Vulnerability Database Integration with Intrusion Detection Systems

By

VINCENT LAW
B.S. (Boston University) 1990

THESIS

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

In

Computer Science

In the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Dr. Matt Bishop (Chair)

Dr. Karl Levitt

Dr. Felix Wu

Committee in Charge

2003

Table of Contents

Page

List of Figures

Page

List of Tables

Acknowledgements

There are so many people that I would like to thank. First, I thank God and my parents for bringing me to this world. The intelligence and wisdom given by God through my parents, and their supports and encouragement have been fantastic, especially whenever I was dealing with tough times. The guidance from my parents also helped me realize human values. They always emphasize the importance of continuous self-improvement. I also thank my sister for bringing joy to my life and making me forget the “growing pain” I was going through, especially when as a baby she “trusted” me by letting me hold her and “leaving” her safety net to me – an elementary school kid.

I am indebted to all the teachers who have taught me throughout the years because if not for them, I would not have been able to be at my current education level. I regret not being able to repay all of them as some of them are not around any more. My classmates throughout these years have been great too because they are the ones who I shared most of my daily life with, both in good and bad times. Even though we are having our own lives now, we still consider our friendships precious.

If not for the guidance from my undergraduate professors, I would not have been able to go on to my current graduate studies right now. They presented to me what college education is supposed to be, both in terms of academic achievements and of personal interactions and maturing processes. In addition, they showed me the meanings and significances of having appropriate attitudes in advanced-level studying. I also thank the Solomon family for their volunteer assistance in my adaptation to American life during my undergraduate years. The American social life and traditions that they demonstrated were invaluable to me when I was trying to settle down as a college student. Their hospitality helped ease my transition from a high school graduate in Hong Kong to a college freshman here in the United States when I began my college life in Boston.

I am also grateful to all my current and former employers and colleagues. They helped me build up my professional experience. Their experience led to my understanding on professionalism and the significance of pursuing further education in order to enhance and extend the materials I learned from college.

Without the relentless work from my graduate professors and advisors, I would not have been able to learn more materials which are helpful to my thesis work. I pay my tribute to my thesis advisor, Dr. Matt Bishop, for his non-stop willingness, assistance, patience, insights, reviews and comments on my thesis progress. I have also been grateful for other faculty members in related areas, especially Dr. Karl Levitt and Dr. Felix Wu, who have provided me additional supports and inspirations, both from their respective courses, which are helpful to my thesis, and from the expertise of their own research areas in computer security, networking and operating systems. Special thanks are for Mark Crosbie and Pierre Pasturel for their extra insights on their own professional projects and research works, giving me invaluable ideas during the progress of my thesis work.

Last but not least, this acknowledgement would not be complete without mentioning Kim Reinking, Melinda Day, and other administrative staff member in the department office for their tips on the administrative side of my studies. Without their guidance, I would not have been able to make the administrative deadlines during my academic progress. Their efforts for the graduate students have been amazing.

Vincent Law
June 2003
Computer Science

Abstract

One of the factors affecting the effectiveness of an intrusion detection system is how well it can deal with and protect against potential threats. Vulnerability schema offers a classifying scheme to existing vulnerabilities. This provides IDSes a more dynamic way to detect attacks, even if the attack has not been launched before. Hence, it is more desirable to integrate vulnerability database with an IDS so that IDSes can react to potential threats, such as previously-unfound buffer overflow and exploitation due to incomplete address space cleanup, in a more real-time manner. Such integration eliminates the dependence on attack-pattern signature, and the vulnerability database does not rely on any update or upgrade in the operating system.

The integration of vulnerability database with the IDS also involves the use of specification language. Specification language describes a specification of a module by defining negations to all vulnerabilities that this module can be exploited by. For easy implementation, an easy-to-understand schema is desired and the schema should base on a vulnerability taxonomy which classifies vulnerabilities in unambiguous fashion.

This thesis suggests a model that integrates a vulnerability database with existing IDSes. In order for this model to work well as a practical solution for intrusion

detection systems, this thesis also suggests a relational schema for the vulnerability database. A specification-based intrusion detection system integrated with a vulnerability database allows specifications to be dynamically updated upon any discovery of new vulnerability. This brings research on both vulnerability analysis and intrusion detection closer together.

Chapter 1 Introduction

1.1 Motivations

Computer security is an ongoing area of research in computer science, and it will continue to be so as long as there is a possibility of security breaches. The consequences of someone breaking into a system and stealing, destroying, or altering another person's identity have been pictured in the movie "The Net" [41], where Sandra Bullock's character suddenly became a non-person. A more terrifying situation can happen when the whole system was held for ransom, as in the movie "Die Hard 2" [48], when the intruders broke into the systems for the airport control center and "kidnapped" everybody inside the airport and every incoming flight by forcing the airport authorities to follow their instructions or else they would give the pilots of those incoming flights instructions that would lead to fatality. These situations happened in movies, but they could also happen in real life, as shown by the theft of credit card numbers which are later used in fraud and identity thefts [10].

Vulnerability analysis and intrusion detection have been two separate but related branches of computer security research. Advances in automation in software development and software analysis have led vulnerability analysis researchers to study methods for automating vulnerabilities detection. Vulnerability researchers often reference and update the vulnerability databases that they are constantly using. Meanwhile, with some modifications, any of these vulnerability databases can also be integrated for use in an intrusion detection system because the database provides a source of vulnerability data for the intrusion detection system. During a process, the intrusion detection system could verify that process with its specifications, which were created or updated based on the resulting vulnerability data for that process from the most recent vulnerability search. The IDS can use this information to detect attacks in a more real-time fashion. Hence, it makes potential break-ins more difficult to take advantage of the vulnerabilities during that process.

This thesis discusses a vulnerability classification scheme compatible with intrusion detection systems. It also presents examples to demonstrate how the database, based on this unambiguous classification scheme, can be used with a specification language to define consistent specifications that the vulnerabilities will violate. The IDS will use these specifications to detect exploitations of these vulnerabilities.

1.2 Background and Recent Works

The general public did not pay much attention to cyber attacks until the very first computer virus was documented during the early 1980s [42]. Reports on the first computer worms were released at about the same time. Most of the reports regarding a system under attack were unknown to the general public until one of the first high-profile network attacks in occurred 1988. In that attack, a worm spread throughout the Internet. One of its propagation methods relied on a program that used a system call, *gets*. Since *gets* lacks a boundary check [16, 43], the programs calling it are vulnerable to a buffer overflow attack. The worm exploited this *vulnerability* and other vulnerabilities in standard services provided by two versions of the UNIX operating systems. *Vulnerabilities*, or *security holes*, are *weaknesses* introduced by security-related bugs during one or more program executions [14]. The worm was apparently released accidentally and did not delete or alter files. The outcome would have been more destructive and disastrous had it been designed to do damage.

Past experience has shown that most of the attacks perform one or more of the following actions:

1. System file modifications
2. Unexpected and undesired user privilege modifications during a running process
3. Log file modifications
4. A *setuid* creation allowing root access to files or processes
5. Password guessing and cracking
6. Symbolic link modifications during program executions

These actions are the *building blocks* of attacks that kernel-based intrusion detection systems can detect because these actions are the results of the vulnerabilities existing inside system calls [13, 52]. Unfortunately, most software developers are not aware of these vulnerabilities, neglect them, do not understand how they can be exploited, or assume that these vulnerabilities will not be exploited [50]. One good example, *sendmail* version 8.9.3, calls *strcpy* 285 times [50], even though the function *strcpy* is prone to off-by-one error in buffer management, which is critical because the syntax of C language requires the a string variable assignment to include the null character at the end of the assigned string value [51]. This buffer overflow vulnerability allows the attacker to insert malicious code to obtain privileged access or change accessibilities, etc. Besides buffer overflow, the following is a list of high-level security problems when flaws are found in most system calls [27, 28].

1. Improper I/O validation, including the famous buffer overflow problem
2. Improper program and data sharing, such as the race condition
3. Improper use of cryptography, like the use of only the first 8 characters of the password in many versions of UNIX [31]
4. Weak authentication, allowing attacks such as the *man-in-the-middle* attack
5. Insecure bootstrapping, leaving undesired privileges for non-privileged users after system initialization
6. Improper configuration of the system, including access control settings and installation management/operation control

The *Internet Worm* attack mentioned above highlights the importance of intrusion detection systems that can use vulnerability information because the existence of vulnerabilities enables intruders to break into systems. Since vulnerabilities can be classified, we can also define consistent specifications based on vulnerability classifications. With these specifications, the attack patterns will then be able to be detected during an intrusion attempt. More details on how this is done will be explained in later chapters.

There are three major approaches in intrusion detections: anomaly detection, misuse detection/signature-based detection, and specification-based detection [45]. Most of the traditional intrusion detection systems are signature-based, in which the audit data is collected by the operating systems. Detection is based on matching the audit data to patterns corresponding to currently known attacks. Nonetheless, attacks can be changed so that the audit data will not match the attack patterns, and the IDS will not be able to detect the modified attack. Katherine Price demonstrated that “the audit data supplied by conventional operating systems lack content useful for misuse detection” [37]. So signature-based IDSes will miss some attacks.

Anomaly detection uses a statistical profile to define a single class of data, regarded as normal behavior, coupled with a threshold selection procedure to define anomalies. Anomaly detection does not miss attack patterns because it does not use attack signatures. However, in practice it has too many false positives to be useful. The difference between normal behaviors and anomalies can be difficult to define [26]. For example, user A almost never creates a file, but when one day he creates a file, this action will be identified as an anomaly, even though the user is creating a file to store his own works. This “anomaly” is in fact a false alarm.

By contrast, specification-based intrusion detection is based on conformance to specifications. It emphasizes detection of intrusions as they occur, instead of depending on anomaly definition, or the static data comparisons with the known attack patterns. It eliminates delay in accessing and filtering the log data. Also, it rules out any dependence on the log data as log data could be altered by the intruder before it is logged, or could be missing or inaccessible due to certain other related crucial internal operations [43]. Furthermore, an attacker could inhibit the collection of further information. On the other hand, intrusion patterns can also be modified, as seen in the mutations shown in computer viruses. Therefore, an ideal intrusion detection system needs the ability to detect new attacks.

In recent years, papers have suggested real-time specification-based intrusion detection instead of the offline signature-based approach. Real-time intrusion detection enables IDSes to identify exploitations and to react as the attack progresses. Distributed intrusion detection systems, especially with the interest-base approach proposed by Gopalakrishna [17], introduce the use of intrusion detection agents and components placed in different locations to detect attacks at different portions of the network [35, 45]. The kernel model proposed by Crosbie and Kuperman [13] also introduced the use of a template as a reference for unknown attack detections.

In 1994, Calvin Ko, George Fink, and Karl Levitt introduced a specification language to look for violations of security properties during executions [21]. The specification language allows the use of parameters in order to specify different allowable scenarios based on different states. To enforce these specifications in real-time intrusion detections, using the system kernel is a viable approach. In 2001, Mark Crosbie and Benjamin Kuperman suggested an approach in IDSes called the “Building Block Approach”, in which a minimal but adequate intrusion detection architecture is added to the kernel. Thus the kernel examines the system calls and the corresponding parameters to detect if there is any malicious usage [13]. The paper also outlines a template for determining whether there is any malicious call made during any moment. The template mentioned in the paper is primitive because it was an initial proposal to an innovative design. Refinements are needed to provide a more complete scheme for the template without sacrificing the overall performance inside the kernel. Using a schema based on a vulnerability taxonomy will improve the template because the schema represents the structure of the vulnerability database which, according to the taxonomy, is classified unambiguously. A vulnerability taxonomy is scalable with respect to the number of vulnerabilities, as demonstrated in the models proposed by both Taimur Aslam [2] and Ivan Krsul [24]. Besides, since in general the database is independent of the vendor and version of the operating system, its data can be referenced in any operating system as long as its client program supports that particular operating system. This means that a vulnerability database will not be affected by any change, update, or upgrade in the operating system.

There are several existing sites that store vulnerability data, some of them private and some of them public. Some of these sites simply provide vulnerability data storage and descriptions, while others have additional classification information. They serve various purposes such as penetration analysis, code analysis, and auditing. Among them, the regularly maintained ones provide a helpful data source for computer security researchers to obtain up-to-date information about the vulnerabilities. Ivan Krsul lists some in his thesis [24]:

1. The CVE site is maintained by MITRE [30], and keeps records of each vulnerability, its description, and other references information.
2. The Computer Emergency Response Team (CERT) maintains a database [12].
3. Security Focus Online stores vulnerability data according to vendor, product or technology name, keyword, impact, Bugtraq ID number, and CVE number. It also offers discussion forums, other users’ suggestions, and solutions if available, for the reported vulnerabilities [39].

4. Open Source Vulnerability Database (OSVDB) is an independent and open source database created by and for the community. Their goal is “to provide accurate, detailed, current, and unbiased technical information” [33].
5. National Institute of Standards and Technology’s ICAT metabase has a searchable index of information on CVE-compatible vulnerabilities [19].
6. The database in CERIAS from Purdue University uses a model proposed first by Taimur Aslam and later enhanced by Ivan Krsul [11].
7. The University of California at Davis maintained a vulnerability database based on the model defined by Matt Bishop.

1.3 Objectives

One of the goals for this thesis is to propose a new “intrusion prevention” model merging both intrusion detection and vulnerability database. A security mechanism aimed at prevention is more effective than detecting known attacks because prevention blocks all attacks. So, just as we have to prevent the known vulnerabilities from being exploited, we also want to detect any new, exploitable vulnerabilities. In the proposed model, a vulnerability database keeps current vulnerability information obtained from search engines or other sources. The schema for the vulnerability database serves as the guidelines for the specifications inside an intrusion detection system, allowing the IDS to query the database. The IDS can then check whether any of the vulnerabilities is being exploited during the current transaction. Since the IDS references the vulnerability schema, the schema must be suitable to this purpose. Part of this thesis proposes a revised version of the Aslam/Krsul classification model. This suggests a new direction towards better intrusion detection.

1.4 Thesis Organization

The organization for the thesis is as follows. Chapter 2 reviews the background of vulnerability analysis, including a discussion of some existing vulnerability taxonomies. Chapter 3 explains the concepts in the integrations of a vulnerability database into intrusion detection systems. It also illustrates the relations between a vulnerability database and the specification language used for intrusion detections. Chapter 4 proposes modifications to the model from Aslam and Krsul appropriate for use with an intrusion detection system. It goes on to illustrate how the vulnerability database can be efficiently used together with a specification language for intrusion detection systems. Chapter 5 describes the use of the vulnerability schema on specification-based distributed intrusion detection systems. Chapter 6 concludes this thesis and offers questions as directions for possible future research.

Chapter 2 Vulnerability Analysis Concepts

2.1 Detections

A fundamental approach in security is *detection*. Detection can be either dynamic, where one tests a program by executing it, or static, which involves semantic and syntactic analysis of source code. The decision between whether the detection should be dynamic, static, or a mixture involves trades-offs between accuracy and efforts. *Automated detection* is any effective detection method performed with little to no human interaction. It can be divided into two parts: the automated *detection of intrusions*, as done by an *intrusion detection system* (IDS), and the automated *detection of vulnerabilities*, as done by a vulnerability search engine.

Bugs are the errors inside computer systems and programs. It is impossible to design and implement error-free software or systems, especially when the software or systems will be modified in the future. However, various techniques and tools can analyze software and systems for bugs such as memory leaks [16]. Hence, some bugs that lead to errors can be found. These bugs should be reported back to the developers so they can fix them.

In typical software engineering practice, testers are responsible for reporting bugs and suggesting functional improvements before any upgrade is made by the developers. However, security-related bugs do not introduce lack of functionality, which would directly affect use of the system. Instead, security-related bugs introduce *weaknesses* during the executions [14], and, as mentioned before, these weaknesses are called *vulnerabilities* or *security holes*. Examples include unauthorized access, unprivileged consumption of privileged resources or race conditions, etc. Vulnerabilities are errors in programming, configurations, and operations [2], as a result of either poorly or incorrectly designed or implemented security implementations or conflicts within the implementations. The determination of whether a bug is a vulnerability also depends on the precise security policy of the analyzed system.

2.2 Vulnerability Analysis Overview

A formally verified computer system can be mathematically described by a *formal top-level specification* (*FTLS*). The system is proved to be consistent with the security policy once we mathematically verify that it satisfies the FTLS at all times [8]. Unfortunately, most systems cannot be described mathematically. Therefore, *vulnerability analysis* comes into the picture. In vulnerability analysis, a set of classifications allow the vulnerability data to be categorized into useful schemes [8] such as signatures for intrusion detections, or into categories describing environment conditions necessary for an attack. Vulnerability analysis schemes include *RISOS* [1] and *Protection Analysis* (*PA*) [6, 32].

In Bishop's scheme [8], assume a given vulnerability $v = \{v_1, v_2, \ldots, v_n\}$ has a set of characteristics denoted as $C_v = \{C_{v1}, C_{v2}, \ldots, C_{vn}\}$. For any combination of vulnerabilities to be exploited at the same time, the intersection of all of their respective characteristics set should result in a non-empty set, i.e., these vulnerabilities should have at least one common characteristic in order to be exploited at the same time. For example, for a race condition, the two possible characteristics are that two system calls reference a file by name, and between the calls, the binding of the file name is changed. Negating one characteristic will simultaneously eliminate the exploitability of all vulnerabilities having the characteristic [8]. One effective method of describing a characteristic is to use a *specification language*, which allows the definitions of intrusion specifications based on vulnerability classification. This will be explained in detail in the next two chapters.

2.3 Vulnerability Categorization

One goal of vulnerability analysis is to identify and classify vulnerabilities before they are exploited by the attackers. Data about vulnerabilities must be organized to be useful for characterizing faults and designing solutions [3]. How the vulnerabilities are classified depends on how the data is to be referenced or used. In addition, having an organized, systematic classification of vulnerabilities can avoid data inconsistency and ambiguity. Therefore, various domain-specific vulnerability classifications have been proposed by Rubey (1975) [38], Potier (1982) [36], Bezier (1983) [5], Weiss (1985) [54], and Knuth (1989) [20]. The following sections describe three general classification schemes.

2.4 Landwehr Classification Scheme [25]

Landwehr, Bull, McDermott, and Choi from the Naval Research Laboratory proposed a taxonomy based on the observations that most software failure histories were not documented. They studied about 50 security flaws described in the literature. Their proposed classification had three general categories: *Genesis*, *Time of Introduction*, and *Location*. *Genesis* refers to the introduction of each vulnerability, whether it is intentional or not, and if it is intentional, whether it is malicious or not. It further classifies the intentional vulnerabilities into one of the six subclasses or the unclassified “other” class, while the unintentional ones are classified based on the categories adopted in RISOS [1].

Time of Introduction and *Location* record the stage of software development and where during the program execution the vulnerability is introduced. An obvious weakness of this classification is that if the circumstances of the introduction are unknown, it is usually not clear whether the introduction of the vulnerability is intentional or not, and whether it is malicious or not. Another difficulty is correctly identifying the time of introduction of the vulnerability because different parties may disagree when the vulnerability is introduced. Similarly, such controversy also affects determining where the vulnerability is introduced. This is because, based on this scheme, the factors for determining a vulnerability include its nature of exploitation, time of introduction, and where it is introduced. Different definitions of a vulnerability based on nature of exploitation may involve conflicting times and places of introduction.

2.5 Aslam’s Taxonomy Scheme

In his Master’s thesis [2], Taimur Aslam presented a taxonomy for the vulnerability database in the COAST Laboratory at Purdue University. This database had 49 documented UNIX security faults collected from various resources, including the Computer Emergency Response Team (CERT), mailing lists, and literature surveys. The purpose of his proposal was to create a systematic scheme to classify faults and to avoid ambiguity by placing each fault into one category. His taxonomy excludes classifications of any non-software-related faults, and focuses only on software implementation and operation. He defines three categories, namely *operational*, *environmental*, and *coding* faults. Configuration and permission errors are classified as operational faults. Environmental faults are those caused by individual functionally correct components interacting together incorrectly. Any error caused by poor programming such as buffer overflow and race condition is a coding fault. However, Aslam’s model does not consider the case where one vulnerability leads to another, different vulnerability. Two examples illustrate this weakness:

1. A program execution calls two components that are individually functionally correct but have interaction errors when called with a particular set of parameters. The decision procedure presented in the taxonomy classifies this error as an environmental fault because of the interaction error between the two components. However, since the fault is introduced in the source code of the components, this is also a coding fault. The source codes of the components fail to handle the error condition introduced when the components interact with each other using these parameters.
2. A packet filtering firewall's vulnerability allows attackers to upload a program. This program can compromise the host protected by the firewall by allowing unauthorized access to the host resources and may result in a DoS attack to other users. This is an example of a coding vulnerability. The code allows an open window because the code “opens” the window upon meeting the conditions that the incoming request in the header is legitimate. This leads to another coding fault when additional malicious source code attached to the end of the request was compiled and executed in the compromised host. If the resulted program grants additional access privileges to the malicious user, the host becomes environmentally vulnerable because the program, which is functionally correct, is incorrectly interacting with the access list of the system.

2.6 Krsul Classification Scheme

Ivan Krsul extended Aslam’s taxonomy in his PhD thesis [24]. In addition to the three categories that Aslam’s model adopts, Krsul’s model adds a new category for all unidentifiable vulnerabilities, maintaining the scalability of the taxonomy itself. In addition, his model includes other important information like impacts, exploitations, and references, etc. The additional information is extremely useful in vulnerability research because it allows developers or administrators to monitor the vulnerabilities. An important feature in his taxonomy is the realization of impacts, which defines both the immediate and ultimate results upon the exploitation of the vulnerability. Sometimes when a vulnerability occurs, it will also open the door to another vulnerability. Consider a packet filtering firewall that has to rely on the routers to permit data flow based on the packet information. When the packet is valid, the router will “open its window” for the packet to continue its transmission. However, this “open window” is also a vulnerability itself because it allows a valid packet to go through. However, there is no guarantee that the data buffer inside a valid packet does not have an overflow problem. The buffer overflow vulnerability in a valid packet becomes an impact caused by the “open window” vulnerability in the firewall. The realization of impacts caused by a vulnerability allows direct, immediate preventions on residual vulnerabilities without waiting for the usual precautions to start only upon when they are detected. This makes both the detections and preventions more effective.

Chapter 3 Applying Vulnerabilities in Intrusion Detections

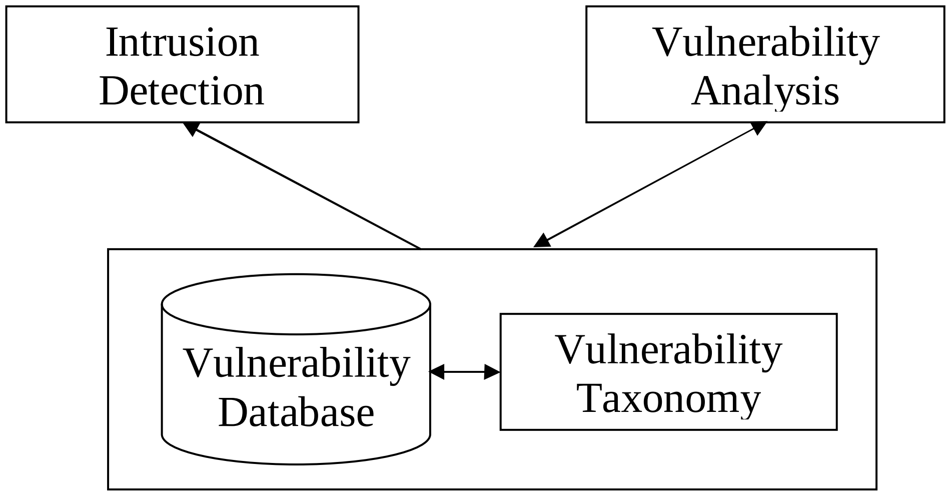

Figure 1. Intrusion Prevention – A High-Level Infrastructure

3.1 Integrating Intrusion Detection and Vulnerability Analysis

Figure 1 shows a picture of an organization in which a vulnerability database responds to queries from an intrusion detection system. This extends the current intrusion detection model to take advantage of the vulnerability data in the database. It prevents any new attack from exploiting any vulnerability that an IDS may not be able to detect based on only the attack pattern known to the IDS. A vulnerability database is added to the model to maintain the vulnerabilities data on the systems being monitored. Therefore, as each vulnerability is discovered, it will be classified according to the adopted taxonomy and added to the database, so that the vulnerability data is as current as possible. Also, any previously unknown attack exploiting any of the newly discovered vulnerabilities will be detected.

Given an attack pattern, a specification-based intrusion detection system can use a specification language to describe how the pattern can be negated. Whenever a vulnerability is discovered, its exploitation method will be classified based on the adopted taxonomy and saved into the vulnerability database as part of the vulnerability data. The exploitation method describes the attack pattern used in order to exploit this vulnerability. The negation of this attack pattern will then be defined using specification language. Therefore, using the vulnerability database helps the specification language define the negation of any attack pattern. Since all or indicated processes that can be exploited by the same vulnerability must follow the same negation, specifications can be defined based on the data queried from the vulnerability database. In this way, vulnerability classification helps define the specification for each process. The IDS can then obtain and apply this specification to detect the vulnerability being exploited.

The schema for the vulnerability database presents a structural and relational data source that allows storage, queries, maintenance, and expansion [29]. The database stores the vulnerability data gathered from the results of various vulnerability tests and other external information such as newly-applied patches and specifications, etc. The IDS can query the data to extract vulnerability information about a specific system call/module. For example, suppose that a system call is known to have vulnerabilities v_1, v_2, …, v_n, and the database has already stored the data for these vulnerabilities. The specification of this system call can then be defined based on this data. In addition, the schema for the vulnerability database used in this model, which will be detailed in Chapters 4 and 5, includes the specification for each classified vulnerability. Therefore, we can also obtain a list of system calls that can be exploited by a given vulnerability so that we can add the specification of this vulnerability to every system call in the list. For instance, system calls m_1, m_2, …, m_n have a specific vulnerability. Querying the list of system calls exploitable by this vulnerability should return a list $M = \{m_1, m_2, \ldots, m_n\}$. As the queried vulnerability data describes how and when this vulnerability is being exploited, it enables the specification-based intrusion detection system to determine which system calls can be exploited by this vulnerability. Let $M' \subseteq M$ be the list of system calls that have not included this vulnerability in their specifications yet. After the query, we can add the negation of the attack pattern of this vulnerability to the specifications of all of the system calls inside M' because the query returns M and $M' \subseteq M$. Since each vulnerability in the database is already classified into a unique, unambiguous category, using the classification based on a taxonomy like Aslam's or Krsul's also makes its corresponding specification unambiguous. In essence, the vulnerabilities in the database produce the guidelines to enforce the security policy of the system.

Another advantage of having a vulnerability database inside this model is that the data is accessible to all components that need to reference it. Issue of *who* should have access to the database is not part of the module. While this issue is important, it relates to the database itself and not to the proposed model. Thus, we will not explore this issue any further.

3.2 Specification-based Intrusion Detections

When an application program is to be executed, certain shell process(es) inside the operating system will be called upon to start the execution. Any process involved in beginning this execution becomes an active entity, while the application program by itself is a passive entity. Something needs to exist in order to monitor the execution of this application program, especially when it is performed in a distributed or concurrent fashion under a given set of system environments. A program execution is characterized by a set of attributes and sequences of events, and a monitor, such as an intrusion detection system, can determine the state of the program execution based on these attributes. For example, the *daemon* is a UNIX request-handling procedure that is invoked whenever an alteration, an addition, or a deletion or other event occurs, and its purpose is to decide what to do with the event(s).

The requirements for confining the state of execution to a set of allowed states, given certain input parameters, are the *specifications* of the program. Specifications are security definitions describing a set of allowed states determined by the security policy. Specifications can be defined based on the vulnerability data in the vulnerability database. As described in 3.1, vulnerability data serves as the guidelines for the security specifications because these specifications are the conglomeration of the negations of the vulnerabilities stored in the vulnerability database. These specifications are expressed by means of a programming or scripting language, including the *specification language* defined in Calvin Ko's thesis [21]. Specification language expresses the specifications in such a way that a list of allowed operations or processes is defined during a program execution based on the input state, as long as the initial state passed to the program belongs to the set of allowed states for this program. For example, if writing a file requires that the current state be in "close", then the following specifications describe the write action:

1. The allowed processes before the write action should include
    a. Opening the file for writing and the state of the file is in “open” state
    b. Assigning a file handle to the file before writing to the file
2. The “open” state has to be maintained throughout the write action before the file is closed.
3. Before the next write action can occur, the file has to be closed and the state of the file must return to “close” state.

3.3 Integrating the Vulnerability Database and the Specification Languages

Unknown attacks can be detected if specification-based detection is adopted. The specifications describe the allowed behavior of security-critical programs. Attacks violate these behaviors.

Execution event sequences are the sequences of events corresponding to the operations performed by a distributed process. Traces can be modeled mathematically. For example, given a distributed process $p = \{p_1, p_2, \ldots, p_n\}$ where p_i is a process, the execution trace V_p is the merge of all V_{pi}’s, where each V_{pi} is the individual process trace. Traces can be classified into *System Traces* and *Process Traces* [23]. System traces are sequences of events for the whole distributed system, and process traces are sequences of events for process(es) within the system. Thus, V_p is a subtrace of the system trace V, and V_{p1} is a subtrace of V_p.

A specification language focusing on program behaviors applies program traces and *parallel-environment* grammars (PE-grammars) to specify trace policies. The PE-grammar adopted in specification language defines a formal language for program operations. Like other languages, it has a set of terminals, a set of rules called *hyperrules*, and a start expression. It also has environment variables to keep track of the state of the system. Instead of using static rules directly to define a language, the PE-grammars parameterize these rules, as a template for the dynamic generation of actual production rules during the parsing stage by replacing the parameters with actual environment values. For example, consider the following piece of specification template:

```
Codes                                                    Line
-----                                                    ----

Environment Variables                                    1
ENV int E = 0;                                           2
LOCAL ENV int L = 0;                                     3

Start Expression                                         4
SE: <progA> || <progB>                                   5

Hyperrules                                               6
<progA> -> <writeA, E>.                                  7
<writeA, 0> -> <openA> <closeA> { E = E – 1; }.          8
<openA> -> open_A { E = E + 1; L = 1; }.                 9
<closeA> -> close_A.                                     10

<progB> -> <writeB, E>.                                  11
<writeB, 0> -> <openB> <closeB> { E = E – 1; }.          12
<openB> -> open_B { E = E + 1; L = 1; }.                 13
<closeB> -> close_B.                                     14
```

In terms of high-level illustration, this piece of sample specification tries to prevent two programs from writing to an opened file at the same time. The environment variable E keeps track of the state of a file. Lines 7 to 10 are the hyperrules for program A, where it can write to a file if E is 0, as shown on line 8. Likewise, lines 11 to 14 are the same hyperrules for program B. Therefore, these hyperrules try to prevent a race condition vulnerability during the file writing process, which can be exploited in both program A and program B. Through the environment variable E, the specification parameterizes the PE-grammar because the value of E is assigned based on the environment, and the hyperrules depend on the current value of E, which changes upon opening or closing a file. Hence, different write permission will be enforced based on the current value of E. The concept of using environment variable E is analogous to the use of a variable in programming language [21]. To generalize this specification so that the same hyperrules will be applied to all programs that write a file, we first make a query to the vulnerability database to return a list of modules having this race condition vulnerability. Then we assign the same hyperrules for this race condition vulnerability to all the modules in the queried list so that they will all have the same negation of the vulnerability. This integration of vulnerability database with specification language helps prevent a vulnerability from exploiting any system call that was previously found to be exploitable by this vulnerability. It allows all vulnerabilities of the same category, subcategory and exploitation method to be described by the same defined hyperrules. Further explanations on how this specification can be generalized will be discussed in 4.4.

In a second example, assume that query result for vulnerability v shows that it can be exploited in processes p_1, p_2, …, p_n, and assume that specification s_v is the set of hyperrules needed to prevent vulnerability v from being exploited. Then we can use the specification language to include s_v in the specifications for all of these processes. For example, consider the following generic specification which will further be illustrated in 4.5:

```
Codes                                                    Line
-----                                                    ----

SPEC <(?, Generic, U, H)>

      ENV int CREATTMP = 0;                             1
      ENV int PID = getpid ();                          2

      SE: <Generic> || <other>                          3
```

```
            <Generic> -> <init> <mktemp> <rest>.           4
            <init> -> <not_mktemp> <init> | Nil.           5
            <rest> -> any_op <rest> | Nil.                 6
            <mktemp> -> open_tmpfile-PID { CREATTMP = 1; }. 7
            <not_mktemp> -> not_open_tmpfile-PID.          8
            <other> -> <vop, CREATTMP> <other> | Nil.      9
            <vop, 0> -> not_chgtmp.                        10
            <vop, 1> -> any_op.                            11

      END;
```

In this example, v is the race condition vulnerability allowing a privileged file to be symbolically linked by the intruder before another program changes the contents of this file, allowing the intruder to access the information inside this file. Therefore, s_v is the above specification. The environment variable CREATTMP indicates whether the temporary file, which uses PID as its identification, is currently being accessed or not. If the file is not accessed by any process, CREATTMP is 0. CREATTMP will be set to 1 upon access, as shown on line 7. The hyperrule on line 4 first executes the hyperrule on line 5, which conducts a test defined on line 8 to check if the file is currently being accessed or not. If the file is being accessed by another program, the test on line 8 will fail and so will line 5. As a result, line 4 will not move on to the hyperrule defined on line 7, i.e., the program will not be able to access the file. The hyperrules prohibit a file to be accessed or renamed if another program has already been accessing it. Any system call exploitable by vulnerability v will have this generic specification assigned to it by replacing the string “Generic” in the specification with the actual name of the system call. Specification for any vulnerability, therefore, also becomes unique because there will not be conflicts any more. This prevents specifications from being ambiguous or contradictions within the specification for any vulnerability because any of these problems can affect the overall effectiveness of intrusion detection. Further explanations on how to integrate the vulnerability database and the specification language will be discussed in the next chapter.

3.4 Benefits

This model uses a vulnerability database. The database can be updated as new vulnerabilities are found. Hence, the vulnerability database in this model allows the specification-based IDS to use specifications about the most up-to-date vulnerability information. The IDS can access data from the database to analyze the targeted module or system call. This model also enables the corresponding specifications in the IDS to be updated through the use of specification language whenever new vulnerabilities are discovered.

Chapter 4 Interoperable Vulnerability Schema for IDSes

4.1 Considerations for the Schema

As mentioned in Chapter 3, the data in the vulnerability database helps defining specifications for the intrusion detection system. The vulnerability data can also be used as reference information for the administrators and the developers. They can examine the details of the discovered vulnerabilities inside the current version of the systems being monitored. The vulnerability database should reflect the following considerations:

1. The schema has to be unambiguous to ensure that the IDS avoids producing ambiguous alerts. If the classification is ambiguous, a vulnerability may fall into one class in one system and another class in another system. Assume that a vulnerability v is classified to be in the class V_1 in one IDS and in the class V_2 in another IDS due to ambiguity. When the specification of each vulnerability in the class V_1 needs to be updated in both IDSes, a query is first made in each IDS to obtain a list of vulnerabilities belonging to the class V_1 in that IDS. Due to the ambiguity, v is not in the queries list in the IDS that classifies v as a vulnerability in the class V_2. Therefore, v's specification will not be updated in that IDS. As a result, both IDSes will have different specifications for v after the update. Besides inconsistent specifications, ambiguity in classification can also result in inconsistent alerts. For example, using the same ambiguous example, when v is being exploited, one IDS will issue a V_1 alert because v is in the class V_1, while in another IDS a V_1 alert will not be issued because v is not in the class V_1. Let v have the same "open window" vulnerability in the packet filtering firewall

mentioned in 2.6. One IDS may classify the vulnerability as a coding-related vulnerability because of root access due to buffer overflow caused by some program execution during the “open window” period. On the other hand, another IDS may classify the vulnerability as an environmental vulnerability because of the existence of the “open window”. Because of ambiguous classification, when v is being exploited, one IDS will raise a coding-related vulnerability alert while the other IDS will raise an environmental vulnerability alert.

2. The taxonomy should not use any subjective field, such as severity rankings or references, etc., for vulnerability classification. Subjective information is for internal reference only by the vulnerability analysts and researchers for investigation purposes. If the system administrator needs to prioritize attack data, this should be done externally to the vulnerability information.
3. The schema should not deviate too much from any unambiguous model such as the one proposed by Aslam and Krsul. This means that while some variations will be made in order to be more suitably used by an intrusion detection system, the fundamental classification scheme should still be based on a taxonomy using unambiguous classification. The classification scheme adopted by an IDS has to be unambiguous because the IDS requires consistent specifications for modules having the same vulnerabilities. A classification scheme similar to the one used by Aslam/Krsul is recommended. Since the Aslam/Krsul taxonomy avoids ambiguity in classifications, it meets the needs of IDS. Note that the *impacts* field in Krsul’s scheme is crucial because it provides information on residual vulnerabilities. Another important reason for the schema to be based on the

Aslam/Krsul taxonomy is that this taxonomy is being adopted in many current vulnerability researches. The schema adopted in the IDS should be based on a taxonomy that is familiar with the vulnerability researchers, since they may need to regularly investigate the vulnerability data and update the patch information inside the database.

4. For a vulnerability with a given category/subcategory, there may be more than one exploitation method. For example, a race condition vulnerability may be exploited by renaming a file after writing but before calling *chown* or *chmod*, as occurs in *rdist*, or may be exploited by symbolically linking of a privileged file before the file is "re-created", such as in *binmail*. This shows that different exploitation methods for a single vulnerability can generate different signatures. The scheme will be similar to the *Detailed Information About Exploitation* session in Krsul's scheme, except that the subjective items like *ease of exploit* and *complexity of exploit* are not necessary. Exploitation methods provide another hierarchy in the classification criteria, in addition to category and subcategory, needed to define unambiguous specifications.
5. Specifications are required for every class of category/subcategory/exploitation combination so that each class uses the same specifications. This maintains the desired consistency for each set of vulnerabilities, and allows any newly discovered vulnerability of the same class to adopt the same specifications.

Sections 2.5 and 2.6 mentioned that the Aslam and Krsul schemes enable vulnerability classification to be unambiguous, which is critical in defining specifications for use in an intrusion detection system. However, the taxonomy proposed by Aslam and extended by

Krsul has some categories that are not useful for intrusion detection because they are neither necessary nor updated by the system administrators. For example:

1. The field *Information Regarding the Source of the Information* provides details on the source of the information regarding the vulnerability. While the existence of a vulnerability and its information is relevant, the source where the information on a vulnerability can be found is not. An IDS is supposed to reference the vulnerability data to find and prevent potential attacks. Its major task is simply detecting and reacting to the detected attack. The source of information for the vulnerability data helps researchers in ranking and prioritizing various vulnerabilities. However, ranking and prioritizing vulnerabilities are not relevant inside the suggested model because the task of an intrusion detection system is to try to prevent any vulnerability from being exploited, no matter what the ranking and priority are.
2. Similarly, the field *References* provides additional references regarding the found vulnerability, such as which websites have further descriptions on the vulnerability. This is for human use only.
3. The fields *ease_of_exploit* and *complexity_of_exploit* inside the schema field *Detailed Information About Exploitation* are subjective rankings on how easy the vulnerability can be exploited.

Generally speaking, an intrusion detection system detects an attack by examining the current operation or sequence of operations. The IDS needs to prevent any vulnerability arising from these operations from being exploited.

The IDS will use the following information:

1. System information, such as name, version and vendor of both the server and the operating system, together with the current environment of the system when the vulnerability is detected. This information identifies under what situation(s) a vulnerability can be exploited. It helps the system administrators determine the appropriate patch(es) to be uploaded to the system.
2. Identification for the vulnerability, including information like what category, subcategory and exploitation method(s) it belongs to. For all vulnerabilities that belong to the same category and subcategory, the exploitation method for each of them may be different. Therefore, we need to classify the vulnerability with one more hierarchy level such that for each classified vulnerability, it belongs to a unique exploitation method under one subcategory of a certain category. This additional hierarchy level is necessary in defining the negation of each vulnerability because the specification for an operation is a conglomeration of unambiguous negations of all vulnerabilities found in that operation. In addition, Chapter 3 has already illustrated the importance of unambiguous vulnerability classification in the generation of a unique vulnerability negation.
3. Information regarding how the vulnerability is exploited, including which set of system calls are used, and in what module or program the calls occur. The exploitation method is also critical to specification definition because the vulnerability arises when a sequence of system calls arises with a certain set of parameters, so we need to make a specification such that this sequence can never occur with this set of parameters. Therefore, the IDS needs to allow queries on the

exploitation method so that specifications can be added to these system calls to help prevent the vulnerability from being exploited when this set of parameters is passed to the system calls.

4. Impact caused by the vulnerability, for example whether it will open the door to another vulnerability. This enables the IDS to identify any residual vulnerability that can be exploited as a result of the exploitation of the first vulnerability. The IDS can thus prevent any disastrous chain of attacks. For example, in the “open window” vulnerability mentioned in section 2.6, conventional signature-based IDS may not be able to detect any residual attack if none of the residual attacks forms an attack pattern known to the IDS. However, with the existence of the *Impact_ID* field in the vulnerability database schema to be shown later in this chapter, the IDS in this model is able to detect any possible malicious code execution in addition to the “open window” vulnerability.
5. Patch information, which needs to be updated regularly by the security officers. Security officers not only manage the IDS but also regularly update the system with patches to address reported vulnerabilities. The patch information identifies how the vulnerability is addressed. It may also provide additional descriptions on certain vulnerabilities. These descriptions can serve as further references on how to handle these vulnerabilities within the system in this suggested model. The references mentioned here are different from the references mentioned earlier. Although these references here are not for detections inside the IDS, they are for maintenance purposes in the IDS. The additional descriptions here may be about the vulnerability identification, the category it belongs to, the corresponding

environment, how it is exploited, and the resulting impact(s). The main purpose for the patch information inside the database is for maintenance. The administrators need this information for reference in the future so that they know whether and what they have applied any appropriate patches to the IDS. On the other hand, the references mentioned much earlier are mainly about subjective information like rankings and priorities.

6. The specification for each vulnerability, which needs to be referenced by both the IDS itself and the system administrators and programmers. System administrators and programmers also need to update the specifications when necessary. The IDS references these specifications by obtaining every vulnerability that belongs to a particular combination of category and subcategory. The negation of a vulnerability's specification is a general specification for all processes, in addition to any particular per-process specifications.

4.2 IDS-compatible Vulnerability Schema

The proposed vulnerability schema adopts the format of the Aslam/Krsul schema with some adjustments to make it more applicable to an intrusion detection system. The vulnerability ID naming is based on the CVE reference for the benefit of system administrators or programmers. This reference is used in many existing public vulnerability databases. The schema for the tables used in the IDS-compatible vulnerability database is as follows.

Field Name	**Field Description**
Vulnerability_ID	The CVE-format vulnerability ID
Category_ID	Aslam category (i.e., coding, operational or environmental)
Subcategory_ID	Aslam-format additional subcategory within a category
Patch_ID	ID for the patch, if any
Date_found	Date when this vulnerability is first discovered
Date_addressed	Date when a patch starts to apply on the vulnerability
Description	Manually added details about the vulnerability

Table 1. Vulnerability_Identification

Field Name	**Field Description**
Category_ID	Aslam category (i.e., coding, operational or environmental)
Subcategory_ID	Possible additional subcategory within a category
Exploit_ID	ID for the exploitation method
System_ID	ID for the affected system
Description	Manually added details about the subcategory
Specification	Specification for each classification

Table 2. Vulnerability_Classification

Field Name	**Field Description**
System_ID	ID for this system
Server_Name	Name of the server
Server_Version	Version of the server
Server_Vendor	Vendor for the server
OS_Name	Name of the OS
OS_Version	Version of the OS
OS_Vendor	Vendor for the OS

Table 3. System

Field Name	**Field Description**
Vulnerability_ID	The CVE-format vulnerability ID
Impact_ID	The CVE-format ID for the subsequent vulnerability, if any
Description	Manually added details about the impact and/or consequence

Table 4. Vulnerability_Impact

Field Name	**Field Description**
Vulnerability_ID	The CVE-format vulnerability ID
Exploit_ID	ID for the exploitation method
Module_Name	Name of the module or program
Module_Version	Version of the module or program
System_Call	System call in which the vulnerability is exploited
Description	Manually added details about the nature of the exploit

Table 5. Vulnerability_Exploit

Field Name	Field Description
Vulnerability_ID	The CVE-format vulnerability ID
Name	Environment name
Value	Environment value when vulnerability is exploited
Description	Manually added details about the environment

Table 6. Environment

Field Name	Field Description
Vulnerability_ID	The CVE-format vulnerability ID that the patch is for
Patch_ID	ID for the patch
Description	Manually added details about the environment

Table 7. Patch

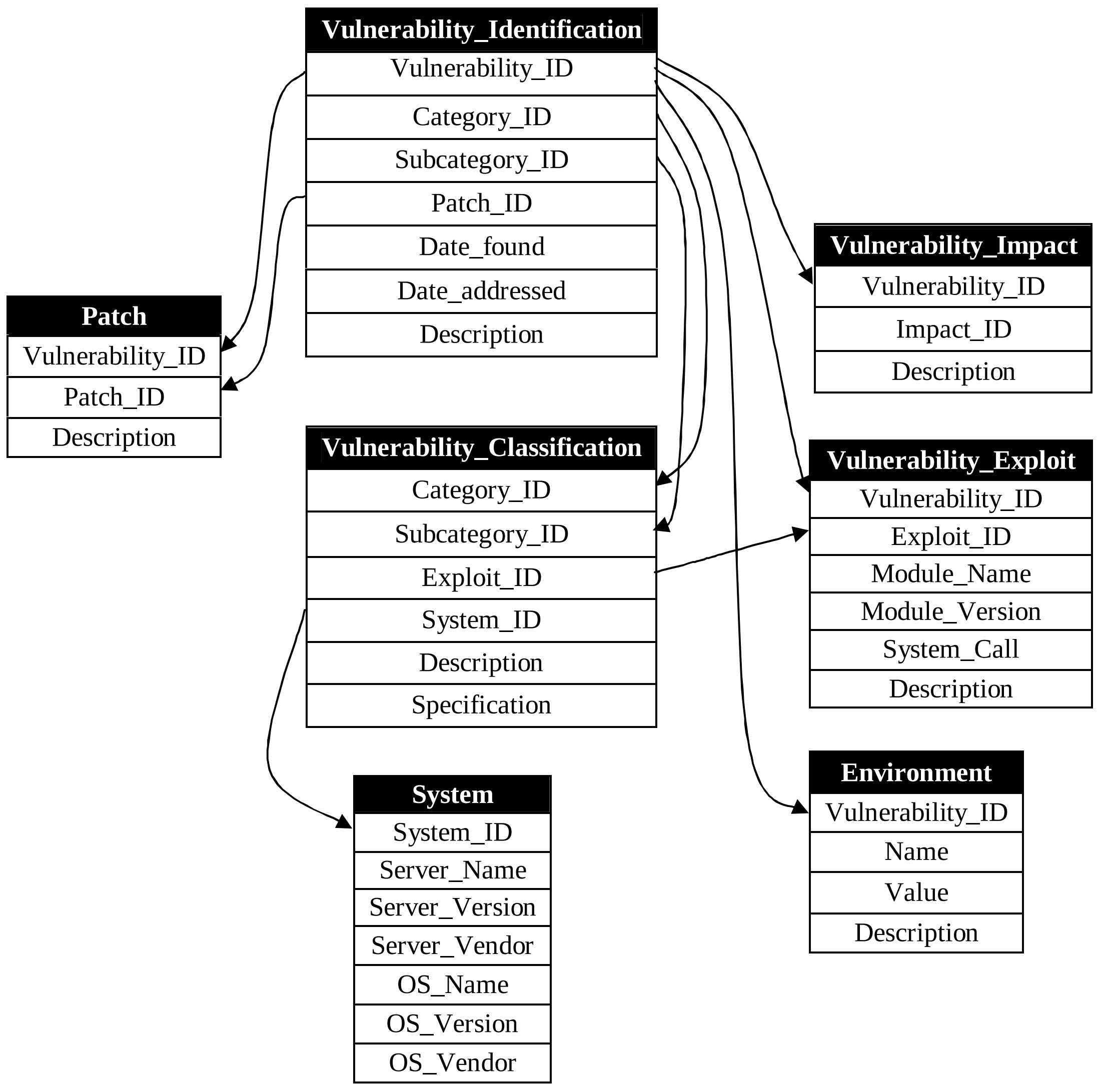

Figure 2. Relation Diagram among the tables in the vulnerability database

Figure 2 illustrates how the tables in the proposed schema relate to each other. Some of the fields in the tables, such as the date_addressed field in the vulnerability_identification table and the entire Patch table and System table, may only be updated manually by system administrators. Even though the IDS does not directly use them, those fields provide more user-friendly information about when and how the vulnerabilities are addressed within the system, so that any future detection will be more effective. In addition, some tables and fields of other tables need to be initialized to make the vulnerability database and IDS useful over the entire network right after the deployment of the vulnerability database. For example, the whole *System* table has to be initialized with system data so that the servers can report data regarding what system is exploited upon any newly discovered vulnerability.

4.3 Examples on How to Populate and Use the Vulnerability Database

Consider the buffer overflow vulnerability, with “AllowOverride” not set to “None”, in the system call mod_compat_directive of the module mod_ssl inside an Apache module (CVE number CVE-2002-0653) [12, 30], resulting in denial of service against an Apache HTTP server (CVE number CAN-2003-0132). The following sample codes show the basics on how to populate and use the vulnerability database. Let 4 be the System_ID for the system describing Apache 1.3 webserver and Red Hat Linux version 8.2, and let 5 be the Exploit_ID.

First of all, the database needs to have the system data populated, in order to understand the information on the existing systems available in the real world. For example, to insert the data for Red Hat Linux version 8.2 on Apache 1.3 webserver, the SQL source code will be like the following:

```
INSERT INTO System
      (System_ID, Server_Name, Server_Version,
      OS_Name, OS_Version, OS_Vendor)
VALUES (“4”, “Apache”, “1.3”, “Linux”, “8.2”, “Red Hat”);
```

In addition, the vulnerability database needs to include all classifications of vulnerabilities. For the vulnerability used in this example, it can be done in the SQL codes shown below:

```
INSERT INTO Vulnerability_Classification
      (Category_ID, Subcategory_ID, System_ID, Description)
VALUES
      (“3”, “a5”, “4”, “Buffer overflow vulnerability”);
```

Before the IDS can be used, the programmer or system administrator needs to provide the already-available generic specification to each classification of vulnerability. The form of this information is:

```
INSERT INTO Vulnerability_Classification (Specification)
VALUES (“Some Specification”)
WHERE Category_ID = “3” AND
      Subcategory_ID = “a5” AND
      Exploit_ID = “5”;
```

Suppose this vulnerability was exploited by using an unprivileged username. The data for the newly found vulnerability will be added into the Vulnerability_Identification, Vulnerability_Classification, Vulnerability_Impact, Vulnerability_Exploit, and Environment tables, so that the IDS can prohibit the unprivileged user from actually exploiting this vulnerability in the future.

```
SELECT DISTINCT Sys_ID = System_ID
FROM  System
WHERE Server_Name = “Apache” AND
      Server_Version = “1.3” AND
      Server_Vendor = “Apache Software Foundation” AND
      OS_Name = “Linux” AND
      OS_Version = “8.2” AND
      OS_Vendor = “Red Hat”;
INSERT INTO Vulnerability_Identification
      (Vulnerability_ID, Category_ID, Subcategory_ID,
      Date_found)
VALUES (“CVE-2002-0653”, “3”, “a5”, NOW ());
INSERT INTO Vulernability_Category
      (Category_ID, Subcategory_ID, Exploit_ID, System_ID)
```

```
VALUES (“3”, “a5”, “5”, “4”);
INSERT INTO Vulnerability_Impact
      (Vulnerability_ID, Impact_ID)
VALUES (“CVE-2002-0653”, “CAN-2003-0132”);
INSERT INTO Vulnerability_Exploit
      (Vulnerability_ID, Exploit_ID, Module_Name,
      Module_Version, System_Call)
VALUES
      (“CVE-2002-0653”, 5, “mod_ssl”, “2.4.9”,
      “ssl_compat_directive”);
INSERT INTO Environment (Vulnerability_ID, Name, Value)
VALUES (“CVE-2002-0653”, “user_name”, “*”);
INSERT INTO Environment (Vulnerability_ID, Name, Value)
VALUES (“CVE-2002-0653”, “account_type”, “user”);
```

In another example, consider querying all system calls that can result in denial of service attacks against an Apache HTTP server. Based on the previous situation, one of the library calls returned from the query should be ssl_compat_directive.

```
SELECT DISTINCT ve.System_Call
FROM  Vulnerability_Exploit ve,
      Vulnerability_Impact vim,
      Environment e,
      Vulnerability_Identification vid
WHERE vim.Impact_ID = “CAN-2003-0132” AND
      vim.Vulnerability_ID = ve.Vulnerability_ID AND
      vim.Vulnerability_ID = e.Vulnerability_ID AND
      e.Name = “account_type” AND
      e.Value = “user” AND
      vim.Vulnerability_ID = vid.Vulnerability_ID;
```

When the administrator applies a patch for this vulnerability by installing an upgraded version of the module *mod_ssl*, like version 2.8.10, as a fix for the vulnerability, the following updates the vulnerability database:

```
INSERT INTO Patches
VALUES
      (“CVE-2002-0653”, Some_Patch_ID, “mod_ssl 2.8.10 has
      been installed and this version has reported no
      vulnerability”);
```

The next two sections will show more practical examples of applying vulnerability data to IDSes.

4.4 Integration with Specification Language using a General Example

Consider a race condition vulnerability for writing to a file [9]. An attacker can exploit this vulnerability by first running program A to create an arbitrary file. Then he can run program B to create a symbolic link of that arbitrary file to a privileged file like the *passwd* file. As a result, he will be able to alter the password(s) for all account(s) that he wants to get access to [9], before destroying the symbolic link and returning the operations to program A. A simple, generic example in the specification language paper by Ko, Ruschitzka, and Levitt [23] demonstrated how to use the specification language to handle this vulnerability. In that example, program A can write a file if the environment variable E is 0, and it will change that environment variable to a non-zero value upon opening the file. The value of the environment variable remains non-zero until that file is closed by program A. Meanwhile, program B follows the same hyperrules as program A. So, in this case, program B cannot write to the file if A has already opened it. The specification language describes this in such a way as to prevent A and B from writing to the same file. One of them can only write to the file when the other has closed its handle for that file. The specification is as follows [22, 23].

```
Codes                                                 Line
-----                                                 ----

Environment Variables                                 1
ENV int E = 0;                                        2
LOCAL ENV int L = 0;                                  3

Start Expression                                      4
SE: <progA> || <progB>                                5

Hyperrules                                            6
<progA> -> <writeA, E>.                               7
<writeA, 0> -> <openA> <closeA> { E = E – 1; }.       8
<openA> -> open_A { E = E + 1; L = 1; }.              9
<closeA> -> close_A.                                  10

<progB> -> <writeB, E>.                               11
<writeB, 0> -> <openB> <closeB> { E = E – 1; }.       12
<openB> -> open_B { E = E + 1; L = 1; }.              13
```

```
        <closeB> -> close_B.                                 14
```

In the specification, lines 7 to 10 are for program A. If program A successfully opens a file for write access, the global environment variable E will be incremented from 0 to 1. Since there is no rule for <writeA, 1> (E is 1 in <writeA, E>), program A is not allowed to write to that file if this file has already been opened by another program (E = 1). Similarly, program B has the same requirements as shown in lines 11 to 14. This specification handles in a way such that when the environment variable E is set to 1 upon the creation of the arbitrary file by program A, program B will not be able to link and write to that file as long as it stays open.

It is undoubtedly desirable to use a specification language to trace possible vulnerability exploitations within the operation sequences. However, the programmer will sometimes have a hard time coding each vulnerability specification manually if each process inside the program has a lot of vulnerabilities to be addressed, especially when any particular process occurs in several places inside the program. The programmer will also have to spend time to write additional programs if many modules share the same hyperrules. In this case, the programmer has to create a utility program to generate a specification for each of these programs by automating the modifications of the same hyperrules with a different module name. However, the programmer needs to obtain a manually generated list of modules sharing the same hyperrules before he can make the utility program work, and without an organized data storage, it is always possible that the list is incorrect or incomplete. Besides, the more modifications that he needs to change in the list, the more probable that he will make mistakes. An alternative and more efficient way is to integrate specifications with a vulnerability database. In this method, we populate various vulnerabilities in the database, and when the IDS needs to address a particular vulnerability, it can obtain the specification constraining all the functions or system calls related to that vulnerability.

As explained above, the following lines for the hyperrules can be duplicated as follows:

```
<progGeneric> -> <writeGeneric, E>.
<writeGeneric, 0> ->
      <openGeneric> <closeGeneric> { E = E – 1; }.
<openGeneric> -> open_Generic { E = E + 1; L = 1; }.
<closeGeneric> -> close_Generic.
```

The same specifications will then be applied to those modules or system calls sharing the same race condition vulnerability. For example, one of the race condition vulnerabilities in *Samba smbmnt* (CVE-1999-0812) allows local users to mount file systems in arbitrary locations. This is one of the practical examples in the race condition vulnerability that allows a file to be symbolically linked to a system file after its creation. We can use Perl to write a script to automate the replication of the same sample specification for each module resulted from the query. The entire Perl script is shown in Appendix A.1.

To look for all modules that can be exploited by this vulnerability, a query is made to the vulnerability database to retrieve all modules that can be exploited by the vulnerability having the CVE ID of “CVE-1999-0812”. Since the vulnerability classification is unambiguous before a vulnerability is stored into the database, and since only one vulnerability is being looked at in this example, all modules in the queried list must belong to the same vulnerability classification because all of these modules can be exploited by the same vulnerability. With only one set of generic hyperrules for each vulnerability class, a specification of the same hyperrules should be assigned to all modules exploitable by this vulnerability. The generic version of the hyperrules should already be defined in and queried from the *Specification* field of the *Vulnerability_Classification* table in the vulnerability database. In the specification, all of these modules exploited by the same vulnerability should be appended to the list appearing in the Start Expression section. Besides, the same piece of generic hyperrules should be appended to the current specification after replacing every existence of the string “Generic” in the generic specification with the module name. To specify the hyperrules for program A, every existence of the string “Generic” in the *Hyperrules* portion of the generic specification will be replaced by the string “A”, before A’s hyperrules are added to the specification. Similarly, every existence of the string “Generic” in the *Start Expression* portion of the specification will also be replaced by the string “A”. As a result, after the hyperrules for program A are appended to the specification but before those for program B are appended, the specification will look like this:

```
Environment Variables
ENV int E = 0;
LOCAL ENV int L = 0;

Start Expression
SE: <progA>

Hyperrules
<progA> -> <writeA, E>.
<writeA, 0> -> <openA> <closeA> { E = E – 1; }.
<openA> -> open_A { E = E + 1; L = 1; }.
<closeA> -> close_A.
```

After the hyperrules for program B are also appended, the specification will finally be the same as the one shown at the beginning of this section .

4.5 Integration with Specification Language using a Practical Example

This section presents a more practical example on the integration of vulnerability database and specification language. This example focuses on the UNIX module *binmail*. In Calvin Ko's PhD thesis, he demonstrated the use of specification language on four UNIX modules that have well-documented vulnerabilities such as *binmail*. We use the *binmail* module as an example. This UNIX backend mail delivery module, which is responsible for appending a mail message directly to the users' mailbox files, creates a temporary file with an internally specified file name. However, if an attacker knows the temporary file name, then he can create a symbolic link pointing a locally created file to a system file that he wants to modify, such as the *passwd* file [22]. The following is the original specification for *binmail* presented in his thesis.

```
SPEC <(?, binmail, U, H)>

      ENV int CREATTMP = 0;
      ENV int PID = getpid ();

      SE: <binmail> || <other>

      <binmail> -> <init> <mktemp> <rest>.
      <init> -> <not_mktemp> <init> | Nil.
      <rest> -> any_op <rest> | Nil.
      <mktemp> -> open_tmpfile-PID { CREATTMP = 1; }.
      <not_mktemp> -> not_open_tmpfile-PID.
      <other> -> <vop, CREATTMP> <other> | Nil.
      <vop, 0> -> not_chgtmp.
      <vop, 1> -> any_op.

END;
```

As an extension, instead of explicitly naming *binmail*, the same hyperrules can be assigned to any module that calls *open_rwtc*. In the vulnerability category, subcategory, exploitation method and the system ID that *binmail* belongs to, the *Specification* field in the *Vulnerability_Classification* table should have the generic version of the above specification. This is obtained by replacing every occurrence of the string “binmail” with the string “Generic”. Hence, the generic specification stored inside the database is:

```
SPEC <(?, Generic, U, H)>

      ENV int CREATTMP = 0;
      ENV int PID = getpid ();

      SE: <Generic> || <other>

      <Generic> -> <init> <mktemp> <rest>.
      <init> -> <not_mktemp> <init> | Nil.
      <rest> -> any_op <rest> | Nil.
      <mktemp> -> open_tmpfile-PID { CREATTMP = 1; }.
      <not_mktemp> -> not_open_tmpfile-PID.
      <other> -> <vop, CREATTMP> <other> | Nil.
      <vop, 0> -> not_chgtmp.
      <vop, 1> -> any_op.

END;
```

To obtain a list of modules which call *open_rwtc*, a query is made to get all values in the *Exploit_ID* field of the vulnerability database that has the value of the *System_Call* field set to “open_rwtc”. The query result would include the value of the *Exploit_ID* field used for *binmail*. This follows by assigning the specification to every module in the query result by replacing the “Generic” string in the generic specification with the module name. The Perl codes in Appendix A.2 outline all these steps. This specification allows the IDS to detect any symbolic link attempt to a system file whenever any program that calls *binmail* is being processed, and helps prevent any unprivileged write access to any system file.

4.6 Consistency and Error Prevention

As shown in the above two examples, augmenting the vulnerability database with a set of specifications provides a mechanism for consistent specifications for all vulnerabilities in the same classification. It also eliminates recoding the same specification for different vulnerabilities. Since the specifications describe a set of negations of vulnerabilities, the IDS can detect any attack based on the patterns shown in the exploitations.

Chapter 5 Integrations with Specification-Based IDSes

5.1 Integration with Non-distributed Intrusion Detection Systems

As shown in Chapter 4, a vulnerability database can be integrated with a specification-based intrusion detection system. Sections 3.3, 4.4 and 4.5 provided some examples of how to populate and update the vulnerability database given a specific vulnerability. To review, the integration consists of two stages: *initial data population* and *data update*. During the *initial data population* stage, relevant information on all known vulnerabilities has to be inserted accurately into the database so that the analysts can provide precise specifications for use by the intrusion detection system. After that comes the *data update* stage. This stage spans the entire lifetime of the intrusion detection system because new vulnerability information will be added to the vulnerability database as it becomes known. This information can be either a previously unknown vulnerability, a new vulnerability introduced by reconfiguration, or a new exploitation of an existing vulnerability. A newly discovered vulnerability belonging to an existing vulnerability classification can use the hyperrules from that classification. So the hyperrules of this classification will be retrieved from the vulnerability database. In the other case, a new classification will be created by negating the relevant specifications of the exploit. The new classification and its corresponding specification will then be added to the vulnerability database. The specification will also be used in the creation or update of the specification for the current module. If the currently executing module does not have a specification yet, the programmer will be responsible for providing the specification that should at least include these hyperrules for this vulnerability. If there is already a specification for this module but the new hyperrules are not featured in the specification yet, then the programmer will be responsible for adding the new hyperrules to the specification.

Both the initial data population stage and the data update stage apply the same procedures to add the data to the database. While the initial data population stage adds all currently known information, the data update stage always adds previously unknown information, as this stage inserts the information that has just been discovered. Using the same Apache server vulnerability example in Chapter 4, the necessary steps to add data, together with their corresponding source codes written in Perl-embedded SQL, are described as follows:

1. Add identification information of the vulnerability if it is a new vulnerability.

```
my $sth = $dbh->prepare (
      '
      SELECT Vul_ID = ?;
      SELECT Desc = ?;

      SELECT DISTINCT Total_Vuls = COUNT (*)
      FROM  Vulnerability_Identification
      WHERE Vulnerability_ID = Vul_ID;

      CASE Total_Vuls < 1 THEN
            INSERT INTO Vulnerability_Identification
                  (Vulnerability_ID, Date_found,
                  Description)
            VALUES (Vul_ID, NOW (), Desc);
      END
      '
);
$sth->execute (“CVE-2002-0653”,
      “Buffer overflow caused by AllowOverride set to None
      in mod_ssl inside an Apache 1.3 server running Red
      Hat Linux 8.2”);
```

2. Derive the category and subcategory of the vulnerability based on the Aslam/Krsul scheme, and create the classification if it does not exist yet. Then add this initial classification data to the database.

```
my $sth = $dbh->prepare (
      '
      SELECT Vul_ID = ?;
      SELECT Cat_ID = ?;
      SELECT Subcat_ID = ?;
      SELECT Desc = ?;

      SELECT DISTINCT Total_classes = COUNT (*)
      FROM  Vulnerability_Identification
      WHERE Vulnerability_ID = Vul_ID AND
            Category_ID = Cat_ID AND
            Subcategory_ID = Subcat_ID;

      CASE Total_classes < 1 THEN
            INSERT INTO Vulnerability_Identification
                  (Category_ID, Subcategory_ID)
            VALUES (Cat_ID, Subcat_ID)
            WHERE Vulnerability_ID = Vul_ID;
      END

      SELECT DISTINCT Total_classes = COUNT (*)
      FROM  Vulnerability_Classification
      WHERE Vulnerability_ID = Vul_ID AND
            Category_ID = Cat_ID AND
            Subcategory_ID = Subcat_ID;

      CASE Total_classes < 1 THEN
            INSERT INTO Vulnerability_Classification
                  (Vulnerability_ID, Category_ID,
                  Subcategory_ID, Description)
            VALUES (Vul_ID, Cat_ID, Subcat_ID, Desc);
      ELSE
            INSERT INTO Vulnerability_Classification
                  (Description)
            VALUES (Desc)
            WHERE Vulnerability_ID = Vul_ID AND
                  Category_ID = Cat_ID AND
                  Subcategory_ID = Subcat_ID;
      END
      '
);
$sth->execute (“CVE-2002-0653”, “3”, “a5”,
      “Buffer overflow”);
```

3. Determine how the vulnerability is exploited. Obtain information about the system call, under each version of each module being executed when the exploitation occurs. The exploit information, which includes the name and version of the module and a system call sequence, will then be compared against the existing sets of exploit information in the Vulnerability_Exploit table of the database. As shown in the Vulnerability_Exploit table (Table 5) in Chapter 4,

each set of exploit information in the table includes the name and version of the module, the system call sequence involved in the exploitation, and is represented by a unique exploitation ID. If this set of exploitation information does not exist in the table, a new, unique ID will be assigned to represent the set, and the new exploitation ID will be added to the database. This enables the classification of this vulnerability to include the exploitation ID.

```
my $sth = $dbh->prepare (
      '
      SELECT Vul_ID = ?;
      SELECT Cat_ID = ?;
      SELECT Subcat_ID = ?;
      SELECT Mod_name = ?;
      SELECT Mod_version = ?;
      SELECT Sys_call = ?;
      SELECT Desc = ?;

      SELECT DISTINCT Exp_ID = Exploit_ID
      FROM  Vulnerability_Exploit
      WHERE Vulnerability_ID = Vul_ID AND
            System_Call = Sys_call;

      SELECT Total_IDs_returned = COUNT (Exp_ID);

      CASE Total_IDs_returned > 0 THEN
            INSERT INTO Vulnerability_Classification
                  (Exploit_ID)
            VALUES (Exp_ID)
            WHERE Vulnerability_ID = Vul_ID AND
                  Category_ID = Cat_ID AND
                  Subcategory_ID = Subcat_ID;
      ELSE
            SELECT DISTINCT Max_Exp_ID = MAX (Exploit_ID)
            FROM  Vulnerability_Exploit;

            INSERT INTO Vulnerability_Classification
                  (Exploit_ID)
            VALUES (Max_Exp_ID + 1)
            WHERE Vulnerability_ID = Vul_ID AND
                  Category_ID = Cat_ID AND
                  Subcategory_ID = Subcat_ID AND
                  LENGTH (Exploit_ID) = 0;

            INSERT INTO Vulnerability_Exploit
            VALUES
                  (Vul_ID, Max_Exp_ID + 1, Mod_name,
                  Mod_version, Sys_call, Desc);
      END
      '
);
$sth->execute (“CVE-2002-0653”, “3”, “a5”, “mod_ssl”,
      “2.4.9”, “ssl_compat_directive”,
      “AllowOverride not set to None”);
```

4. Identify the system information under which the vulnerability may be exploited. The system information should include the name, version and vendor name of the server that the exploited station belongs to, and the corresponding information of the operating system used by the server. If this set of system information has not been represented by an existing system ID yet, it means that this set of system information does not exist in the database. Therefore, a new, unique ID has to be generated to represent this set of system information. The system ID will then be included in the classification of this vulnerability. As shown in Table 3, each combination of server name, version, vendor, and OS name, version, vendor constitutes a unique system ID. Therefore, for example, the set of system information {"Apache", "1.3", "Apache Software Foundation", "Linux", "8.2", "Red Hat"} should have a different system ID from the set {"Apache", "1.3", "Apache Software Foundation", "Linux", "8.1", "Red Hat"} due to the difference in the versions of their respective Linux operating systems.

```
my $sth = $dbh->prepare (
      '
      SELECT Vul_ID = ?;
      SELECT Cat_ID = ?;
      SELECT Subcat_ID = ?;
      SELECT Serv_name = ?;
      SELECT Serv_version = ?;
      SELECT Serv_vendor = ?;
      SELECT Oper_name = ?;
      SELECT Oper_version = ?;
      SELECT Oper_vendor = ?;

      SELECT DISTINCT Sys_ID = System_ID
      FROM  System
      WHERE Server_Name = Serv_name AND
            Server_Version = Serv_version AND
            Server_Vendor = Serv_vendor AND
            OS_Name = Oper_name AND
            OS_Version = Oper_version AND
            OS_Vendor = Oper_vendor;

      SELECT Total_System_IDs = COUNT (Sys_ID);

      CASE Total_System_IDs > 0 THEN
```

```
                        INSERT INTO Vulnerability_Classification
                              (System_ID)
                        VALUES (Sys_ID)
                        WHERE Vulnerability_ID = Vul_ID AND
                              Category_ID = Cat_ID AND
                              Subcategory_ID = Subcat_ID;
                  ELSE
                        SELECT DISTINCT Max_Sys_ID = MAX (System_ID)
                        FROM  System;

                        INSERT INTO Vulnerability_Classification
                              (System_ID)
                        VALUES (Max_Sys_ID + 1)
                        WHERE Vulnerability_ID = Vul_ID AND
                              Category_ID = Cat_ID AND
                              Subcategory_ID = Subcat_ID;

                        INSERT INTO System
                        VALUES
                              (Max_Sys_ID + 1, Serv_name,
                              Serv_version, Serv_vendor, Oper_name,
                              Oper_version, Oper_vendor);
                  END
                  ’
            );
            $sth->execute (“CVE-2002-0653”, “3”, “a5”, “Apache”, “1.3”,
                  “Apache Software Foundation”, “Linux”, “8.2”,
                  “Red Hat”);
```

5. Determine the environment(s) in which the vulnerability, denoted as v, is exploited, and, if so, what settings and values are involved. The environmental information includes some variable name(s) representing the environment(s) and their corresponding value(s) when v is exploited. For example, the information may include the input parameter (environment variable) of the UNIX system call *getenv* and its corresponding output (environment value for that variable), or some Windows registry keys and their corresponding values. A list of existing environment variables can be made by executing a shell command or a batch program. In UNIX case, shell command *printenv* lists all current environment variables and their respective values. Each set of environmental information {E_n, e_n} is recognized by v’s vulnerability ID in order to indicate that e_n is the setting of E_n during v’s exploitation. On the other hand, v can have multiple sets of

environmental information. For instance, {E_1, e_1} and {E_2, e_2} can be two different sets of environmental information recognized by v's vulnerability ID because when v is exploited, e_1 must be an explicit value of environment variable E_1 and e_2 must be an explicit value of environment variable E_2. For each {E_n, e_n} that has not yet been recognized by v, it will be added to the entry for v to indicate that this setting plays a part in the exploitation.

```
my $sth = $dbh->prepare (
      '
      SELECT Vul_ID = ?;
      SELECT Env_name = ?;
      SELECT Env_val = ?;
      SELECT Env_desc = ?;

      SELECT DISTINCT Total_envs = COUNT (*)
      FROM  Environment
      WHERE Vulnerability_ID = Vul_ID AND
            Name = Env_name AND
            Value = Env_val;

      CASE Total_envs < 1 THEN
            INSERT INTO Environment
            VALUES (Vul_ID, Env_name, Env_val, Env_desc);
      END
      '
);
$sth->execute (“CVE-2002-0653”, “user_name”, “*”,
      “Unauthorized user”);
```

6. Look for possible impacts caused by the vulnerability to see if any residual vulnerability will also be exploited. If, for example, Krsul's taxonomy is being used in a vulnerability search, the data found in the vulnerability search will contain information not only about the vulnerability itself, but also the possible impacts if any attack successfully exploits the vulnerability, as Krsul's taxonomy includes information about how to derive the impacts of a vulnerability using decision trees [24]. For other taxonomies that do not consider impacts caused by a vulnerability, the Requires/Provides model proposed by Templeton and Levitt can

be a viable option to determine the possible impacts of a vulnerability [47]. This model helps relate a vulnerability to its residual vulnerabilities if it is exploited by an attack. It describes an attack as a model, in terms of *capabilities* and *concepts*, based on the required components, their capabilities of leading to components needed for other attack(s), and the method in composing these secondary components into another form of attack. By its definition, an attack is a composition of abstract attack *concepts*, each of which requires certain *capabilities* to occur for a particular instance of the concept to be entailed or to introduce another concept.

For example, consider the “open window” vulnerability in a packet filtering firewall mentioned in section 2.6. A vulnerability search using Krsul’s taxonomy should be able to include buffer overflow as one of the possible impacts (CVE number CAN-2003-0132 as seen in section 4.3) because it allows an unauthorized program to be attached to the end of a valid data packet, allowing the program to be executed in the router after the packet gets past the open window. As a result, the router can be compromised if the unauthorized program execution changes access permission filters in favor of the attacker, and this enables the attacker to launch a denial of service attack. Therefore, a buffer overflow vulnerability is one of the possible impacts for the “open window” vulnerability, and for the buffer overflow vulnerability, unauthorized program execution becomes a possible impact.

Likewise, based on the Requires/Provides model, the concept of “open window” requires the router window to be open for valid data packet and provides the data packets a limited time of authorized actions. On the other hand, the concept of buffer overflow requires the data packets to look valid and provides an attacker an opportunity by running the authorized execution(s) as an unauthorized user from the extra data packet(s) appended beyond the valid size of the valid data buffer during the authorized state inside the “open window”. Consequently, the relation between “open window” vulnerability and unauthorized executions becomes apparent because the “open window” vulnerability introduces an impact – the buffer overflow vulnerability, which can lead to another impact – unauthorized program executions.

Similar to the system information and the exploitation information, if there is no existing ID for the impact information, a new, unique impact ID will be generated and added to the database together with some relevant description provided by the system administrator or programmer.

```
my $sth = $dbh->prepare (
      ‘
      SELECT Vul_ID = ?;
      SELECT Imp_ID = ?;
      SELECT Desc = ?;

      SELECT DISTINCT Total_impacts = COUNT (*)
      FROM  Vulnerability_Impact
      WHERE Vulnerability_ID = Vul_ID AND
            Impact_ID = Imp_ID;

      CASE Total_impacts < 1 THEN
            INSERT INTO Vulnerability_Impact
            VALUES (Vul_ID, Imp_ID, Desc);
      END
      ’
);
$sth->execute (“CVE-2002-0653”, “CAN-2003-0132”,
      “DOS on Apache server as a result of open window”);
```

7. The programmer or system administrator for the IDS should generate or update the specification based on the vulnerability information in the database. The specification has to include generic hyperrules for its entire vulnerability classification, and must be added to the database as soon as it is ready, so that the intrusion detection system can detect attempts to exploit this vulnerability.

```
my $sth = $dbh->prepare (
      '
      SELECT Vul_ID = ?;
      SELECT Cat_ID = ?;
      SELECT Subcat_ID = ?;
      SELECT Exp_ID = ?;
      SELECT Sys_ID = ?;
      SELECT Spec = ?;

      INSERT INTO Vulnerability_Classification
            (Specification)
      VALUES (Spec)
      WHERE Vulnerability_ID = Vul_ID AND
            Category_ID = Cat_ID AND
            Subcategory_ID = Subcat_ID AND
            Exploit_ID = Exp_ID AND
            System_ID = Sys_ID;

      INSERT INTO Vulnerability_Identification
            (Date_addressed)
      VALUES (NOW ())
      WHERE Vulnerability_ID = Vul_ID AND
            Category_ID = Cat_ID AND
            Subcategory_ID = Subcat_ID;
      '
);
$sth->execute (“CVE-2002-0653”, “3”, “a5”, “5”, “4”,
      “Environment Variables
       ENV int E = 0;
       LOCAL ENV int L = 0;

       Start Expression
       SE: <progGeneric>

       Hyperrules
       <progGeneric> -> <writeGeneric, E>.
       <writeGeneric, 0> ->
            <openGeneric> <closeGeneric> { E = E – 1; }.
       <openGeneric> -> open_Generic { E = E + 1; L = 1; }.
       <closeGeneric> -> close_Generic.”);
```

5.2 Integration with Distributed Intrusion Detection Systems

An intrusion detection system can be either centralized or distributed. Most of the traditional IDSes are centralized, so data is processed and analyzed inside a single host housing the intrusion detection director. In recent years, with the increased use of distributed operating systems and the emergence of the World Wide Web, distributed intrusion detection systems are becoming available. A distributed intrusion detection system is defined as a system where data monitoring can be performed at a number of locations, while data analysis can be conducted at one single location or various locations throughout the network [4, 34, 35, 40, 45, 46].

A well-known framework, called *Common Intrusion Detection Framework* (*CIDF*), uses an event-and-response approach to define a list of components that can make up each agent or sensor of an intrusion detection system. These components are the *event generators* (*E-boxes*), *event analyzers* (*A-boxes*), *event databases* (*D-boxes*), and *response units* (*R-boxes*), respectively [35].

Distributed intrusion detection systems use *intelligent agents* or *external sensors*. The approach emphasizes multiple-location data analysis in addition to multiple-location data collection. They communicate actively with each other, or with a distributed analysis unit, using communication events and alerts. The data collection components and data analysis components are analogous to the E-boxes and A-boxes in the CIDF definition, respectively. Each agent is designed to report events of a particular kind of interests, and the agents dynamically monitor in response to event notifications or alerts. Analyses are mostly performed hierarchically. They observe the behaviors of the data packets by one of two ways before allowing the data packets to invoke any library functions. One way is to capture the packets and compare with the system states, and the other way is to intercept the packets and analyze them [44, 45].

The intelligent agents approach is found in many popular frameworks of distributed intrusion detection systems like DIDS, GrIDS, EMERALD, and AAFID. DIDS [40] uses only one level of hierarchy, the centralized director, to analyze data, despite distributed data monitoring. GrIDS [46] uses activity graphs to build a hierarchy of departments and hosts based on an organization model. The graphs represent hosts and network activities. EMERALD [34] employs monitors at the levels of hosts, domains, and enterprises to form an analysis hierarchy, and uses a subscription-based communication scheme both within and between monitors. The subscription scheme between monitors is hierarchical. AAFID [4, 45] employs autonomous agents at the lowest level for data collection and analysis, and monitors at higher hierarchical levels to control the agents and overlook the activities in the global sense of view.

A vulnerability database is a data source that all agents and monitors can share. The information in the database indicates the version and OS involved in the vulnerability and specification, so that all agents and monitors can determine if certain information is relevant or not for their own purposes. As the database is OS-independent, data sharing among the agents is possible regardless of the system that each agent or monitor runs on. Since the agents are the only components conducting data analysis, any new updates to the vulnerability database will require notifying the agents so that every other intelligent agent can update itself and send information back to the vulnerability database. Alternatively, the agents can regularly perform polling to look for and pick up updated information. However, this would require all agents to continuously retrieve status information from the database, and since most of the time the database remains unchanged, the majority of pollings become pointless, making resource overhead a concern in this approach.

The importance of communications among the agents requires that any integration of a vulnerability database with an intrusion detection system must not affect inter-agent communications efficiency. Many relational database systems use B-tree data structure to store information. With cardinality n, i.e., total keys in O (2^n), most dynamic-set operations in a B-tree have performance of O (log 2^n) = O (n). Hence, a typical database retrieve operation SELECT depends on the cardinality and is O (n), which is very efficient. The overall cost for a query is the sum of query cost and communication cost. If M is the cost of sending the request or response, the overall cost is 2M + O (n) or 2M + nD, where D is the number of database accesses and there are O (2^n) keys. In a wide area network environment, as M >> D, the communication cost is the dominant factor, which means that the query cost will not affect the overall performance very much. On the other hand, in a local area network environment, the values of M and D are much closer to each other. As a result, the query cost will play as an important role as the communication cost. In a usual LAN environment, the size of the database is also relatively smaller because less data is shared among fewer users in the network than in the WAN environment. The smaller database in LAN environment implies that the cardinality in the SELECT operation will be in limited size, meaning that the query cost will also be limited. Therefore, a database with vulnerability taxonomy as its schema can be integrated with the intrusion detection system so as to increase security functionalities without sacrificing efficiency.

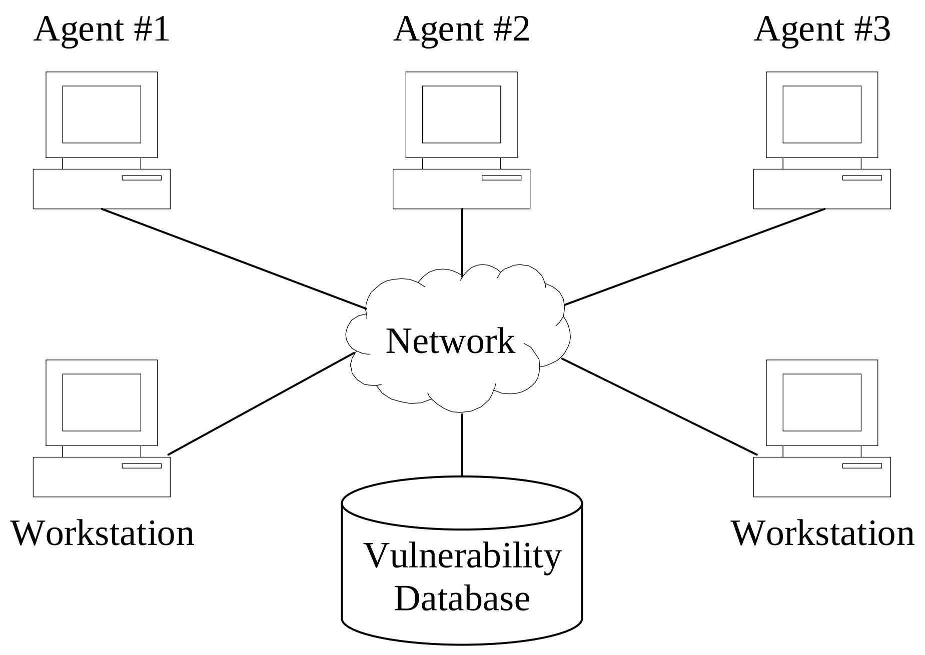

Figure 3. A Distributed IDS model integrated with vulnerability database

Figure 3 shows a specification-based distributed intrusion detection model which has three agents performing intrusion detection and is integrated with a vulnerability database. Since a distributed IDS can perform intrusion detection in more than one location, the same specifications have to be present in all the locations that participate in the detections. Therefore, each of these locations has to maintain the same specification for each involved vulnerability. This means that whenever a specification is updated in one location, the updates have to be reflected in the vulnerability database. This allows the database to send automated alerts to the agents so that the agents can pick up the updates from the database and apply the updates onto the same specifications for their respective locations. For example, if a programmer updates some specifications at agent

#1, he also has to update the same specifications in the database, so that the database can alert both agent #2 and agent #3 regarding the updates. For agent #2 and agent #3, if these new updates are needed in their own detection participations, they can immediately pick up the updates and make necessary modifications on the same specifications on their respective agents.

5.3 Case study: Building Block Approach

Crosbie and Kuperman, citing the shortcomings in the attack-signature matching approach in most conventional intrusion detection systems, proposed an IDS model which monitors the system in real-time fashion by looking for the *building blocks* of misuse actions. These building blocks of misuse actions are listed in section 1.2. In this approach, a detection tool, based on a detection template listing all the building blocks of misuse actions, resides in the kernel together with the detection itself as well as a specialized kernel data source. The detection tool in the kernel references this template to look for possible attack patterns [13]. In order to avoid much sacrifice in kernel efficiency, both the template and the data source have to be minimal and effective. As a result, any future growth in either the template or the data source will have to be very limited. Scalability, therefore, becomes a concern.

Since the detection template in the building block model has its own classification of misuse actions, we can view it as a schema based on a primitive taxonomy. The only difference between this building block model and the general model described in Chapter 3 is that the schema in the building block model needs to explicitly reside inside the kernel. As a schema in general merely describes the organization and classification of the data inside a database, the size of a kernel-based schema is not big enough to deteriorate the kernel efficiency. Hence, a vulnerability schema can be used to replace the detection template in the building block model as it is much more detailed and well-structured. On the other hand, the building block data source inside the kernel will be replaced with the vulnerability database outside the kernel, so that the data source can be scalable. The delay caused by kernel latency as a result of each query is minimal because experiments have shown that, on a Linux system with the 2.4.17 kernel, each disk access averages 79 ms [53], so database queries on that system have an acceptable cost. Thus, the kernel will not have much additional burden to hinder its performance during data query even if the database is located outside the kernel. This suggests that with the vulnerability database residing outside the kernel, the kernel efficiency will not be affected much, as long as all other kernel tasks performed inside the building block model will still be performed in the same manner as they are originally defined. As a result, kernel efficiency becomes a non-issue. In addition, this modified building block model can still enjoy the same advantages of vulnerability database integration with a general intrusion detection system described in previous chapters.

Chapter 6 Conclusions and Future Works

6.1 Summary

An intrusion detection system integrated with a vulnerability database has the following advantages:

1. Specification updates upon a discovery of new vulnerability
2. Unambiguous specifications for all vulnerabilities under the same classification so that there will be no inconsistency in handling any attack that exploits the same vulnerability
3. Computation overhead reduction by saving the efforts of complicated and possibly erratic manual programming statements upon any human update on specifications
4. No additional scalability concern as long as the whole network system is scalable because the vulnerability database itself is already scalable

One purpose of vulnerability analysis is to look for unknown vulnerabilities based on the knowledge of known vulnerabilities combined with different possible methods of exploitations before the unknown vulnerabilities become exploitable to actual attacks. On the other hand, specification-based intrusion detections apply the knowledge of known vulnerabilities onto dynamic detections of actual attacks without requiring attack signatures. Therefore, with the vulnerability database as the bridge between vulnerability analysis and intrusion detection, an intrusion prevention model will be able to both look for new vulnerabilities and detect intrusions at the same time.

6.2 Applying Penetration Analysis

There are two aspects for the security characteristics of a system, the *expected security functionality* and the *possible implied security behaviors* affecting the environment. *Expected security functionality* reflects how secure the module is when it executes, while *implied security behavior* demonstrates the security state when the module runs simultaneously with other modules. A module being secure when it runs by itself by no means guarantees that it is also secure when it runs simultaneously with other modules. Race conditions demonstrate this.

Formal verification begins with the *preconditions*, which hold states before a system begins, and analyzes the *postconditions*, the states resulting from some execution(s) of the system [7]. To verify security requirements, functional testing is mostly a straightforward solution. To verify security implications and security interactions of separately designed packages is more difficult due to more complications [29], when each module needs to check the states against each of the other components running at the same time in order to verify that the required security is still maintained. These verifications are usually handled by *penetration analysis*.

Penetration analysis, a technique that has security analysts try to violate the security policy, tests not only procedural and operational, but also technological controls [7]. A common model for performing penetration analysis is the *Flaw Hypothesis Methodology* (*FHM*) [27, 49, 55, 56, 57]. Also, Gupta and Gligor presented two hypotheses regarding penetration testing. The *Hypothesis of Penetration Patterns* asserts system flaws are caused by penetration patterns arising from errors in system condition checks or integrated flow conditions. The *Hypothesis of Penetration-Resistant Systems* asserts a system is resistant to penetration as long as it adheres to a specific design property set [18].

To construct a foundation for a penetration-resistant computer system, penetration analysis selects different vulnerability detection methodologies and taxonomies based on a set of formalized design properties that characterize resistance to penetration. It uses both flow-based models and a state-analysis approach for possible flaws. It helps define a system state as a set of integrated flow paths traversed by the system up to a certain point during the execution. It is a simulation by an appointed *tiger team* or *red team* with goals to exploit known and new vulnerabilities by attempting to break the security such as gaining unauthorized access, causing denial-of-service, and bypassing system accountability, etc. The penetration pretends that system calls are being made at certain points of the penetration analysis.

Before a run of penetration analysis, each system entity has a set of penetration-resistance policies associated with some permissions to be altered, viewed, and invoked within an atomic sequence during a penetration test. While these rules define access control and accountability in a high-level sense, penetration analysis can be viewed as the low-level analysis of how these rules are complied and cooperated during the penetration test. Besides, both rules and penetration analysis must be consistent, preserving and trusted, meaning that they have to be carefully reviewed before the test, and during the test they must always be enforced and verified during any possible state change.

During the penetration test, *Primitive Flow Generator* (*PFG*) converts the source codes into execution flow paths conducted in an *atomic sequence*. These execution flow paths can consist of *information flows, function calls, conditions,* or a combination of them [18]. Any improper state at some certain location along the series of integrated execution flow paths for a particular system call results in a flaw. This flaw can lead to illegal or unintended gain of access to the system and its resource if an unprivileged user takes advantage of it.

Penetration analysis can also work with *fault injection*. Fault injection is a simulation that looks for, and predicts, a modified function, say *f'*, which is different from the targeted original normal function, *f*, of the program. Hence, *f'* can result in different program states at least for some inputs. Using such algorithms as Adaptive Vulnerability Analysis (AVA) [16] for fault injection, primitive flow generator converts both *f* and *f'* into two different sets of execution flows so that state difference will be identified at any point that normal state is diverted into another undesired state.

Given a defined standard environment, not only can penetration analysis provide a list of found vulnerabilities and how they can be exploited, but also can enable the testers to rate the potential damage from an exploitation based on such usual factors as ease of exploitation, likelihood that the flaw exists, and possible effects of the exploitation of the flaws [15]. All of these are helpful for security research, designs and improvements. In addition, the rankings enable an analyst to compare vulnerabilities among different products with similar security functionalities. Another importance of the vulnerability information obtained through penetration analysis is that it helps a specification-based intrusion detection system to learn how to detect these security violations in the future.

The papers by Gupta and Gligor also presented the automation of penetration analysis in the theoretical points of view, but so far there has not been much work on putting these ideas into practical or experimental achievements. As shown in the automated penetration analysis paper by Gupta and Gligor [18], certain aspects of penetration analysis can be automated. However, a completely automated penetration analysis will enable automatic updates in the vulnerability database, allowing the entire vulnerability analysis to be automated.

6.3 Future Works

The proposed model is ideal for dynamic detections of both known and unknown attacks. The integration of vulnerability database is even more desirable with the never-ending discovery of new vulnerabilities as a result of growing complexities of new systems and new software applications. Yet there are still some future works required. One major area that requires a lot of effort is the complete automation of penetration analysis. It will be an important addition if the whole vulnerability analysis can be automated inside this model so that the vulnerability data can also be updated automatically. This feature adds another significance in the design, implementation, and verification for an intrusion detection system. Applying automated penetration analysis with a vulnerability database allows *adaptations*, as the vulnerability data inside the database is updated automatically after each penetration test.

Since the vulnerability database used in this proposed model also includes information on not only the vulnerabilities but also fixes, regular users of the database will include the system administrators. As a result, network maintenance becomes one of the purposes of the vulnerability database, and system administrators need to provide additional information to the database regularly for this purpose. Therefore, several entities will reference the vulnerability database inside the suggested model. It would also be nice if the specifications can be updated automatically upon changes in the vulnerability database. To accomplish this, a nice correlation module is required among the newly discovered vulnerabilities, the updated vulnerability data, and the corresponding specifications. The correlation module needs to easily and efficiently *alert* the kick-off of any specifications update using specification language whenever there is any addition or modification of vulnerability data.

Future research on how to best correlate a discovered vulnerability to a category without any interface can lead to full automation of penetration analysis and vulnerability data update, while future research on how to best correlate vulnerability data changes to corresponding specification modifications without human intervention can lead to full automation of specification updates. The combined automation will, therefore, lead to full automation of this integrated model.

References

[1] Robert P. Abbott, J. S. Chin, J. E. Donnelley, W. L. Konigsford, S. Tobuko, D. A. Webb; **Security Analysis and Enhancements of Computer Operating Systems**; *NBSIR 76-1041*; Institute for Computer Sciences and Technology, National Bureau of Standards; April; 1976

[2] Taimur Aslam; **A Taxonomy of Security Faults in the UNIX Operating System**; *Master Thesis*; Purdue University; August; COAST TR 95-09; 1995

[3] Taimur Aslam, Ivan Krsul, Eugene H. Spafford; **Use of A Taxonomy of Security Faults**; *Proceedings of 19th NIST-NCSC National Information Systems Security Conference*; September; COAST TR 96-05; 1996

[4] Jai Sundar Balasubramaniyan, Jose Omar Garcia-Fernandez, David Isacoff, Eugene H. Spafford, and Diego Zamboni; **An Architecture for Intrusion Detection using Autonomous Agents**; *Proceedings of 14th Annual Computer Security Applications Conference*; IEEE Computer Society; December, 13-24; 1998

[5] Boris Bezier; **Software Testing Techniques**; *Electrical Engineering/Computer Science and Engineering Series*; Van Nostrand Reinhold; 1983

[6] Richard Bisbey II, Dennis Hollingsworth; **Protection Analysis Project Final Report**; *Technical Report ISI/RR-78-13, DTIC AD A056816*; University of Southern California Information Science Institute; May; 1978

[7] Matt Bishop; **Computer Security: Art and Science**; Addison-Wesley; 2002

[8] Matt Bishop; **Vulnerability Analysis**; *Proceedings of the Second International Symposium on Recent Advances in Intrusion Detection*; September, 125-136; 1999

[9] Matt Bishop, Michael Dilger; **Checking for Race Conditions in File Accesses**; *Computing Systems*; Volume 9, Number 2; Spring, 131-152; 1996

[10] Mark Bruno; **A New Breed Of Criminals**; *Bank Technology News*; January; 2003

[11] Center for Education and Research in Information Assurance and Security (CERIAS); **CERIAS Vulnerability Database**; Purdue University

[12] Computer Emergency Response Team (CERT); **CERT/CC Vulnerability Notes**

[13] Mark Crosbie and Benjamin Kuperman; **A Building Block Approach to Intrusion Detection**; *Proceedings of the Recent Advances in Intrusion Detection*; October; 2001

[14] Dan DaCosta, Christopher Dahn, Spiros Mancoridis, Vassilis Prevelakis; **Characterizing the "Vulnerability Likelihood" of Software Components**; *Department of Mathematics & Computer Science*; Drexel University; 2003

[15] Deborah D. Downs, Ranwa Haddad; **Penetration Testing – The Gold Standard for Security Rating and Ranking**; *Proceedings of 1st Workshop on Information-Security-System Rating and Ranking*; May; 2001

[16] Anup K. Ghosh, Tom O’Connor, Gary McGraw; **An Automated Approach for Identifying Potential Vulnerabilities in Software**; *Proceedings of the IEEE Symposium on Security and Privacy*; May, 104-114; 1998

[17] Rajeev Gopalakrishna; **A Framework for Distributed Intrusion Detection using Interest-Driven Cooperative Agents**; *CERIAS Tech Report*; 2001-44; 2001

[18] Sarbari Gupta , Virgil D. Gligor; **Automated Penetration Analysis System and Method, Part 1 and Part 2**; *U.S. Patent # 5, 485, 409*; 1992

[19] ICAT Metabase; **A CVE Based Vulnerability Database**; National Institute of Standards and Technology

[20] Donald E. Knuth; **The Errors of TEX**; *Software Practice and Experience*; Volume 19 Number 7; 607-685; 1989

[21] Calvin Ko, George Fink, and Karl Levitt; **Automated Detection of Vulnerabilities in Privileged Programs by Execution Monitoring**; *Proceedings of 10th Annual Computer Security Applications Conference*; Orlando, FL; Dec, 134-144; 1994

[22] Calvin Ko; **Execution Monitoring of Security-Critical Programs in A Distributed System: A Specification-based Approach**; *PhD Dissertation*; University of California, Davis; August; 1996

[23] Calvin Ko, Manfred Ruschitzka, and Karl Levitt; **Execution Monitoring of Security-Critical Programs in Distributed Systems: A Specification-based Approach**; *Proceedings of the 1997 IEEE Symposium on Security and Privacy*; Oakland, CA; May, 175-187; 1997

[24] Ivan V. Krsul; **Software Vulnerability Analysis**; *PhD Thesis*; Purdue University; May; COAST TR 98-09; 1998

[25] Carl E. Landwehr, Alan R. Bull, John P. McDermott, William S. Choi; **A Taxonomy of Computer Program Security Flaws, with examples**; *ACM Computer Surveys*; Volume 26 Number 3; September, 211-254; 1994

[26] Yihua Liao, Rao Vemuri; **Using Text Categorization Techniques for Intrusion Detection**; *11th USENIX Security Symposium*; August; 2002

[27] Richard R. Linde; **Operating System Penetration**; *Proceedings of National Computer Conference*; Volume 44; May, 361-368; 1975

[28] Stefan Lindskog, Erland Jonsson; **Different Aspects of Security Problems in Network Operating Systems**; *Proceedings of the Third Annual International Systems Security Engineering Association Conference (2002 ISSEA Conference)*; March; 2002

[29] Lingfeng Ma, Salvador Mandujano, Guangfeng Song, Pascal Meunier; **Sharing Vulnerability Information using a Taxonomically-correct, Web-based Cooperative Database**; Center for Education and Research in Information Assurance and Security, Purdue University; May; Technical Report 2001-03; 2001

[30] MITRE; **Common Vulnerabilities and Exposures**

[31] Robert Morris, Ken Thompson; **Password Security: A Case History**; *Communications of the ACM*; Volume 22 Number 11; November, 594-597; 1979

[32] Peter G. Neumann; **Computer System Security Evaluation**; *1978 National Computer Conference Proceedings (AFIPS Conference Proceedings 47)*; June, 1087-1095; 1978

[33] Open Source Vulnerability Database; **OSVDB**

[34] Philip A. Porras and Peter G. Neumann; **EMERALD: Event Monitoring Enabling Responses to Anomalous Live Disturbances**; *1997 National Information Systems Security Conference*; October; 1997

[35] Philip A. Porras, Dan Schnackenberg, Stuart Staniford-Chen, Maureen Stillman, and Felix Wu; **The Common Intrusion Detection Framework architecture**; *Interoperability Experiment*; September; 1999

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

[37] Katherine E. Price; **Host-based Misuse Detection and Conventional Operating Systems' Audit Data Collection**; *Master Thesis*; Purdue University; December; COAST TR 97-15; 1997

[38] Raymond J. Rubey, Joseph A. Dana, Peter W. Biché; **Quantitative Aspects of Software Validation**; *Proceedings of the International Conference on Reliable Software*; April, 246-251; 1975

[39] SecurityFocus; **SecurityFocus Vulnerabilities Archive**

[40] Steven R. Snapp, James Brentano, Gihan V. Dias; **DIDS (Distributed Intrusion Detection System) -- Motivation, Architecture, and An Early Prototype**; *Proceedings of the 14th National Computer Security Conference*; October; 1975

[41] Sony Pictures; **The Net**; 1995

[42] Eugene H. Spafford; **Computer Viruses as Artificial Life**; *Journal of Artificial Life*; Volume 1 Number 3; 249-265; COAST TR 94-02; 1994

[43] Eugene H. Spafford; **The Internet Worm Program: An Analysis**; *ACM Computer Communications Review*; Volume 19 Number 1; January, 17-57; 1989

[44] Eugene H. Spafford and Diego Zamboni; **Data Collection Mechanisms for Intrusion Detection Systems**; *CERIAS Technical Report*; 2000-08; June; 2000

[45] Eugene H. Spafford and Diego Zamboni; **Intrusion Detection using Autonomous Agents**; *Computer Networks*; Volume 34 Number 4; October, 547-570; 2001

[46] Stuart Staniford-Chen, Steven Cheung, Rick Crawford, Michael Dilger, Jeremy Frank, James Hoagland, Karl Levitt, Christopher Wee, Raymond Yip and Dan Zerkle; **GrIDS -- A Graph-Based Intrusion Detection System for Large Networks**; *Proceedings of the 19th National Information Systems Security Conference*; September; 1996

[47] Steven J. Templeton, Karl Levitt; **A Requires/Provides Model for Computer Attacks**; *Proceedings of the New Security Paradigms Workshop 2000*; September; 2000

[48] 20th Century Fox; **Die Hard 2**; 1990

[49] United States Department of Defense Computer Security Evaluation Center; **Trusted Computer System Evaluation Criteria**; *DoD 5200.28-STD*; 1985

[50] John Viega, J. T. Bloch, Tadayoshi Kohno, Gary McGraw; **ITS4: A Static Vulnerability Scanner for C and C++ Code**; *Proceedings of 16th Annual Computer Security Applications Conference*; December; 257-269; 2000

[51] David Wagner, Jeffrey S. Foster, Eric A. Brewer, Alexander Aiken; **A First Step towards Automated Detection of Buffer Overrun Vulnerabilities**; *Proceedings of the Year 2000 Network and Distributed System Security Symposium (NDSS)*; 3-17; 2000

[52] Christina Warrender, Stephanie Forrest, Barak Pearlmutter; **Detecting Intrusions Using System Calls: Alternative Data Models**; *Proceedings of 1999 IEEE Symposium on Security and Privacy*; IEEE Computer Society; 133-145; 1999

[53] Andrew Webber; **Realfeel Test of the Preemptible Kernel Patch**; Linux Journal; October; 2002

[54] David M. Weiss and Victor R. Basili; **Evaluating Software Development by Analysis of Changes: Some Data from the Software Engineering Laboratory**; *IEEE Transactions on Software Engineering*; Volume 11 Number 2; February, 157-168; 1985

[55] Clark Weissman; **Penetration Testing**; *Information Security: An Integrated Collection of Essays*; Volume 6 Essay 11; IEEE Computer Society Press; 269-296; 1995

[56] Clark Weissman; **Security Penetration Testing Guideline**; *Handbook for the Computer Security Certification of Trusted Systems*; Chapter 10; Technical Memorandum 5540:082A; Naval Research Laboratory; January; 1995

[57] Clark Weissman; **System Security Analysis/Certification Methodology and Results**; SP-3728; System Development Corporation; October; 1973

Appendix A Source Listings

A.1 Source listing for generic example in section 4.4

```
#!/bin/perl –w

use Mysql;

$new_file = 0;

# Open the specification file if it exists,
# otherwise create one.
unless ( open ( PEG, "+<./peg" ) )
{
      open ( PEG, ">./peg" );
      $new_file = 1;
}

# Generic specifications.
my $generic_spec = join ( "",
      "    <progGeneric> -> <writeGeneric, E>.\n",
      "    <writeGeneric, 0> -> ",
      "<openGeneric> <closeGeneric> { E = E – 1; }.\n",
      "    <openGeneric> -> open_Generic ",
      "{ E = E + 1; L = 1; }.\n",
      "    <closeGeneric> -> close_Generic.\n" );
my $new_spec = "Hyperrules\n";

# Environment variable declarations and module lists.
my $vars_decl = join ( "", "    Environment Variables\n",
      "    ENV int E = 0;\n",
      "    LOCAL ENV int L = 0;\n " );
my $module_list = "    Start Expression\n    SE: ";

# Assume using the database "test" inside this server.
my $dbh = Mysql->connect;
$dbh = selectdb ( "test" );

# Query all system calls that have this vulnerability.
my $sth = $dbh->prepare (
      '
      SELECT DISTINCT ve.System_Call
      FROM  Vulnerability_Exploit ve,
            Vulnerability_Classification vc,
            Vulnerability_Identification vi
      WHERE ve.Vulnerability_ID = ? AND
            ve.Module_Name = ? AND
            ve.Exploit_ID = vc.Exploit_ID AND
            ve.Vulnerability_ID = vi.Vulnerability_ID AND
            vc.Category_ID = vi.Category_ID AND
            vc.Subcategory_ID = vi.Subcategory_ID;
      '
);
$sth->execute ( "CVE-1999-0812", "submnt" );

# Handle each system call in the specification language.
```

```
my $sys_call = "";
my $join_string = "";
while ( $sys_call = $sth->fetchrow () )
{
      $i = 0;

      # Create the specifications string for this module.
      my @s = split ( "Generic", $generic_spec );
      my $current_spec = join ( $sys_call, @s );

      # Check if @sys_call has already been handled.
      CHECKLINE: while ( !$new_file )
      {
            last CHECKLINE if eof PEG;
            $line = <PEG>;
            $i++;
            last CHECKLINE
                  if index ( $line, $current_spec, 0 )
                        >= 0;
      }

      # If @module_name does not have these specifications,
      # then add these specifications.
      if ( $i <= 0 )
      {
            # Append another module name into the
            # start expression string.
            if ( length ( $module_list ) < 20 )
                  $join_string = "";
            else
                  $join_string = " || ";
            my $current_module = join ( "", "<",
                  $sys_call, ">" );
            $module_list = join ( $join_string,
                  $module_list, $current_module );

            # Append another generic hyperrules.
            $new_spec = join ( "\n", $new_spec,
                  $current_spec );
      }
}

# Write everything as specifications.
print PEG $vars_decl;
print PEG "\n";
print PEG $module_list;
print PEG "\n";
print PEG $new_spec;

$sth->finish ();
$sth2->finish ();

close PEG;
```

A.2 Source listing for practical example in section 4.5

```
#!/bin/perl –w

use Mysql;

$new_file = 0;

# Open the specification file if it exists,
# otherwise create one.
unless ( open ( PEG, "+<./peg" ) )
{
      open ( PEG, ">./peg" );
      $new_file = 1;
}

# Assume using the database "test" inside this server.
my $dbh = Mysql->connect;
$dbh = selectdb ( "test" );

# Query the exploit ID that have this vulnerability.
my $sth = $dbh->prepare (
      '
      SELECT DISTINCT Exploit_ID
      FROM  Vulnerability_Exploit
      WHERE System_Call = ?;
      '
);
$sth->execute ( "open_rwtc" );
my ( $exploit_ID ) = $sth->fetchrow ();

# Query all the modules and the specification codes
# for this exploit ID.
my $sth2 = $dbh->prepare (
      '
      SELECT DISTINCT ve.Module_Name, vc.Specification
      FROM  Vulnerability_Exploit ve,
            Vulnerability_Identification vi,
            Vulnerability_Classification vc
      WHERE ve.Vulnerability_ID = vi.Vulnerability_ID AND
            ve.Exploit_ID = ? AND
            ve.Exploit_ID = vc.Exploit_ID AND
            vi.Category_ID = vc.Category_ID AND
            vi.Subcategory_ID = vc.Subcategory_ID;
      '
);
$sth2->execute ( $exploit_ID );
my @data = ();

# Handle each module in the specification language.
my $module_name = "";
my $spec = "";
while ( @data = $sth2->fetchrow () )
{
      $module_name = $data[0];
      $spec = $data[1];
```

```
            $i = 0;

            # Replace the generic specification codes
            # with a customized one by replacing every
            # existence of the string "Generic" with
            # the name of the module.
            my @s = split ( "Generic", $spec );
            $new_spec = join ( $module_name, @s );

            CHECKLINE: while ( !$new_file )
            {
                  last CHECKLINE if eof PEG;
                  $line = <PEG>;
                  $i++;
                  last CHECKLINE
                        if index ( $line, $new_spec, 0 ) >= 0;
            }

            # If $module_name does not have these specifications,
            # then add these specifications.
            if ( $i <= 0 )
            {
                  # Write everything into the specification file
                  # after customizations.
                  print PEG $new_spec;
            }
      }

      $sth->finish ();
      $sth2->finish ();

      close PEG;
```