework

Software development is a dynamic field with new development paradigms and new programming languages being discovered constantly. It is important that the process of using the framework maintain currency in light of this dynamism. This can be accomplished by using the framework to test additional real-world software applications for presence of vulnerabilities. This testing can provide additional insights into the relationship between the process and a software application or a domain of software applications. An understanding of this relationship will help refine and evolve the process of using the framework.

7.2.2 Identification and Development of Tools

It is important to both identify and develop tools that simplify the process of using the framework. Such tools assist in and help reduce the time required for testing a software application, thereby increasing the usefulness of the framework. For example, tools that assist in identification of potentially vulnerable process objects present in the software application and those that assist in the execution of base V&V strategies are highly desirable.

7.2.3 Adding New Vulnerabilities to the Taxonomy

Vulnerabilities are dynamic in nature, with new variations of old vulnerabilities being discovered constantly. Moreover, completely new vulnerabilities are being periodically discovered. Hence, it is important to keep the taxonomy of vulnerabilities updated to ensure the longevity and usefulness of both the taxonomy and the framework. Therefore, newly identified exploits and vulnerabilities should be incorporated into the taxonomy using the process outlined in Section 3.3.

7.2.4 Identifying Critical Constraints and Assumptions

Another direction for future work involves analyzing the most common exploits and selecting a subset of constraints and assumptions that are employed by a majority of them. Identifying such a subset minimizes the number of constraints and assumptions against which a software application needs to be tested. This stems from the fact that a single exploit requires an attack on multiple constraints and assumptions. Identifying and testing for the most commonly violated constraints and assumptions can, more effectively, reduce the number of exploits on a software application. Principles of boolean algebra can be used to maximize the number of exploits covered by this set of selected constraints and assumptions.

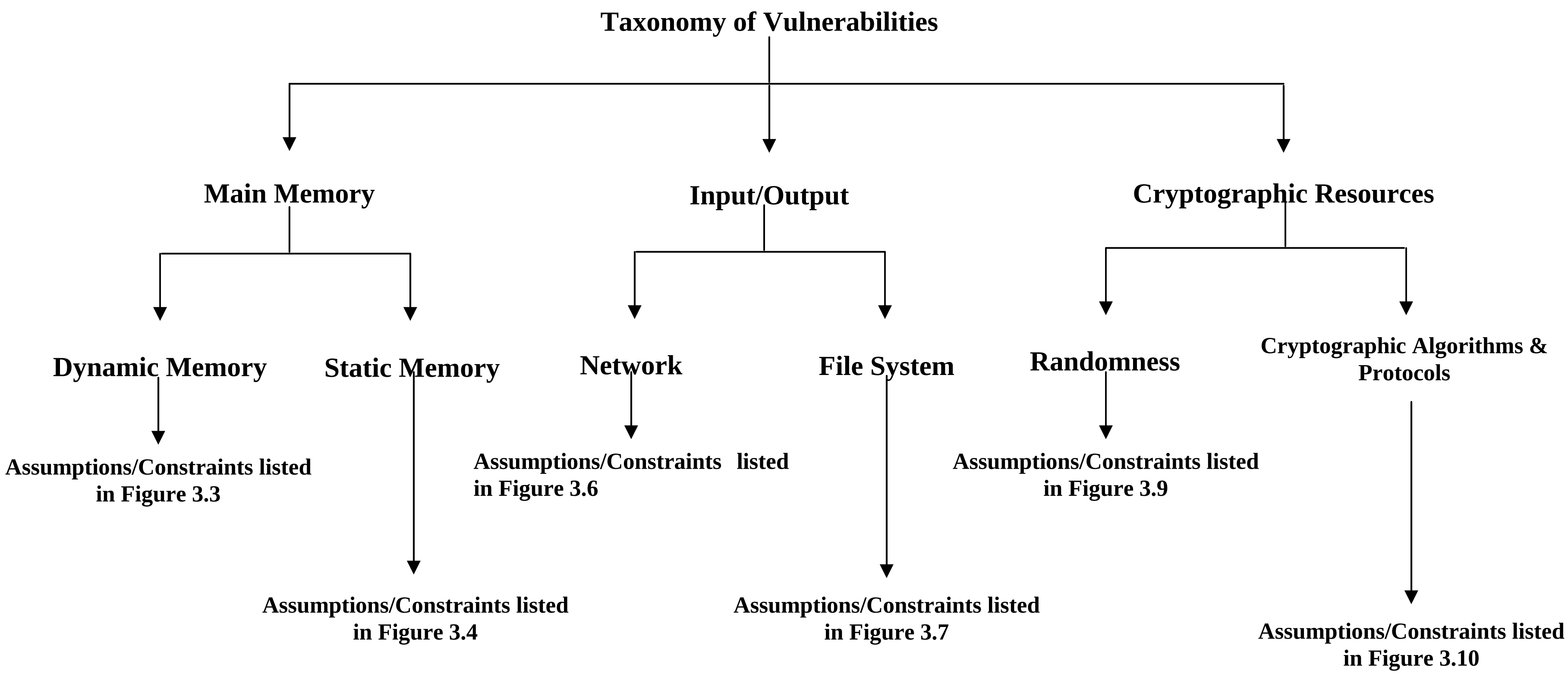
7.2.5 Deriving Requirements for Developing Secure Software

The constraints and assumptions listed in the taxonomy can also be used to derive requirements for developing secure software. These requirements can guide the design and coding phases of the software development lifecycle. In turn, they enable a seamless integration of the framework into a test process for software applications.

Summary

In recent years, software applications have been exploited with increased frequency. These exploits have caused substantial economic damages. Consequently, securing software applications against these exploits has become a top priority. The last line of defense in securing a software application is to test *a priori* if it is vulnerable to exploits. In this dissertation, the author has presented an approach for assessing the extent to which a software application is vulnerable to exploits by testing it for the presence of vulnerabilities. The approach takes the form of a framework, which provides a step-by-step procedure for identifying vulnerabilities present in a software application. The framework not only supports the testing of a software application for the presence of vulnerabilities, but also reduces the reliance on the evaluator's ability and past experience. The framework is both novel and distinctively different from the ones found in current literature.

Appendix A


Taxonomy of Vulnerabilities
Main Memory
Input/Output
Cryptographic Resources
Dynamic Memory
Static Memory
Network
File System
Randomness
Cryptographic Algorithms & Protocols
Assumptions/Constraints listed in Figure 3.3
Assumptions/Constraints listed in Figure 3.6
Assumptions/Constraints listed in Figure 3.9
Assumptions/Constraints listed in Figure 3.4
Assumptions/Constraints listed in Figure 3.7
Assumptions/Constraints listed in Figure 3.10

Appendix B

This appendix presents a summary table of some notable taxonomies, not discussed in the related work section of chapter 3 (section 3.6). In creating this table, we have borrowed heavily from unpublished doctoral dissertations of Krusl [KRSU98] and Lough [LOUG01]. Please refer to their dissertations for a detailed discussion of these taxonomies.

No.	Taxonomy	Comments
1.	Knuth's classification [KNUT89]	Classification is subjective and ambiguous.
2.	DeMillo and Mathur's grammar-based classification [DEMI95]	Classification is ambiguous.
3.	Endres's classification [ENDR75]	Classification is application and machine dependent.
4.	Ostrand and Weyuker's classification [OSTR84]	Classification is ambiguous.
5.	Basili and Perricone's classification [BASI84]	Classification is ambiguous.
6.	Origin and causes classification [LONG97]	Classification is ambiguous and is difficult to use without knowledge of state of mind of programmer.
7.	Access required classification [LONG97]	Classification is ambiguous and there is no clear definition of each category.
8.	Ease of exploit classification [LONG97]	Classification is ambiguous.
9.	Threat classification [POWE96]	Classification is ambiguous.
10.	Cohen's attack classification [COHE95; COHE97]	Classification is descriptive, non-orthogonal, incomplete, and of limited applicability.
11.	Perry and Wallich's attack classification [PERR84]	Classification is ambiguous.
12.	Howard's process-based taxonomy of network attacks [HOWA97]	Categories of the taxonomy are not mutually exclusive.
13.	Anderson's penetration matrix [ANDE80]	It is not a vulnerability taxonomy as it classifies vulnerabilities based on penetrator.
14.	Jayaram and Morse's network security taxonomy [JAYA97]	It is not comprehensive enough to qualify as a taxonomy.
15.	Sandia laboratory taxonomy [CHRI99]	Categories of the taxonomy are not mutually exclusive.
16.	Kumar's classification and detection of computer intrusions [KUMA95]	Classification can only handle intrusions that appear in audit logs.
17.	Brinkley's computer misuse	Categories of the taxonomy are not mutually

	techniques [BRIN95]	exclusive.
18.	Dunnigan and Nofi [DUNN95]	The taxonomy classifies deception techniques.
19.	Straub and Widom's motivation-security response taxonomy [STRA84]	The taxonomy lists motivation of the attacker.

References

[ABBO76] R. P. Abbot et al. Security analysis and enhancements of computer operating systems. Tech. Report No. NBSIR 76-1041, Lawrence Livermore Laboratory, Institute for Computer Science and Technology, 1976.

[AHUJ96] V. Ahuja. *Network & Internet Security*. AP Professional, MA, 1996.

[ALEP96] Aleph one. Smashing the stack for fun and profit. *Phrack Magazine*, Issue 49, 1996.

[ALTU02] V. Atluri and A. Gal. An authorization model for temporal and derived data: Securing information portals. *ACM Transactions on Information and System Security*, Vol. 5, No. 1, pp. 62-94, Feb 2002.

[AMOR91] E. Amoroso. Towards an approach to measuring software trust. IEEE Computer Society Symposium on Research in Security and Privacy, pp. 198-218, 1991.

[AMOR94a] E. G. Amoroso. *Fundamentals of computer security technology*. Prentice-Hall PTR, NJ, 1994.

[AMOR94b] E. Amoroso, C. Taylor, J. Watson and J. Weiss. A process-oriented methodology for assessing and improving software trustworthiness. In Proceedings of 2nd ACM Conference on Computer and Communications Security, pp. 39-50, 1994.

[ANDE80] J. P. Anderson. Computer security threat monitoring and surveillance. Tech. Report Contract 79F296400, James P. Anderson Co., Fort Washington, PA, 1980.

[ANDE93] R. Anderson. Why cryptosystems fail. *Communications of the ACM*, Vol. 37, No. 11, pp. 32-40, November, 1993.

[ARQU93] J. Arquilla and D. Ronfeldt. Cyberwar is Coming! *Comparative Strategy*, Vol. 12, No. 2, pp. 141–165, April–June, 1993.

[ATTA76] C. R. Attanasio, P. W. Markstein, and R. J. Phillips. Penetrating an operating system: A study of VM/370 integrity. *IBM System Journal*, Vol. 15, No. 1, pp. 102–116, 1976.

[BASI84] V. Basili and B. Perricone. Software errors and complexity. *Communications of the ACM*, Vol. 27, No. 1, pp. 42-52, 1984.

[BAZA04] A. Bazaz, J. D. Arthur, and R. Marchany. Reflections on operating in hostile environments. In Proceedings of 37th Hawaii International Conference on System Sciences, IEEE Computer Society, 2004.

[BEIZ83] B. Beizer. *Software testing techniques*. Electrical Engineering/Computer Science and Engineering Series, Van Nostrand Reinhold, New York, 1983.

[BELL92] S. M. Bellovin. There be dragons. In UNIX Security Symposium. USENIX Association, July, 1992.

[BELL94] S. M. Bellovin and W. R. Cheswick. Network firewalls. *IEEE Communications Magazine*, Vol. 32, No. 9, pp. 50–57, Sep 1994.

[BERG02] F. Bergadano, D. Gunetti and C. Picardi. User authentication through keystroke dynamics. *ACM Transactions on Information and System Security*, Vol. 5, No. 4, pp. 367-397, Nov 2002.

[BERN02] M. Bernaschi, E. Gabrielli and L. V. Mancini. REMUS: A security-enhanced operating system. *ACM Transactions on Information and System Security*, Vol. 5, No. 1, pp. 36-61, Feb 2002.

[BERT02] E. Bertino and E. Ferrari. Secure and selective dissemination of XML documents. *ACM Transactions on Information and System Security*, Vol. 5, No. 3, pp. 290-331, Aug 2002.

[BISB78] R. Bisbey II and D. Hollingworth. Protection analysis: Final report. Tech. Report No. ISI/SR-78-13, Information Sciences Institute, University of Southern California, CA, 1978.

[BISH86] M. Bishop. How to write a setuid program. Cray User Group Proceedings, 1986.

[BISH95] M. Bishop. A taxonomy of UNIX system and network vulnerabilities. Tech. Rep. No. CSE-95-10, University of California, Davis, CA, May 1995.

[BISH96a] M. Bishop and M. Dilger. Checking for race conditions in file accesses. *Computer Systems*, Vol. 9, No. 2, pp. 131-152, 1996.

[BISH96b] M. Bishop and D. Bailey. A critical analysis of vulnerability taxonomies. Tech. Rep. No. CSE-96-11, Department of Computer Science, University of California, Davis, CA, 1996.

[BONT] V. Bontchev. Possible virus attacks against integrity programs and how to prevent them. Retrieved Nov 2005 from http://www.people.frisk-software.com/~bontchev/papers/attacks.html

[BRIN95] D. L. Brinkley and R. R. Schell. What is there to worry about? An introduction to the computer security problem. M. D. Abrams, S. Jajodia and H. J. Podell (Eds.), *Information security: An integrated collection of essays*. pp. 11–39. IEEE Computer Society Press, Los Alamitos, CA, 1995.

[BUGT] Bugtraq mailing list. Retrieved May 2005 from http://www.securityfocus.com/archive

[CCIT] Common criteria board. Common criteria for information technology security evaluation, Version 3.0. Retrieved August 2005 from http://www.commoncriteriaportal.org/

[CERT] Computer emergency response team. Retrieved April 2005 from http://www.cert.org

[CEVM] Common evaluation methodology. Retrieved Aug 2005 from http://www.commoncriteriaportal.org/public/files/ccmb-2005-07-004.pdf

[CHEN02] H. Chen, D. Wagner and D. Dean. Setuid demystified. In Proceedings of the 11th USENIX Security Symposium, pp. 171-190, Aug 2002.

[CHRI99] J. Christy. Cyber Threat & Legal Issues. Presentation given at Shadowcon, Dahlgren, VA; J. Christy (Air Force Office of Special Investigations), 1999.

[COHE95] F. B. Cohen. *Protection and Security on the Information Superhighway*. John Wiley & Sons, Inc., Indianapolis, IN, 1995.

[COHE97] F. Cohen. Information system attacks: A preliminary classification scheme. *Computers and Security*, Vol. 16, No. 1, pp. 29-46, 1997.

[COPS] Computer oracle and password system. Retrieved Aug 2005 from http://ciac.llnl.gov/ciac/ToolsUnixSysMon.html

[COWA98] C. Cowan et al. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In Proceedings of the 7th USENIX Security Conference, pp. 63-78, 1998.

[CURR92] D. A. Curry. *UNIX system security*. Addison-Wesley, Boston, MA, 1992.

[DAMI02] E. Damiani, S. de C. di Vimercati, S. Paraboschi and P. Samarati. A fine-grained access control system for XML documents. *ACM Transactions on Information and System Security*, Vol. 5, No. 2, pp. 169-202, May 2002.

[DEAN02] D. Dean, M. Franklin and A. Stubblefield. An algebraic approach to IP traceback. *ACM Transactions on Information and System Security*, Vol. 5, No. 2, pp. 119-137, May 2002.

[DEMI95] R. A. DeMillo and A. P. Mathur. A grammer based fault classification scheme and its applications to the classification of the errors of TeX. Tech. Report No. SERC-TR-165-P, Software Engineering Research Center, Purdue University, IN, 1995.

[DENN99] D. E. R. Denning. *Information warfare and security*. Addison-Wesley, Boston, MA, 1999.

[DESF93] Data encryption standard. Federal Information Processing Standards Publication 46-2, Dec 1993.

[DILD98] Dildog. The tao of windows buffer overflow. File 351 of the Cult of the Dead Cow. Retrieved Oct 2005 from http://www.cultdeadcow.com/cDc_files/cDc-351/index.html (Written in April 1998).

[DoD85] Department of Defense. Trusted computer system evaluation criteria. DOD 5200.28-STD edition, Dec 1985.

[DUNN95] J. F. Dunnigan and A. A. Nofi. *Victory & deceit: Dirty tricks at war*. Quill William Morrow, New York, 1995.

[ENDR75] A. Endres. An analysis of errors and their causes in system programs. In Proceedings of the International Conference on Reliable Software, pp. 327-336, 1975.

[FMON] Filemon. SysInternals freeware. Retrieved Aug 2005 from http://www.sysinternals.com/Utilities/Filemon.html

[GOGU82] J. A. Goguen and J. Meseguer. Security policies and security models. In Proceedings of the IEEE Computer Society Symposium on Research in Security and Privacy, pp. 11-20, 1982.

[GOLD96] D. M. Goldschlag. Several secure store and forward devices. In Proceedings of the 3rd ACM Conference on Computer and Communications Security, pp. 129-137, 1996.

[GORD02] L. A. Gordon and M. P. Loeb. The economics of information security investment. *ACM Transactions on Information and System Security*, Vol. 5, No. 4, pp. 438-457, Nov 2002.

[GUPT91] S. Gupta and V. D. Gligor. Towards a theory of penetration-resistant systems and its applications. In Proceedings of the Computer Security Foundations Workshop IV, pp. 62–78, 1991.

[GUPT92] S. Gupta and V. D. Gligor. Experience with a penetration analysis method and tool, In 15th National Computer Security Conference, pp. 165–183, 1992.

[HAYE96] R. E. Hayes and G. Wheatley. *Information warfare and deterrence*. NDU Press Book, Washington, DC, 1996.

[HENN96] J. Hennessy and D. Patterson. *Computer architecture: A quantitative approach.* Morgan Kaufmann Publishers, San Francisco, CA, 1996.

[HOFF90] L. J. Hoffman (Ed.). *Rogue programs: Viruses, worms, and trojan horses*. Van Nostrand Reinhold, New York, 1990.

[HOLL74] D. Hollingworth, S. Glaseman and M. Hopwood. Security test and evaluation tools: An approach to operating system security analysis. The Rand Paper Series P-5298, Rand Corporation, Santa Monica, CA, 1974.

[HOLL76] D. Hollingworth and R. Bisbey II. Protection errors in operating systems: Allocation/Deallocation residuals. Tech Report No. ISI/SR-76-7, USC/Information Sciences Institute, CA, 1976.

[HOWA97] J. D. Howard. *An analysis of security incidents on the Internet*. Unpublished doctoral dissertation, Carnegie Mellon University, Pittsburgh, PA, 1997.

[HOWA02] M. Howard and D. C. LeBlanc. *Writing secure code* (2nd ed.), Microsoft Press, Redmond, WA, 2002.

[HUWA01] B. Hutchinson and M. Warren. *Information warfare*. Butterworth Heinemann, 2001.

[ISSC] Internet security scanner. Retrieved Aug 2005 from http://ciac.llnl.gov/ciac/ToolsUnixSysMon.html#Iss

[ITSE] Information technology security evaluation criteria (ITSEC). Retrieved Aug 2005 from http://nsi.org/Library/Compsec/eurooran.txt

[ITS4] ITS4: Software security tool. Retrieved Aug 2005 from http://www.cigital.com/its4/

[JAGE94] T. Jager and A. Prakash. Support for file system security requirements of computational e-mail systems. In Proceedings of the 2nd ACM Conference on Computer and Communications Security, pp. 1-9, 1994.

[JAYA97] N.D. Jayaram and P. L. R. Morse. Network security - A taxonomic view. In European Conference on Security and Detection, School of Computer Science, University of Westmister, UK, IEE Conference Publication No. 437, 1997.

[KAHN96] D. Kahn. *The codebreakers: The story of secret writing* (2nd Ed.). Scribner, New York, 1996.

[KARG74] P. A. Karger and R. R. Schell. Multics security evaluation: Vulnerability analysis. Tech Report No. ESD-TR-74-193, Information Systems Technology Applications Office, Deputy for Command and Management Systems, Electronic Systems Division (AFSC), L.G. Hanscon AFB, MA, June 1974.

[KERN88] B.W. Kernighan and D. M. Ritchie. *The C Programming Language* (2nd Ed.). Prentice Hall Software Series, NJ, 1988.

[KIMG94] G. H. Kim and E. H. Spafford. The design and implementation of tripwire: A file system integrity checker. In Proceedings of the 2nd ACM Conference on Computer and Communications security, pp. 18-29, 1994

[KNIG00] E. Knight. Computer vulnerabilities. Retrieved March 2000 from www.securityparadigm.com

[KNUT81] D. E. Knuth. *The art of computer Programming, Volume 2: Seminumerical Algorithms* (3rd Ed.). Addison-Wesley, Boston, MA, 1997.

[KNUT89] D.E. Knuth. The errors of TEX. *In Software-Practice and Experience*, Vol. 19, No. 7, pp. 607-685, 1989.

[KONI76] W. L. Konigsford. A taxonomy of operating-system security flaws. Tech Report No. UCID-17422, Lawrence Livermore Laboratory, Nov 1976.

[KRSU98] I. V. Krsul. *Software vulnerability analysis.* Unpublished doctoral dissertation, Purdue University, IN, 1998.

[KUMA95] S. Kumar. *Classification and detection of computer intrusion.* Unpublished doctoral dissertation, Purdue University, IN, 1995.

[LEEW00] W. Lee, W. Fan, M. Miller, S. Stolfo and E. Zadok. Towards cost sensitive modeling for intrusion detection and response. In Proceedings of the ACM SIGMOD International Conference on the Management of Data, 2000.

[LEVY84] S. Levy. *Hackers heroes of the computer revolution*. Anchor Press/Doubleday, New York, 1984.

[LONG97] T. Longstaff. Update: CERT/CC vulnerability knowledgebase. Technical presentation at a DARPA workshop in Savannah, Georgia, 1997.

[LOSC98] P. A. Loscocco, S. D. Smalley, P. A. Muckelbauer, R. C. Taylor, S. J. Turner and J. F. Farrell. The inevitability of failure: The flawed assumption of security in modern computing environments. In 21st National Information Systems Security Conference, pp. 303–314, 1998.

[LOUG01] D. L. Lough. *A taxonomy of computer attacks with applications to wireless networks.* Unpublished doctoral dissertation, Virginia Polytechnic Institute and State University, Blacksburg, VA, 2001.

[LIND75] R. R. Linde. Operating system penetration. In National Computer Conference, pp. 361–368, Santa Monica, CA, 1975.

[LIND97] U. Lindqvist and E. Jonsson. How to systematically classify computer security intrusions. In Proceedings of the IEEE Symposium on Security and Privacy, IEEE Computer Society Press, Los Alamitos, CA, pp. 154-163, 1997.

[MASS02] M. Bernaschi, E. Gabrielli and L. V. Mancini. REMUS: A security enhanced operating system. *In ACM Transactions on Information and System Security*, Vol. 5, No. 1, pp. 36-61, 2002.

[McGR97] G. McGraw and E. W. Felten. *Java security: Hostile applets, holes, and antidotes*. John Wiley & Sons Inc., New York, 1997.

[MICH02] C. C. Michael and A. Gosh. Simple state-based approaches to program-based anomaly detection. *ACM Transactions on Information and System Security*, Vol. 5, No. 3, pp. 203-237, August 2002.

[MOLV94] R. Molva and E. Rutsche. Application access control at network level. In Proceedings of the 2nd ACM Conference on Computer and Communications Security, pp. 219-228, 1994.

[MUDG97] P. Mudge and Y. Benjamin. D´ej`a vu all over again. *Byte*, Vol. 22, No. 11, pp. 81–86, Nov 1997.

[MWEB93] *Merriam-Webster's collegiate dictionary* (10th Ed.). Merriam-Webster, Springfield, MA, 1993.

[NEED93] R. M. Needham. Denial of service. In Proceedings of the 1st ACM Conference on Computer and Communications Security, pp. 151-153, 1993

[NEED94] R. M. Needham. Denial of service: An example. *Communications of the ACM*, Vol. 37, No. 11, pp. 42–46, Nov 1994.

[NEED97] R. M. Needham. The changing environment for security protocols. *IEEE Network*, Vol. 11, No. 3, pp. 12-15, May/June 1997.

[NEST94] J. P. Nestor and E. S. Lee. Prerequisite confidentiality. In Proceedings of the 2nd ACM Conference on Computer and Communications Security, pp. 282-293, 1994.

[NEUM89] P. G. Neumann and D. B. Parker. A Summary of computer misuse techniques. In the Proceedings of 12th National Computer Security Conference, pp. 396-407, 1989.

[NEUM95] P. G. Neumann. *Computer related risks*. ACM Press (A division of the Association for Computing Machinery Inc.), 1995.

[NETS99] Netscape defective password cryptography exploit. 1999. Retrieved March 2005 from http://www.cigital.com/news/?pg=art&artid=27

[OSTR84] T. Ostrand and E. Weyuker. Collecting and categorizing software error data in an industrial environment. *The journal of Systems and Software*, Vol. 4, No. 4, pp. 289-300, 1984.

[PARN90] D. Parnas et al. Evaluation of safety-critical software. *Communications of the ACM*, Vol. 33, No. 6, 1990.

[PERR84] T. Perry and P. Wallich. Can computer crime be stopped. *IEEE Spectrum*, Vol. 21, No. 5, 1984.

[PETE95] I. Peterson. *Fatal defect: Chasing killer computer bugs*. Time Books, New York, 1995.

[PFLE94] S. L. Pfleeger, N. Fenton and S. Page. Evaluating software engineering standards. *IEEE Computer*, Vol. 27, No. 9, pp. 71-79, 1994.

[PHRA] Phrack- A hacker magazine by the community for the community. Retrieved April 2005 from http://www.phrack.org

[POLK92] W. Timothy Polk. Automated tools for testing computer systems vulnerability. NIST SP 800-6, Dec 1992.

[POWE96] R. Power. Current and future danger: A CSI primer of computer crime and information warfare. *CSI Bulletin*, 1996.

[REIN93] R. B. Reinhardt. An architectural overview of UNIX network security. Tech. Report, ARINC Research Corporation, Feb 1993.

[RMON] Regmon. SysInternals freeware. Retrieved August 2005 from http://www.sysinternals.com/Utilities/Regmon.html

[ROCH89] J. A. Rochlis and M. W. Eichin. With microscope and tweezers: The worm from MIT's perspective. *Communications of the ACM*, Vol. 32, No. 6, pp. 689–698, June 1989.

[RUDY02] A. Rudys and D. S. Wallach. Termination in language-based systems. *ACM Transactions on Information and System Security*, Vol. 5, No. 2, pp. 138-168, May 2002.

[SCHN96] B. Schneier. *Applied cryptography* (2nd Ed.). John Wiley & Sons, Hoboken, NJ, 1996.

[SCHN00] B. Schneier. *Secrets & lies: Digital security in a networked world*. John Wiley & Sons, Hoboken, NJ, 2000.

[SEFO] Security focus. Retrieved March 2005 from http://www.securityfocus.com/

[SIMT01] T. Simth. Hacker jailed for revenge sewage attacks. The Register, 2001. Retrieved May 2001 from http://www.theregister.co.uk/content/4/22579.html

[SING96] A. Singer and R. Scott. Information warfare: An old operational concept with new implications. Institute for National Strategic Studies, No. 99, Dec 1996.

[SING00] S. Singh, *The code book - The science of secrecy from ancient history to quantum cryptography* (1st Ed.), Anchor Books, 2000.

[SMIT01] R. E. Smith. Cost profile of a highly assured, secure operating system. *ACM Transactions on Information and System Security*, Vol. 4, No. 1, pp. 72-101, 2001.

[SORI93] M. Soriano, J. Forne, F. Recacha and J. L. Melus. A particular solution to provide secure communications in an ethernet environment. In Proceedings of 1st ACM Conference on Computer and Communications Security, pp. 17-25, 1993.

[SSEC] Systems security engineering - capability maturity model (SSE-CMM). Model Description Document, Version 3.0. Retrieved Aug 2005 from http://www.software.org/quagmire/descriptions/sse-cmm.asp

[STAL95] W. Stallings. *Network and internetwork security principles and practice*. Prentice Hall, NJ, 1995.

[STEV94] W. R. Stevens. *TCP/IP Illustrated, Vol. 1: The protocols*. Addison-Wesley Professional Computing Series, Boston, MA, 1994.

[STEV98] W. R. Stevens. *UNIX network programming, Vol. 1: The sockets networking API* (2nd Ed.). Prentice Hall, NJ, 1998.

[STEV99] W. R. Stevens. *UNIX network programming, Vol. 2: Interprocess communications* (2nd Ed.). Prentice Hall, NJ, 1999.

[STOL90] C. Stoll. *Cuckoo's egg*. Pocket Books, New York, 1990.

[STRA84] D. W. Straub Jr. and C. S. Widom. Deviancy by bits and bytes: Computer abusers and control measures. In Proceedings of the 2nd IFIP International Conference on Computer Security, 1984.

[SWAT] SWATCH: System for monitoring events. Retrieved Aug 2005 from http://ciac.llnl.gov/ciac/ToolsUnixSysMon.html#Swatch

[TAIM95] T. Aslam. *A taxonomy of security faults in the UNIX operating system*. Unpublished master's thesis, Purdue University, IN, 1995.

[TAIM96] T. Aslam, I. Krsul and E. H. Spafford. Use of taxonomy of security faults. In 19th National Information Systems Security Conference, 1996.

[TANE03] A. S. Tanenbaum. *Computer Networks* (4th Ed.). Prentice Hall, NJ, 2003.

[TCSE] Trusted computer system evaluation criteria (TCSEC). Retrieved Aug 2005 from http://www.radium.ncsc.mil/tpep/library/rainbow/5200.28-STD.html

[THOM84] K. Thompson. Reflections on trusting trust. *Communications of the ACM*, Vol. 27, No. 8, pp. 761-773, 1984.

[TRIP] Tripwire: A cross-platform system integrity tool. Retrieved Aug 2005 from http://sourceforge.net/projects/tripwire/

[TRUS99] Computer Science and Telecommunications Board. *Trust in cyberspace*. National Academy Press, Washington, DC, 1999.

[TZUS63] S. Tzu. *The art of war* (S. B. Griffith Trans.). Oxford University Press, Oxford, UK, 1963 (Originally written in 400–320 B.C).

[VIEG01] J. Viega and G. McGraw. *Building secure software: How to avoid security problems the right way*. Addison-Wesley Professional Computing Series, Boston, MA, 2001.

[VIEG02] J. Viega, J. T. Bloch, T. Kohno and G. McGraw. Token-based scanning of source code for security problems. *ACM Transactions on Information and System Security*, Vol. 5, No. 3, pp. 238-261, Aug 2002.

[WALL90] L. Wall and R. Schwartz. *Programming Perl* (1st Ed.). O'Reilly & Associates, Inc., CA, 1990.

[WEIS73] C. Weissman. System security analysis / certification methodology and results. Tech. Report No. SP-3728, System Development Corporation, 1973.

[WEIS95] C. Weissman. Penetration testing. In M. D. Abrams, S. Jajodia and H. J. Podell (Eds.). *Information security: An integrated collection of essays.* pp. 269-296, IEEE Computer Society Press, 1995.

[WHTH03] J. A. Whittaker and H. H. Thompson. *How to break software security: Effective techniques for security testing*. Addison-Wesley, Boston, MA, 2003.

[WIKI] Wikipedia free encyclopedia. Retrieved Aug 2005 from http://en.wikipedia.org/wiki/Main_Page

[WRIG87] P. Wright. *Spycatcher: The candid autobiography of a senior intelligence officer*. Viking Press, New York, 1987.

[WRIG95] G. R. Wright and W. R. Stevens. *TCP/IP illustrated, Vol. 2: The implementation*. Addison-Wesley Professional Computing Series, Boston, MA, 1995.

Vita

Anil Bazaz was born in Srinagar, Kashmir, India on July 25th, 1976. He graduated with a Bachelor of Engineering Degree in Computer Technology from Nagpur University, India in 1999. He received his Master's degree in computer science from Virginia Tech, Blacksburg, VA in May 2003. He joined IOActive, Inc., Seattle, WA, as a security consultant in spring of 2006. His areas of interest include software security, software engineering, random numbers, cryptography, and quantum physics.