A Practical Formalism for Vulnerability Comparison

Sophie Engle, Sean Whalen, Damien Howard,
Adam Carlson, Elliot Proebstel and Matt Bishop
University of California, Davis
{sjengle, shwhalen, djhoward, ajcarlson, proebstel, mabishop}@ucdavis.edu

August 22, 2006

Abstract

In our efforts to create a vulnerability classification scheme, we encountered a significant obstacle: ambiguous or conflicting notions of security, policy, vulnerabilities, and exploits. This paper defines a framework that explicitly and formally define these and related notions to facilitate vulnerability analysis. We focus our work on the concept of runtime vulnerabilities, exploits, and policy violations. We then provide an abstraction of these concepts to allow for quantitative comparison of vulnerabilities across systems. Finally, we discuss how this framework allows for practical evaluation of secure systems at a formal level.

1 Introduction

When one assesses the security of a system, the assessment is expressed in terms of "vulnerabilities". That term is usually taken to be a primitive, whose definition is either obvious or must be assumed. Unfortunately, the term is ambiguous, and leads to confusion.

For example, Fithen et al. [10] defined a vulnerability as "an unplanned system feature that an intruder may exploit, if he/she can establish certain preconditions, to achieve particular impacts on that system that violate its security policy." In contrast, the CVE effort [1, 3] defined a vulnerability as either "any fact about a computer system that is a legitimate security concern, but only within some contexts" or such a fact that was not intended to exist." (The latter is closer to Fithen et al's definition.) The President's Commission on Critical Infrastructure Protection defined the term to mean "a characteristic of a [system's] design, implementation, oroperation that renders it susceptible to destruction or incapacitation by a threat" [12]. Worse, the same author may use two different definitions at different times. Bishop [5] defined vulnerabilities as "bugs that enable users to violate the security policy"; as a "specific failure of controls" in procedures, technology, and management enabling unauthorized access or actions" [6]; and more formally, as a characterization of a vulnerable state that distinguishes it from a non-vulnerable state, a vulnerable state being "an authorized state from which an unauthorized state can be reached using authorized state transitions" [7]. Other definitions abound.

Underlying these definitions is an implicit notion of "policy". The policy defines what is authorized and what is unauthorized. But the notion of "policy" is also confusing when removed from a formal

context. Does it mean the *intended* policy of the site, that management has asked be enforced? Or does it mean the *actual* policy that the systems enforce? As a vulnerability is defined with respect to a policy, the answer can determine whether some problem should be considered a vulnerability.

This ambiguity is particularly pernicious when dealing with existing vulnerability databases. Those of CERT/CC, SecurityFocus, Secunia, and ISS X-Force are invaluable repositories of information[1] But their entries all tacitly assume that a particular policy is in place, and that the entry describes a violation of that policy. As an (admittedly extreme) example, a buffer overflow that gives one administrator privileges is not a vulnerability if the system policy says that any user can assume administrator rights.

In this paper, we present a model of vulnerabilities. This model allows us to define what a vulnerability is, and provides a basis for comparing different vulnerabilities. We also abstract the notion of a vulnerability from an instance on a particular system, so we can examine vulnerabilities on two different types of systems. Our approach potentially leads to a metric to measure the complexity of vulnerabilities.

We first define what a "system" is. We next review the standard, formal definition of "policy", and apply it to practice, leading to several notions of "policy". By examining disparities in these different types of policies, we can precisely define a specific class of vulnerabilities, namely those that occur as the system is being used (as opposed to design flaws). We examine the effects of having imperfect information about a system when working with vulnerabilities, and consider the implications for comparing vulnerabilities. We then discuss the implications of our work, and conclude with suggestions for future research.

2 Defining Systems

Before we define security, we define the concept of a system:

> **Definition 2.1:** Let the term **system** refer to anything that can directly modify the state space, and the term **machine** to refer to the hardware running the system.

Of course, we must also define what we mean by state space:

> **Definition 2.2:** Let the **machine state** be the set of all values stored in permanent or temporary memory resident on the machine. The **state space** represents every possible machine state and the transitions between them, as defined by the system.

Therefore, a system includes the operating system and any applications and hardware controls that can directly affect the state space during runtime. There is only one active machine state at any given time, and only one state space per system.

However, there could be multiple systems associated with a single machine. For example, consider a machine that can boot either a Windows or Linux operating system at startup. There is a system defined for when is Windows active, and a different system defined for when Linux is active.

Our reasoning behind this definition is directly motivated by how we want to use it. As we detail in

[1] .

[1] These databases can be found online at www.us-cert.gov, www.securityfocus.com, secunia.com, and xforce.iss.net respectively.

the next few sections, vulnerabilities exist in state space but their causes are rooted in the system. To describe these causes we use conditions:

> **Definition 2.3:** A **condition** represents a system property.

For our work, a system property is considered a primitive. Therefore, conditions describe a system at the lowest level of abstraction. We discuss systems in terms of their conditions:

> **Definition 2.4:** Let $\mathbb{D}_x$ represent the set of conditions present on some system x.

3 Defining Policy

Many notions of security include an implicit security policy assumption. We make this assumption explicit, and define security in the context of a security policy:

> **Definition 3.1:** The **security policy** (or just **policy**) defines what actions are or are not allowed for a user on a particular system, dividing the state space into a set of allowed and disallowed states. If a system is unable to transition to a disallowed state, we consider it to be **secure**. Otherwise, the system is **nonsecure**.

Unfortunately, the actual policy representation on a particular system may not always capture the intended policy. For the purposes of this paper, we limit our scope to more concrete levels of policy[2]. Specifically, we focus on the concepts of the chosen policy and actual protection domain of a system as defined by Carlson [8].

The chosen policy reflects the policy actually implemented on a given system:

> **Definition 3.2:** For every subject s, object o, and action a defined by system x, the **chosen policy (CP)** is defined as:
>
> $$\mathrm{CP}_x(s,o,a) = \begin{cases} yes & \text{if action } a \text{ on object } o \text{ by subject } s \text{ is allowed} \\ no & \text{if disallowed} \end{cases}$$

For example, suppose we have two user accounts `xander` and `yasmin`. In Windows XP only users with a computer administrator account can delete other user accounts. So if user account `xander` only has limited privileges then:

$$\mathrm{CP}_{\mathrm{winxp}}(\texttt{xander}, \texttt{yasmin}, \texttt{delete}\ ) = no$$

The problem arises when the chosen policy does not match what the system actually allows. On a nonsecure system, there exists one or more transitions to a disallowed state. For this reason, we need to capture the current protection domain of the system:

> **Definition 3.3:** For every subject s, object o, and action a defined by system x, the **actual protection domain (APD)** is defined as:
>
> $$\mathrm{APD}_x(s,o,a) = \begin{cases} yes & \text{if subject } s \text{ can perform action } a \text{ on object } o \\ no & \text{otherwise} \end{cases}$$

[2] See the appendix for more discussion on our adaptation of the policy hierarchy.

In short, the chosen policy provides us with what should and should not be allowed, while the actual protection domain provides us with what is and is not allowed in the system. Using CP and the APD, we can now provide a definition of runtime policy violations:

> **Definition 3.4:** A **runtime policy violation** occurs whenever the actual protection domain is inconsistent with the chosen policy, i.e. whenever $\mathrm{CP}_x(s, o, a) \neq \mathrm{APD}_x(s, o, a)$.

Consider the previous example. Suppose there exists a bug in the system which allows any user to gain the ability to delete user accounts. If `xander` exploits this bug then:

$$\mathrm{APD}_{\mathrm{winxp}}(\texttt{xander}, \texttt{yasmin}, \texttt{delete}\ ) = yes$$

Since this does not match the chosen policy, a runtime policy violation has occurred on that Windows XP system.

There are two types of runtime policy violations: those that enable a disallowed action and those that disable an allowed action. A denial of service policy violation falls under the second type.

4 Defining Vulnerabilities

A vulnerability exists whenever it is possible to violate the intended policy of a system or organization. However, this definition is too broad for the purposes of this paper[3]. Instead, we focus on runtime vulnerabilities:

> **Definition 4.1:** For some system x, let D_x be a nonempty set of conditions such that $D_x \subset \mathbb{D}_x$ and V_x be a nonempty set of runtime policy violations such that $CP_x \neq APD_x$. We define a **runtime vulnerability** R_x as the tuple (D_x, V_x) such that the set of conditions D_x enables the set of runtime policy violations V_x.

For example, consider the vulnerability which allowed user `xander` to delete the account `yasmin`. We define this runtime vulnerability as $R_x = (D_x, V_x)$ where D_x is the set of conditions on the Windows XP system that enables the policy violation $V_x = \{$ (`xander`, `yasmin`, `delete`) $\}$.

We hypothesize that given a system and its properties, it is possible to find a minimal and sound set of conditions for an instantiation of a vulnerability.

The transitions enabled by the runtime vulnerability exist in state space. However, the system controls the state space. Therefore the causes of the vulnerability (i.e. the conditions) exist in the system. In this context, vulnerabilities define why policy violations occur while exploits define how these violations occur:

> **Definition 4.2:** A **runtime exploit** is a nonempty set of transitions in state space enabled by a runtime vulnerability, which when executed violate policy.

A runtime exploit is different from attack tools or exploit code. It represents the transitions in state space which allow a policy violation to occur. For example, consider a buffer overflow which results in a policy violation. The vulnerability conditions would describe the lack of bounds checking, enabling the policy violation. The exploit, however, would describe the transitions which overflow the buffer and violate policy.

[3] More discussion on the definition of a vulnerability and why we chose to work with runtime vulnerabilities can be found in the appendix.

There must be at least one transition to a disallowed state for a vulnerability to exist. Since a runtime exploit defines these transitions, there exists at least one runtime exploit for every runtime vulnerability. Whether these transitions are ever taken by some attack tool is a different issue.

The instantiation of a runtime vulnerability is directly tied to a specific instantiation of policy on a system. This is an important implication of our runtime vulnerability definition. While two different runtime environments may have the same conditions, those conditions could affect different instantiations of policy. In that situation, there would be two distinct instantiations of a runtime vulnerability even though the set of conditions may be the same.

5 Defining Characteristics

Our goal is to analyze vulnerabilities across systems. However, while the sets of conditions and policy violations for a given system are finite, the sets of all possible systems and policies are not. Therefore it is impossible to define every possible instantiation of a runtime vulnerability for every possible system and policy. Furthermore, defining a vulnerability for a specific system and policy is useful only for that system.

Instead, we first define a finite set of systems for which we are interested in analyzing vulnerabilities. We then focus not on a single instantiation of a runtime vulnerability, but on describing a generalization of multiple instantiations on those systems. To do this, we use characteristics and symptoms:

> **Definition 5.1: Characteristics** are abstractions of nonempty sets of conditions, such that they are no longer tied to a specific instantiation of a system.
>
> **Definition 5.2: Symptoms** are abstractions of nonempty sets of policy violations, such that they are no longer tied to a specific instantiation of policy.

For example, suppose `[password-based authentication]` is a characteristic. This can represent how authentication is implemented across multiple systems. Similarly, a symptom like `[denial of service]` can describe the disabling of privileges without being specific to a chosen policy.

We use symptoms to define our vulnerability abstractions, but further refinement on how symptoms are defined is left for future work (see section 7).

The majority of our work focuses on the development of characteristics. Specifically, we are interested in characteristics which can represent similar conditions across a fixed set of systems. We then examine vulnerabilities at this level of abstraction, looking for which characteristics vulnerabilities share. The goal of these common characteristics is to provide insight into the nature of vulnerabilities, and how to prevent them[4].

To do this, however, we must provide a more precise definition of a characteristic. We first define characteristics assuming perfect knowledge, where we are able to enumerate all of the conditions for a system. Then we show how to approximate characteristics without enumerating conditions.

[4] The insight provided by the characteristic depends on the quality of characteristic defined. For more discussion, refer to section 7.

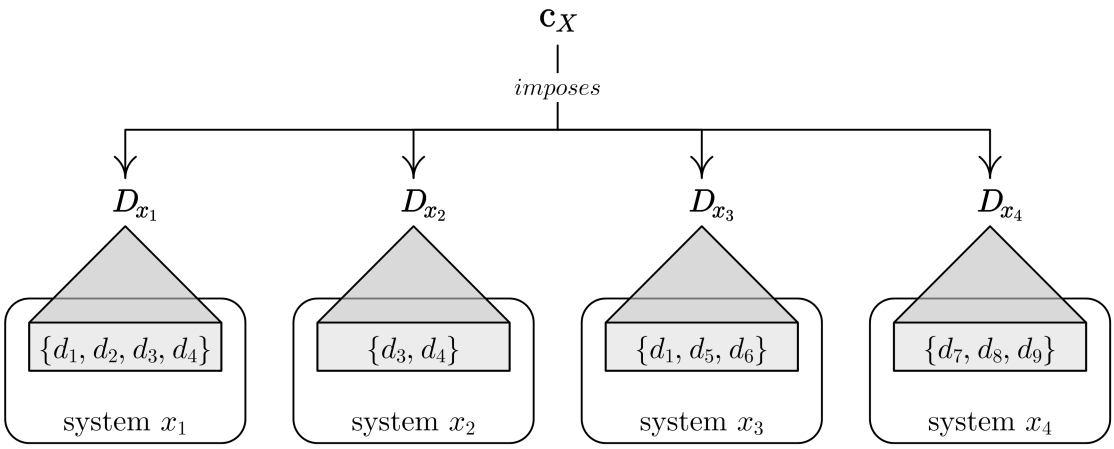

Figure 1: Example characteristic, c_X, for systems $X = \{x_1, x_2, x_3, x_4\}$ where $D_X = \{D_{x_1}, D_{x_2}, D_{x_3}, D_{x_4}\}$. The characteristic c_X imposes the set of conditions $D_{x_1} = \{d_1, d_2, d_3, d_4\}$ on system x_1, conditions $D_{x_2} = \{d_3, d_4\}$ on system x_2, conditions $D_{x_3} = \{d_1, d_5, d_6\}$ on system x_3, and conditions $D_{x_4} = \{d_7, d_8, d_9\}$ on system x_4.

5.1 Perfect Knowledge

Before we can precisely define characteristics, we must state our assumptions:

> **Definition 5.3:** The **perfect knowledge assumption** focuses on our ability to enumerate conditions. Specifically, we assume:
>
> 1. We have an explicitly defined nonempty set of systems $X = \{x_1, x_2, \ldots, x_n\}$.
> 2. For every system $x_i \in X$, we are able to enumerate the conditions $\mathbb{D}_{x_i}$.
> 3. For any runtime vulnerability $R_{x_i} = (D_{x_i}, V_{x_i})$ defined for some system $x_i \in X$, we are able to enumerate the set of conditions D_{x_i}.
>
> We do not, however, make any assumptions regarding the ability to enumerate policy violations.

In this context, a characteristic imposes a set of conditions on a system. The set of conditions imposed by a characteristic depends on the system in question. For example, consider the characteristic `[stack-based memory]`. This characteristic can represent conditions specific to the implementation of stack memory on a Linux system, or conditions specific to a Windows system. More formally:

> **Definition 5.4:** Let $D_X = \{D_{x_1}, D_{x_2}, \ldots, D_{x_n}\}$ be a nonempty set[5]of conditions defined for a nonempty set of systems $X = \{x_1, x_2, \ldots, x_n\}$ such that for every set of conditions $D_{x_i} \in D_X$ we have $D_{x_i} \subset \mathbb{D}_{x_i}$. We define a **characteristic** $c_X = D_X$ such that c_X imposes the set of conditions D_{x_i} on system x_i for every $x_i \in X$ and $D_{x_i} \in D_X$.

Figure 1 illustrates a characteristic defined for a small set of systems.

5.2 Imperfect Knowledge

The perfect knowledge assumption is inconvenient in practice. First, we may not have a well-defined set of systems. For example, we might be interested in the set of systems that run Linux or Windows

[5] Notice that D_X is actually a set of sets of conditions. It is not the union of conditions $\bigcup_{i=1}^{n} D_{x_i}$.

operating systems. This includes many different operating system versions, running applications, and hardware configurations. Furthermore, enumerating conditions in general is impractical without a well-defined set of systems. Therefore we define a system oracle function to avoid having to enumerate conditions on a system, and different instantiations of runtime vulnerabilities:

> **Definition 5.5:** The **system oracle** is a function $\mathcal{F}_x(c_X) = D_x$ such that D_x is the set of conditions which must hold on system $x \in X$ for the characteristic c_X to exist.

In practice, the system oracle can be approximated by an expert in the field. Given a characteristic, a systems administrator should be able to decide what conditions have to hold for that characteristic to exist on a particular system. For example, consider the characteristic `[requires password-based authentication]`. The administrator can decide that to satisfy this characteristic, a Windows system needs to have user accounts with passwords defined, accounts which do not require passwords disabled (like the guest account), and no biometric peripherals installed.

This function only needs to be defined for systems of interest, and can be extended to new systems as needed. Also, the system oracle only needs to be as well-defined as necessary. This allows characteristics to be tailored to the desired level of detail. If we are concerned with a loosely defined set of systems and conditions, then our system oracle can be loosely defined.

The concept of a characteristic can be redefined using the system oracle:

> **Definition 5.6:** Let $c_X = D_X$ be a characteristic defined for the nonempty set of systems $X = \{x_1, x_2, \ldots, x_n\}$. Then $D_X = \{\mathcal{F}_{x_1}(c_X), \mathcal{F}_{x_2}(c_X), \ldots, \mathcal{F}_{x_n}(c_X)\}$ is a nonempty set of conditions such that for every $x_i \in X$, $\mathcal{F}_{x_i}(c_X) = D_{x_i}$.

Therefore, we can define a vulnerability indicator function for characteristics:

> **Definition 5.7:** Let $R_x = (D_x, V_x)$ be a runtime vulnerability defined for a system $x \in X$, and $c_X = D_X$ be a characteristic. We define the **vulnerability indicator function** for the characteristic c_X as:
>
> $$\mathcal{I}_{R_x}(c_X) = \begin{cases} true & \text{if } \mathcal{F}_x(c_X) \subset D_X \text{ for system } x \in X \\ false & \text{otherwise} \end{cases}$$

If we allow the indicator function and system oracle to operate as black boxes, we can use characteristics to capture multiple instantiations of a runtime vulnerability without having to explicitly define each instantiation.

5.3 Characteristic Sets

Our goal is to be able to define vulnerabilities in terms of characteristics instead of conditions. The universal characteristic set is another step towards this goal:

> **Definition 5.8:** The **universal characteristic set** $\mathbb{U}$ is a fixed, nonempty set of characteristics under consideration.

The universal characteristic set does not necessarily include every possible characteristic. It provides the characteristics being considered at a particular moment. As interests change, this set may shrink or grow. Essentially, $\mathbb{U}$ is a subset of all possible characteristics tailored for a specific situation.

For example, suppose you are only interested in characteristics related to buffer overflows. Then the universal characteristic set $\mathbb{U}$ may include characteristics like `[stack-based memory]`, `[lack of bounds checking]`, `[accepts user input]`, and `[allows execution of arbitrary code]`.

The set $\mathbb{U}$ provides the frame of reference used when defining a characteristic set or comparing vulnerabilities. Given $\mathbb{U}$, it is possible to derive the maximal characteristic set of a vulnerability:

> **Definition 5.9:** Let $\mathbb{U}$ be a universal characteristic set, and $R_x = (D_x, V_x)$ be a runtime vulnerability. The **maximal characteristic set** $C_{D_x} \subset \mathbb{U}$ is the set of all characteristics $c_X \in \mathbb{U}$ that satisfy $\mathcal{I}_{R_x}(c_X) = true$.

For example, suppose we have a simple buffer overflow which causes an application to crash but does not require user input or execution of arbitrary code. Using the example $\mathbb{U}$ from before, the maximal characteristic set for the associated vulnerability is:

$$C_{D_x} = \{\texttt{[stack-based memory]}, \texttt{[lack of bounds checking]}\}$$

However, suppose there is a race condition which potentially enables privilege escalation. Since none of the characteristics in the given $\mathbb{U}$ describe this situation, $C_{D_x} = \emptyset$ for this vulnerability.

There is only one maximal characteristic set for a vulnerability given a particular universal characteristic set. This property is useful for comparing vulnerabilities.

5.4 Symptom Sets

In addition to characteristics, we also provide symptoms as an abstraction of policy violations. More specifically:

> **Definition 5.10:** A **symptom** s_v is an abstraction or generalization of a nonempty set of policy violations v. A **symptom set** S_V is the set of symptoms such that $v \subset V$ for every $s_v \in S_V$.

For example, suppose `(ann, config.txt, write)` and `(bob, config.txt, write)` are policy violations. These violations could be represented by single symptom "`[unauthorized users allowed to write config.txt]`."

5.5 Vulnerability Comparison

The goal of characteristics is to allow us to compare vulnerabilities without having to enumerate conditions. By identifying which characteristics are common to multiple vulnerabilities across multiple systems, we hope to gain insight into the nature of vulnerabilities and how to prevent them.

However, before we reach this goal we must define the abstraction of a vulnerability:

> **Definition 5.11:** Let $R_x = (D_x, V_x)$ be an instantiation of a runtime vulnerability. We define $R_{\mathbb{U}} = (C_{D_x}, S_{V_x})$ to be the **runtime vulnerability abstraction** of R_x where C_{D_x} is the characteristic set for R_x given the universal characteristic set $\mathbb{U}$, and S_{V_x} is the symptom set for V_x.

The abstraction of R_x depends on the universal characteristic set $\mathbb{U}$. It is possible that there are no characteristics in $\mathbb{U}$ that describe R_x. In fact, characteristics may not capture all possible conditions,

as we are only interested in identifying characteristics which we find to be practical, interesting, and common across implementations[6]. Similarly, symptoms may not capture all possible policy violations. Therefore the sets C_{D_x} and S_{V_x} may be empty depending on the characteristics and symptoms in question.

Consider again a simple buffer overflow which causes an application (say `webserver`) to crash. For the example set $\mathbb{U}$ defined earlier, the runtime vulnerability abstraction is $R_{\mathbb{U}} = (C_{D_x}, S_{V_x})$ where:

$$\begin{aligned} C_{D_x} &= \{\texttt{[stack-based memory]}, \texttt{[lack of bounds checking]}\} \\ S_{V_x} &= \{\texttt{[access to webserver disabled]}\} \end{aligned}$$

Since characteristics and symptoms are concepts which can span systems, multiple runtime vulnerabilities may have the same abstraction. It is at this stage that we can finally define a class of vulnerabilities:

> **Definition 5.12:** Let $\{R_1, R_2, \ldots R_n\}$ be a set of runtime vulnerability abstractions such that given some universal characteristic set $\mathbb{U}$ we have $R_i = (C_i, S_i)$ for every $i = 1 \ldots n$. We define $\mathbb{C} = (C, S)$ to be a **runtime vulnerability class** (or just **class**) if $C \subset \bigcap_{i=1}^{n} C_i$ and $S = \bigcup_{j=1}^{n} S_j$.

A runtime vulnerability class is based off the intersection of characteristics, while the set S represents the potential policy violations enabled by those characteristics.

For example, if we want to define a class of generic buffer overflow vulnerabilities, we might define $\mathbb{C} = (C, S)$ to be based on:

$$C = \{\texttt{[lack of bounds checking]}\}$$

Our example runtime vulnerability abstraction falls into this class. As such, the symptom $S_{V_x} = \{\texttt{[access to webserver disabled]}\}$ becomes an element of S.

We can also define a stronger relationship between vulnerabilities:

> **Definition 5.13:** Let $\{R_1, R_2, \ldots R_n\}$ be a set of runtime vulnerability abstractions such that given some universal characteristic set $\mathbb{U}$ we have $R_i = (C_i, S_i)$ for every $i = 1 \ldots n$. We define $\mathbb{C}^e = (C, S)$ to be a **runtime vulnerability equivalence class** (or just **equivalence class**) if $\mathbb{C}^e$ is a class such that for every $i = 1 \ldots n$ we have $C_i = \bigcap_{j \neq i} C_j$.

For example, suppose our simple buffer overflow exists on multiple versions of the web server. The conditions may change depending on the version of the server installed, causing multiple runtime vulnerabilities. However, if every set of conditions abstracts to the same maximal characteristic set, then we can say these instantiations belong to the same equivalence class. This allows us to group runtime vulnerabilities across different system instantiations that we abstractly consider to be a single vulnerability.

It is not always necessary to explicitly define $\mathbb{U}$ for these relationships. For classes of vulnerabilities, each abstraction could use a different universal set. However, the characteristics defining the class must be common to each individual $\mathbb{U}$. For any equivalence class of vulnerabilities, there exists

[6] Judgement calls as to what is practical, interesting, and common must be made by an expert in the field.

some $\mathbb{U}$ where the equality condition holds. The point is that the equivalence condition may not hold for every possible set $\mathbb{U}$.

Classes do not always have to be defined by characteristics. Instead, we can create a symptom class:

> **Definition 5.14:** Let $\{R_1, R_2, \ldots R_n\}$ be a set of runtime vulnerability abstractions such that given some universal characteristic set $\mathbb{U}$ we have $R_i = (C_i, S_i)$ for every $i = 1 \ldots n$. We define $\mathbb{S} = (C, S)$ to be a **runtime vulnerability symptom class** (or just **symptom class**) if $C = \bigcup_{i=1}^{n} C_i$ and $S \subset \bigcap_{j=1}^{n} S_j$.

A symptom class is based on the symptoms of the vulnerability, while the set C represents characteristics which could potentially enable those symptoms.

There is also the similar notion of a symptom equivalence class:

> **Definition 5.15:** Let $\{R_1, R_2, \ldots R_n\}$ be a set of runtime vulnerability abstractions such that given some universal characteristic set $\mathbb{U}$ we have $R_i = (C_i, S_i)$ for every $i = 1 \ldots n$. We define $\mathbb{S}^e = (C, S)$ to be a **runtime vulnerability symptom equivalence class** (or just **symptom equivalence class**) if $\mathbb{S}^e$ is a class such that for every $i = 1 \ldots n$ we have $S_i = \bigcap_{j \neq i} S_j$.

Each of these relationships allow us to analyze vulnerabilities in different ways. For example, suppose there are several vulnerabilities which fall under the same class. The shared characteristics of that class would be prime targets to eliminate to prevent that entire class of characteristics from occurring. It may also provide insight into why that class of vulnerabilities exist on systems, hence providing insight into how to prevent or fix them. It also tells us what the potential symptoms these characteristics enable, allowing us to assign a measure of severity to the class of vulnerabilities.

Symptom classes provide a different type of insight. We can look at a symptom class for severe symptoms, and see which characteristics tend to enable those symptoms. If those characteristics exist on our systems, it may be wise to eliminate them even in the absence of a vulnerability. It could be that the vulnerability has not yet been discovered on that system. This allows administrators to take a preventative versus reactive approach to vulnerabilities.

6 Properties

We have found these definitions to have several properties which we found beneficial to vulnerability analysis. Many of these properties either provide flexibility or reduce ambiguity.

6.1 Level of Detail

Our goal with characteristics is to avoid enumerating conditions for systems and vulnerabilities. However, if we choose to enumerate conditions our definitions still hold. This flexibility lets us tailor how we discuss vulnerabilities to our specific goals. For example, if we want to analyze vulnerabilities across systems we can use characteristics. However, if we want to closely analyze vulnerabilities for a single high-assurance system and a fixed policy, we may want to enumerate the conditions for that system.

6.2 Layers of Abstraction

We have two main levels of abstraction. The lowest level of abstraction is specific to the instantiation of a system or policy. Conditions, runtime policy violations, and runtime vulnerabilities exist at this level of abstraction. The higher level of abstraction is not specific to an instantiation of a system or policy. Characteristics, symptoms, vulnerability abstractions, and classes exist at this level of abstraction.

The interface between these levels of abstraction is provided by the system oracle and indicator function[7]. By using an approximation of these functions, we can avoid working at the instantiation-specific level of abstraction.

6.3 Human Intervention

We assume vulnerability analysis requires human intervention at some point. In our definitions, we separate those concepts which require human intervention or judgement calls from those that do not.

Even if the perfect knowledge assumption holds, there are certain concepts that must be defined by an expert in the field. Essentially, every element at our lowest level of abstraction must be defined. A finite set of systems must be identified, and the conditions for each system enumerated. Specific instantiations of runtime vulnerabilities must be created, with their conditions explicitly listed. Finally, characteristics and the conditions they impose must be defined.

However, there is no need to approximate the system oracle and vulnerability indicator functions. Instead, the conditions, runtime vulnerabilities, and characteristics already defined determine the output of these functions.

This changes slightly when the perfect knowledge assumption does not hold, as illustrated by figure 2. Since we do not enumerate conditions, we do not need to define them or any other instantiation-specific element. However, an expert in the field does need to approximate the system oracle and vulnerability indicator functions.

In each situation, whether a vulnerability belongs to a class does not require any judgement calls, nor does defining the characteristic set. Given the universal characteristic set and system oracle, these operations are automatic and repeatable. This repeatability is guaranteed given the elements manually-defined in either the perfect knowledge or imperfect knowledge settings.

So far, we have not mentioned how policy violations are identified. Since this is very dependent on policy implementation, we leave this discussion for the appendix.

6.4 Measure of Security

Our methods also provide a way to measure the severity of a particular vulnerability, and the security of a system.

A simple approach to measuring a vulnerability's severity is to use the size of the set of policy violations enabled by the vulnerability. Using this method, two types of vulnerabilities will always

[7] The system oracle and indicator function provide the interface between conditions and characteristics. For more discussion on the relationship of policy violations and symptoms, see section 7.

	perfect knowledge	imperfect knowledge
manually defined	• set of systems X • $\forall\ x \in X$ conditions $\mathbb{D}_x$ • set of runtime vulnerabilities R • $\forall\ R_x \in R$ conditions D_x • universal characteristic set $\mathbb{U}$ • $\forall\ c_X \in \mathbb{U}$ conditions D_X	• loosely defined set of systems X • universal characteristic set $\mathbb{U}$ • $\forall\ c_X \in \mathbb{U}$ and $x \in X$, give approximation of $\mathcal{F}_x(c_X)$ • $\forall\ c_X \in \mathbb{U}$ and $R \in R_{\mathbb{U}}$, give approximation of $\mathcal{I}_R(c_X)$
automatic	• $\forall\ c_X \in \mathbb{U}$ and $x \in X$ give $\mathcal{F}_x(c_X)$ • $\forall\ c_X \in \mathbb{U}$ and $\forall\ R_x \in R$ give $\mathcal{I}_{R_x}(c_X)$ • $\forall\ R_x \in R$ characteristic set C_{D_x} • membership of every $R_{\mathbb{U}}$ to any $\mathbb{C}$, $\mathbb{C}^e$, $\mathbb{S}$, or $\mathbb{S}^e$	• $\forall\ R_{\mathbb{U}}$ characteristic set C_{D_x} • membership of every $R_{\mathbb{U}}$ to any $\mathbb{C}$, $\mathbb{C}^e$, $\mathbb{S}$, or $\mathbb{S}^e$

Figure 2: Division of concepts based on the assumptions made and amount of human intervention required. Concepts that must be defined or approximated by an expert in the field are listed in the **manual** category.

be ranked as severe:

1. Vulnerabilities which allow a user to gain root privileges.
2. Vulnerabilities which deny access to all system objects for all users (denial of service).

Using this simple method of measuring security, both groups of vulnerabilities have the same severity[8]. For some organizations, this may be an undesired result. If confidentiality is more important than availability, than the denial of service vulnerabilities should be ranked lower (or visa versa).

It may be more worthwhile to assign weights to different types of policy violations. In fact, weights can be applied even to specific subjects, objects, and actions to fine-tune the measure of how critical a particular vulnerability may be. For example, having read access to a password file may be more severe than a configuration file.

The measure of system security can be done similarly. A simple count of the number of policy violations present could suffice as a measure of security, or weights can be applied as needed.

Security may also be measured using symptom sets, to avoid having to enumerate specific policy violations.

7 Future Work

During the course of our work, we ran into several issues which we have yet to fully resolve. We outline a few of these areas which we would like to further explore.

Our goal is to extend the notion of characteristics to unambiguously classify vulnerabilities [5].

[8] The first type allows the attacker to enable privileges for all users, whereas the second type allows the disabling of privileges for all users. Therefore the worst-case scenarios result in the same number of policy violations.

To allow even greater flexibility, we define abstract characteristics which add another level of abstraction to our notion of characteristics defined in this paper. We then hierarchically organize these abstract characteristics using a grammar. So far we have classified several dozen network protocols (see [16]) and host vulnerabilities using this method.

We also have initial results mapping our work to the Requires/Provides model [15], Take-Grant [6, 4], and a correlation method of IDS signatures [17].

In this paper, we make little attempt to actually identify vulnerability characteristics. It is possible to define meaningless characteristics. This poses the question, what makes a characteristic meaningful? For example, we could define the single characteristic `[system has power]` to describe all possible vulnerabilities, but it provides little insight into why, where, or how those vulnerabilities occur. We are interested in looking at various sets of characteristics and the insight they provide, to create a guideline for defining useful characteristics. Such guidelines would further strengthen our classification scheme, allowing others to define meaningful characteristics tailored to different interests.

Our focus in this paper has been on characteristics, with less emphasis on symptoms. However, we believe that to fully understand the nature of vulnerabilities, we must also understand the symptoms they enable.

Part of our ongoing work on this project will be to further examine symptoms and the insights they can provide. Specifically, we must further refine the interface between symptoms and runtime policy violations. This depends on the ability to implement a CP and APD to detect policy violations, which is also discussed further in the appendix.

Our definition of a runtime vulnerability is based off the policy hierarchy given in A. The policy hierarchy itself has many interesting implications. Another branch of our work will be to explore these implications and how they affect our view of vulnerabilities.

8 Related Work

Many other authors have explored the notions of vulnerabilities. Most closely related to our work are the attempts to compare the severity of different vulnerabilities, to measure the impact of vulnerabilities, and to analyze systems looking for vulnerabilities. For example, Alves-Foss and Barbosa, for example, define a System Vulnerability Index that combines a number of factors to determine how vulnerable a system is to attack [2]. Howard, Pincus, and Wing develop a notion of an "attack surface" and use a three-dimensional model to study the vulnerability of a system [11]. Fithen, Hernan, O'Rourke, and Shinberg focus on a different level of abstraction and use a graph-based methodology to analyze specific preconditions (corresponding to our conditions) and postconditions (corresponding to a system-specific instantiation of our symptoms) [10].

Policy issues also come into play. Dobson and McDermid develop a framework of policies [9]; our work assumes the actual and intended policies are established, even if by an oracle. Schneider studies the enforceability of policies in a formal, theoretical vein [13]. Sterne tackles the ambiguity of the term "security policy" by introducing the terms "security policy objective", "organizational security policy", and "automated security policy" [14]. Conceptually, his notions vaguely similar to our notion of "chosen policy", in that when taken together they describe what the system is to be allowed to do, but his goals and analysis are orthogonal to ours.

9 Practical Significance

Despite the amount of formalism behind our definitions, our work has immediate practical significance to the community. We currently have a working vulnerability classification framework based on this work. Our results with network protocols have highlighted some of the common design issues that result in vulnerabilities. Creating characteristics for various types of buffer overflow vulnerabilities has given us greater insight into these vulnerabilities and their potential defenses.

Our framework allows for comparison of vulnerabilities in a consistent, repeatable manner (see 6.3). Given a precisely defined universal characteristic set and system oracle function, the derivation of a vulnerability abstraction is always consistent. Given multiple vulnerability abstractions, the derivation of class membership is always consistent. Therefore we know exactly what must be defined by a classification scheme to always be consistent and repeatable.

Many of our notions are extensible and flexible. This allows our framework to be tailored for specific needs and situations. We can reason about vulnerabilities at a level of abstraction appropriate for the level of detail required. The same framework allows for analysis of vulnerabilities across multiple systems as well as detailed analysis of vulnerabilities on a single high-assurance system.

The measure of security supported by our framework also has interesting applications. Many vulnerability databases assign severity based on metrics they consider important. However, this may not match the policy of other organizations. By allowing other organizations to define weights for different symptoms, vulnerability severity can be tailored for each organization. The weights can be modified without requiring changes to the symptom set of a vulnerability, allowing a single database to serve the needs for diverse organizations.

Another immediate advantage to our framework is the foundation it provides. Many of our formalisms map to already established intuitions, providing a formal grounding for previously ambiguous notions without requiring significant change.

10 Conclusion

Our goal was to unambiguously define a vulnerability and related notions. We choose to focus on runtime vulnerabilities, which we defined to be the set of conditions and the runtime policy violations they enable. To do this, we also had to define concepts such as a system, machine state, and state space. We also defined policy and runtime policy violations in terms of a chosen policy versus the actual protection domain.

However, runtime vulnerabilities are tied to a specific instantiation of a system and policy. To allow discussion of vulnerabilities at a more abstract level, we defined characteristics and symptoms. We let characteristics abstract conditions away from system-specific details, and symptoms abstract away from instantiation-specific policy violations.

In doing this, we created two distinct levels of abstraction. We defined the system oracle and vulnerability indicator function to tie together concepts across levels of abstraction.

We also specified where judgement calls must be made when analyzing vulnerabilities. This allows us to identify which concepts must be defined to achieve repeatability in our analysis.

We finally define vulnerability classes, allowing us to reason about multiple vulnerability abstractions across systems. This allows us insight into the common causes of vulnerabilities, which is necessary to prevent them.

Vulnerabilities will always exist, if for no other reason than fallible humans will misconfigure systems or make errors in their design, implementation, deployment, or maintenance. It is critical that we deepen our understanding of the character of vulnerabilities, so we can detect and eliminate or ameliorate them. This work extends our knowledge of the nature of vulnerabilities.

References

[1] Common vulnerabilities and exposures. Online at http://cve.mitre.org/about/terminology.html.

[2] Alves-Foss, J., and Barbosa, S. Assessing computer security vulnerability. *ACM SIGOPS Operating Systems Review 29*, 3 (July 1995), 3–13. Available online at http://doi.acm.org/10.1145/206826.206829.

[3] Baker, D. W., Christey, S. M., Hill, W. H., and Mann, D. E. The development of a common vulnerability enumeration. In *Recent Advances in Intrusion Detection (RAID)* (September 1999). Available online at http://www.raid-symposium.org/raid99/PAPERS/Hill.pdf.

[4] Bishop, M. Theft of information in the take-grant protection model. *Journal of Computer Security 3*, 4 (1995), 283–308.

[5] Bishop, M. Vulnerability analysis: An extended abstract. In *Recent Advances in Intrusion Detection (RAID)* (September 1999), pp. 125–136. Available online at http://nob.cs.ucdavis.edu/ bishop/papers/1999-vulclass/1999-vulclass.pdf.

[6] Bishop, M. *Computer Security: Art and Science*. Addison-Wesley, 2003.

[7] Bishop, M., and Bailey, D. A critical analysis of vulnerability taxonomies. Tech. Rep. CSE-96-11, UC Davis Department of Computer Science, September 1996. Available online at http://www.cs.ucdavis.edu/research/tech-reports/1996/CSE-96-11.pdf.

[8] Carlson, A. The unifying policy hierarchy model. Master's thesis, University of California at Davis, 2006.

[9] Dobson, J. E., and McDermid, J. A. A framework for expressing models of security policy. In *Proceedings of IEEE Symposium on Security and Privacy* (May 1989), pp. 229–239. Available online at http://ieeexplore.ieee.org/iel2/243/1514/00036297.pdf.

[10] Fithen, W. L., Hernan, S. V., O'Rourke, P. F., and Shinberg, D. A. Formal modeling of vulnerability. *Bell Labs Technical Journal 8*, 4 (2004), 173–186. Available online at http://dx.doi.org/10.1002/bltj.10094.

[11] Howard, M., Pincus, J., and Wing, J. Measuring relative attack surfaces. Tech. Rep. CMU-CS-03-169, School of Computer Science at Carnegie Mellon University, August 2003. Available online at http://reports-archive.adm.cs.cmu.edu/anon/2003/CMU-CS-03-169.pdf.

[12] Report of the President's Commission on Critical Infrastructure Protection. *Critical Foundations: Protecting America's Infrastructures*, October 1997.

[13] Schneider, F. B. Enforceable security policies. *ACM Transactions on Information and System Security (TISSEC) 3*, 1 (February 2000), 30–50. Available online at http://doi.acm.org/10.1145/353323.353382.

[14] Sterne, D. F. On the buzzword 'security policy'. In *Proceedings of IEEE Computer Society Symposium on Research in Security and Privacy* (May 1991), pp. 219–230. Available online at http://ieeexplore.ieee.org/iel2/349/3628/00130789.pdf.

[15] Templeton, S., and Levitt, K. A requires/provides model for computer attacks. In *New Security Paradigm Workshop* (September 2000), pp. 31–38. Available online at http://doi.acm.org/10.1145/366173.366187.

[16] Whalen, S., Engle, S., and Bishop, M. Protocol vulnerability analysis. Tech. Rep. CSE-2005-4, UC Davis Department of Computer Science, May 2005. Available online at http://www.cs.ucdavis.edu/research/tech-reports/2005/CSE-2005-4.pdf.

[17] Zhou, J., Heckman, M., Reynolds, B., Carlson, A., and Bishop, M. Modelling network intrusion detection alerts for correlation. Submitted to *ACM Transactions on Information and System Security (TISSEC)*, 2006.

Appendix

A Policy Hierarchy

Intended policy and implemented policy often disagree. The policy hierarchy by Carlson provides a basis for studying this difference [8]. We present certin key concepts of that hierarchy here.

Figure 3 shows the four levels of policy. The oracle policy is defined at the highest level, providing a perfect policy representation:

> **Definition A.1:** The **Oracle Policy (OP)** completely and accurately captures the intent of policy makers. For every possible subject s, object o, and action a:
>
> $$\mathrm{OP}(s,o,a) = \begin{cases} yes & \text{if action } a \text{ on object } o \text{ by subject } s \text{ is allowed} \\ no & \text{if disallowed} \end{cases}$$

The oracle policy is able to provide a policy decision for every possible subject, object, and action. These elements may reside across systems or outside any given system. For example, suppose Xander is assigned the user account `xander` and Yasmin is assigned the user account `yasmin` on some system. Ideally, Xander should not be able to authenticate to the system as `yasmin`. Therefore for the person Xander and user account `yasmin` we have:

$$\begin{aligned} \mathrm{OP}(\text{ Xander, } \texttt{xander}, \texttt{authenticate} \text{ }) &= yes \\ \mathrm{OP}(\text{ Xander, } \texttt{yasmin}, \texttt{authenticate} \text{ }) &= no \end{aligned}$$

We would have similar policy decisions for Yasmin. However, this requires the oracle policy to have knowledge of the users Xander and Yasmin and not just the user accounts. This is infeasible with current policy implementations. Since currently methods are often unable to fully capture the intent of policy makers, we must define the feasible oracle policy:

> **Definition A.2:** The **Feasible Oracle Policy (FOP)** takes into account the restrictions of policy implementation. For every subject s, object o, and action a defined by some system x:
>
> $$\mathrm{FOP}_x(s,o,a) = \begin{cases} yes & \text{if action } a \text{ on object } o \text{ by subject } s \text{ is allowed} \\ no & \text{if disallowed} \end{cases}$$

The disconnect between oracle policy and the feasible oracle policy might also be caused by environmental limitations. For example, suppose we have an embedded system with limited performance

Level	Policy Type
4	Oracle Policy (OP)
3	Feasible Oracle Policy (FOP)
2	Chosen Policy (CP)
1	Actual Protection Domain (APD)

Violation	Cause
OP $\neq$ FOP	Feasibility Restrictions
FOP $\neq$ CP	Configuration Choices/Errors
CP $\neq$ APD	Implementation Errors

Figure 3: Summary of different policy and policy violation types.

capability and memory. It may be impossible to provide the level of security required by the intended policy given those constraints. In this situation, tradeoffs may have to be made to make policy achievable.

The feasible oracle policy should be identical to the oracle policy except in those situations where available access controls cannot properly implement the oracle policy. For example, suppose the account `xander` should not have access to the file `passwords.txt`. Since this involves elements defined by the system, we should have:

$$\begin{aligned} \text{OP}(\ \texttt{xander}, \texttt{passwords.txt}, \texttt{read}\ ) &= \textit{no} \\ \text{FOP}_x(\ \texttt{xander}, \texttt{passwords.txt}, \texttt{read}\ ) &= \textit{no} \end{aligned}$$

However, since the system is only aware of user accounts and not the actual users:

$$\begin{aligned} \text{OP}(\ \text{Xander}, \texttt{xander}, \texttt{authenticate}\ ) &= \textit{yes} \\ \text{FOP}_x(\ \text{Xander}, \texttt{xander}, \texttt{authenticate}\ ) &= \textit{unknown} \end{aligned}$$

The feasible oracle policy is only defined for those subjects, objects, and actions which can be captured by the system. As such, it may not accurately capture the intent of the policy makers, and may not be able to provide correct policy decisions for elements undefined by the system. Instead, it approximates the oracle policy to the greatest amount possible.

Specification does not always match implementation. Sometimes, the policy actually implemented on a system does not match the feasible policy. It is for this reason we define the **chosen policy** (see definition 3.2).

Finally, what is actually allowed by the system may not match what should be allowed. The **actual protection domain** captures this disconnect (see definition 3.3). The APD, like the CP and FOP, is only able to reason about subjects, objects, and actions defined by the system.

B Vulnerabilities

In general, a policy violation occurs whenever adjacent levels are inconsistent, and vulnerabilities cause these policy violations. However, we choose to work exclusively with runtime vulnerabilities and runtime policy violations (where CP $\neq$ APD).

Our decision was motivated by interest and feasibility. We are primarily interested in violations caused by bugs in a system, not violations caused by incorrect configurations. Therefore we are not interested in violations where FOP $\neq$ CP. Furthermore, the chosen policy can be inferred to some degree for modern systems. Most systems assign allowed or disallowed permissions to user accounts. If the systems implement some sort of access control matrix (ACM) or access control list (ACL) we can use these mechanisms to infer the chosen policy. Without access to the policy maker or the policy specification, it is difficult to infer the feasible oracle policy and the oracle policy.

Theoretically, the APD can be tracked through perfect auditing of all actions performed. More realistically, however, the APD can be approximated by someone familiar with the system. By careful analysis of how different vulnerability conditions affect the system, we can enumerate the possible policy violations. For example, if we see that a buffer overflow allows a user to circumvent the access controls of a system, we can estimate how this would effect the APD.

Given that the CP may be inferred and the APD approximated, we decided that runtime vulnerabilities would be both feasible and interesting to analyze.

C Summary of Notation

The following tables provide a summary of notation used in this paper.

Notation	Description	Def.
x	A system	2.1
X	A set of systems	2.1
$\mathbb{D}_x$	The set of all conditions present on system x	2.4
CP_x	The chosen policy on system x	3.2
APD_x	The actual protection domain on system x	3.3
R_x	A runtime vulnerability for some system x	4.1
D_x	A nonempty set of conditions such that $D_x \in \mathbb{D}_x$	4.1
V_x	A nonempty set of runtime policy violations defined for system x	4.1
D_X	A nonempty set of conditions defined for the set of systems X	5.4
c_X	A characteristic defined for the set of systems X	5.4
$\mathcal{F}_x$	A system oracle function defined for system x	5.5
$\mathcal{I}_{R_x}$	A vulnerability indicator function defined for runtime vulnerability R_x	5.7
$\mathbb{U}$	A universal characteristic set	5.8
C_{D_x}	A maximal characteristic set	5.9
s_v	A symptom	5.10
S_V	A set of symptoms	5.10
$R_{\mathbb{U}}$	A runtime vulnerability abstraction given $\mathbb{U}$	5.11
$\mathbb{C}$	A runtime vulnerability class	5.12
$\mathbb{C}^e$	A runtime vulnerability equivalence class	5.13
$\mathbb{S}$	A runtime vulnerability symptom class	5.14
$\mathbb{S}^e$	A runtime vulnerability symptom equivalence class	5.15