A Model for Vulnerability Analysis and Classification

Sophie Engle and Matt Bishop
Department of Computer Science
University of California, Davis
One Shields Avenue, Davis, CA, 95616 USA
{engle,bishop}@cs.ucdavis.edu

Abstract

In this paper, we present a model for vulnerability analysis that enables us to mitigate the complexity of modern systems through well-defined layers of abstraction. We use this model to build a new framework for vulnerability classification. Finally, we present our results classifying buffer overflow vulnerabilities.

1. Introduction

The study of system vulnerabilities has been obscured by a lack of precision and clarity. Providing a formal model for vulnerability analysis may help remove this ambiguity by providing precisely-defined concepts and relationships between these concepts. However, it is a widely held belief that systems are too complex for formal models. For example, the state space of a modern computer is too complex to enumerate. Most Turing machines are given in terms of high-level algorithms instead of states, transitions, and configurations.

Further, it is not clear how security fits into the classic formal model for modern computers – the Universal Turing Machine. Does a vulnerability exist on the tape of the Turing machine, or in the system control? How is a vulnerability defined at this level?

While modern systems are growing in complexity, it is possible to build a precise model for vulnerability analysis. Just as abstraction of well-defined components is used when defining Turing machines, we can use abstraction to describe system vulnerabilities in a practical and meaningful way.

Our research focuses on building such a model. We then use this model to address another need – a robust vulnerability classification framework.

Most current classification frameworks classify errors, bugs, or faults with tend to lead to vulnerabilities. When these classification schemes are then used by vulnerability databases, vulnerabilities are often reduced to just one of these components. This obscures the relationships between vulnerabilities when multiple errors lead to a single vulnerability, or when policy issues and not implementation issues are to blame. Arguably the most important part of the vulnerability – the associated policy violation – is not even mentioned.

Our model addresses these issues by providing a vulnerability classification framework which integrates important aspects of a vulnerability and its related policy violations.

The next section introduces our model, beginning with the definition of a system. We then introduce our vulnerability classification framework. We illustrate the power of this framework by presenting our initial results on classifying buffer overflow vulnerabilities. Finally, we compare our work to that of others.

2. Model for Vulnerability Analysis

Our model for vulnerability analysis begins with the states and transitions of a Universal Turing machine. We use these elements to define systems, security, policy, vulnerabilities, and related concepts. Each level of abstraction mitigates the complexity of dealing directly with state space. Similar to how a high-level algorithm for a Turing machine may be broken down into actual states and transitions, our vulnerabilities may be broken down into actual conditions and policy violations existing in the state space.

2.1. Defining Systems

We define security and policy in a context that more closely matches current vulnerability discussions today. Informally, we define a *machine* as is a set of hardware components that, when activated, runs a *system*. The system controls the *state space*, which dictates what actions may be performed. More precisely:

Definition 2.1.1. Machine. A set of hardware components operating together to manipulate and store information.

Essentially, the machine contains any hardware component required to realize a Universal Turing machine. The possible set of hardware components includes a motherboard and processor, hard drive, memory, graphics and sound cards, and other miscellaneous devices.

Each machine has an active *machine state*. This represents the finite sequence currently on the tape. More precisely:

Definition 2.1.2. Machine State. The sequence of all values stored in permanent or temporary memory resident on the machine.

Since we assume the tape is finite, the number of possible sequences on the tape is also finite. We define *state space* to include every possible machine state and the possible transitions between them. More precisely:

Definition 2.1.3. State Space. The tuple $\mathbb{S} = (\, M, T\,)$ where M is the set of all possible machine states for a machine and $T = \{\, (u, v) \mid u, v \in M \,\}$ is the set of transitions from machine state u to machine state v.

This is different from a state diagram, which illustrates the transition function of a Turing machine. The nodes in a state diagram represent computation states. Each edge is labeled with a transition that moves between states depending on the current symbol on the tape. Taking the transition rewrites the tape, and moves the tape head.

The components of state space are different. Each node represents a machine state, or possible tape sequence. Transitions between nodes in state space indicate that it is possible to change from one tape sequence to another. This is determined by examining possible configurations of the Turing machine based on its transition function.

Therefore, the state space provides the possible output of the machine. The *system* adds or removes transitions between nodes in state space.

Definition 2.1.4. System. The set of hardware and software components that can add or remove transitions in state space.

For example, the system might include hardware security controls (such as biometric devices), device drivers, operating systems, and/or applications. These components have the ability to enable or disable certain actions and abilities of the user. In relation to Turing machines, the system includes those harward and software components which implement the transition function.

Using these definitions, we may state a key assumption:

Assumption 2.1.5. Vulnerabilities exist in state space, but their causes are rooted in the system.

We want to identify the causes of vulnerabilities, and hence focus on aspects of the system. We use the term *condition* to describe properties of the system.

Definition 2.1.6. Condition. A property describing either the transitions or states of the system.

Conditions describe aspects of the system and the state space at the lowest possible level of abstraction. In this paper, we consider conditions to be primitives.

2.2. Defining Security

Most definitions of security make implicit assumptions about policy. We make these assumptions explicit, and define security in the context of a security policy.

Security policy defines what actions should or should not allowed for a user on a particular system, partitioning the state space into a set of allowed and disallowed machine states. More formally:

Definition 2.2.1. Security Policy. The pair $\mathbb{P} = (\, A, D\,)$ such that $A \subseteq M$ is the set of allowed states and $D = M - A$ is the set of disallowed states for a state space $\mathbb{S} = (\, M, T\,)$.

If a system is unable to transition to a disallowed state, we consider it to be *secure*. Otherwise, the system is *nonsecure*. More formally, a system is secure only when:

Definition 2.2.2. Secure. For any $a \in A$ and $d \in D$, there exists no transition $(\, a, d\,) \in T$ where $\mathbb{P} = (\, A, D\,)$ is the security policy for state space $\mathbb{S} = (\, M, T\,)$.

If we could define policy and security in this manner, determining if a system is secure would be trivial. We could simply search through the state space for any unwanted transitions and remove them. However, for nontrivial systems, state space is too large to enumerate and search. Not only is the size of state space exponential, but traversal through the possible transitions in state space may also be exponential. Thus we assume:

Assumption 2.2.3. Anything requiring enumeration of the state space is impractical.

Moreover, the policy enforced on a particular system may not capture the intent of the policy makers due to issues of practicality, configuration, and implementation. This archetypal notion of security as a partition of state space is not only impractical, but ineffective in practice.

2.3. Defining Policy

The distinction between *intended* and *implemented* policy is critical in understanding the notion of security. Consider the Linux access control mechanism, which controls object access based on whether the user is the owner of the object, belongs to the group which owns the object, or belongs to neither. Suppose we have two files, `file1` and `file2`, and the three users `xander`, `yasmin`, and `zaria`. The desired access control policy is to allow all three users access to `file1`, but allow only `xander` and `yasmin` to have access to `file2`. One solution is to create two groups and have the files owned by the respective groups. However, when the number of files and users increase, this mechanism quickly becomes impractical to manage. Hence, the implemented policy will reflect what is feasible given the system controls and may not reflect the intended policy. The notion of state space is unable to capture this difference, as the set of *allowed* states may includes states which are "allowed by *intention*" or "allowed by *implementation*."

We need to capture the differences between intended policy and implemented policy without relying on state space. For this, we adapt Carlson's *Unifying Policy Hierarchy Model* [10], which presents a hierarchical arrangement of four different types of policies. At the top of the hierarchy is the *oracle policy*, which accurately captures the intent of the policy makers. More formally:

Definition 2.3.1. Oracle Policy (OP). The function $\mathrm{OP} : S \times O \times A \rightarrow \{\mathtt{valid}, \mathtt{invalid}\}$ such that for any subject $s \in S$, object $o \in O$, and action $a \in A$:

$$\mathrm{OP}(s, o, a) = \begin{cases} \mathtt{valid} & \text{if action } a \text{ on object } o \text{ by } s \text{ should be allowed} \\ \mathtt{invalid} & \text{otherwise} \end{cases}$$

These elements of the sets S, O, or A 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:

$$\begin{aligned} \mathrm{OP}(\,\text{Xander}, \mathtt{xander}, \text{authenticate}\,) &= \mathtt{valid} \\ \mathrm{OP}(\,\text{Xander}, \mathtt{yasmin}, \text{authenticate}\,) &= \mathtt{invalid} \end{aligned}$$

Here, the oracle policy must have knowledge of the users Xander and Yasmin and not just the system-defined user accounts.

We make two assumptions regarding the oracle policy:

Assumption 2.3.2. The oracle policy provides a consistent policy decision for every possible subject, object, and action.

Assumption 2.3.3. The oracle policy always exists for every system.

Both of these claims depend on having access to the policy makers themselves. Whether the oracle policy is explicitly specified or intuitively referenced, the policy makers can always provide a policy decision. In this sense, the policy makers themselves represent the oracle policy.

Unfortunately, the oracle policy is often the expression of an ideal, and cannot be implemented precisely due to system, procedural, or environmental constraints. The *feasible oracle policy* takes these restrictions into account:

Definition 2.3.4. Feasible Oracle Policy (FP). The function $\mathrm{FP} : S_x \times O_x \times A_x \rightarrow \{\mathtt{valid}, \mathtt{invalid}\}$ such that for any subject $s \in S_x$, object $o \in O_x$, and action $a \in A_x$:

$$\mathrm{FP}(s, o, a) = \begin{cases} \mathtt{valid} & \text{if action } a \text{ on object } o \text{ by } s \text{ should be allowed} \\ \mathtt{invalid} & \text{otherwise} \end{cases}$$

where $S_x \subseteq S$, $O_x \subseteq O$, and $A_x \subseteq A$ are the finite sets of subjects, objects, and actions defined by some system x.

The feasible oracle policy represents the subset of the oracle policy which is possible and practical to implement. An *inherent policy violation* is a disagreement between the OP and FP:

Definition 2.3.5. Inherent Policy Violation. The tuple (s, o, a) where s is some subject, o is an object, and a is an action such that $\mathrm{OP}(s, o, a) \neq \mathrm{FP}(s, o, a)$.

If the subject, object, and action are not defined for FP, then the function is undefined. This type of inherent policy violation is often caused by limitations of policy representation and implementation. Returning to the earlier example, the system is only aware of user accounts and not the actual (human) users:

$$\begin{aligned} \mathrm{OP}(\,\text{Xander}, \mathtt{xander}, \text{authenticate}\,) &= \mathtt{valid} \\ \mathrm{FP}(\,\text{Xander}, \mathtt{xander}, \text{authenticate}\,) &= \mathtt{undefined} \end{aligned}$$

Sometimes, inherent policy violations are caused by environmental limitations. For example, an embedded system with limited performance capability and memory may be unable to provide the level of security required by the intended policy. In this situation, the feasible oracle policy captures the tradeoffs required to make the oracle policy achievable.

The *configured policy* represents the instantiation of the feasible oracle policy. Intuitively, it consists of the security policy defined by the configuration settings of the system:

Definition 2.3.6. Configured Policy (CP). The function $\mathrm{CP} : S_x \times O_x \times A_x \rightarrow \{\texttt{valid}, \texttt{invalid}\}$ such that for any subject $s \in S_x$, object $o \in O_x$, and action $a \in A_x$:

$$\mathrm{CP}(s,o,a) = \begin{cases} \texttt{valid} & \text{if action } a \text{ on object } o \text{ by subject } s \text{ is allowed} \\ \texttt{invalid} & \text{otherwise} \end{cases}$$

where $S_x \subseteq S$, $O_x \subseteq O$, and $A_x \subseteq A$ are the finite sets of subjects, objects, and actions defined by some system x.

Note that the FP provides what **should be** allowed or disallowed, whereas the CP provides what **is** allowed or disallowed (see figure 1). The differences between the FP and CP are caused by configuration errors, which may arise from human error or lack of understanding of the access control features. A *configuration policy violation* occurs when the configuration is inconsistent with the FP:

Definition 2.3.7. Configuration Policy Violation. The tuple (s,o,a) where s is some subject, o is an object, and a is an action such that $\mathrm{FP}(s,o,a) \neq \mathrm{CP}(s,o,a)$.

Finally, implemented policy controls may fail, or may be bypassed. An example of the latter is when a race condition allows a user to acquire unauthorized privileges. The access controls inhibit this delegation of privilege, but the race condition bypasses these inhibitors. The *actual policy* captures this distinction by representing what actions are actually possible in the system:

Definition 2.3.8. Actual Policy (AP). The function $\mathrm{AP} : S_x \times O_x \times A_x \rightarrow \{\texttt{valid}, \texttt{invalid}\}$ such that for any subject $s \in S_x$, object $o \in O_x$, and action $a \in A_x$:

$$\mathrm{AP}(s,o,a) = \begin{cases} \texttt{valid} & \text{if action } a \text{ on object } o \text{ by subject } s \text{ is possible} \\ \texttt{invalid} & \text{otherwise} \end{cases}$$

where $S_x \subseteq S$, $O_x \subseteq O$, and $A_x \subseteq A$ are the finite sets of subjects, objects, and actions defined by some system x.

Differences between the CP and AP indicate the policy configuration constraints were somehow subverted. This results in a *runtime policy violation*:

Definition 2.3.9. Runtime Policy Violation. The tuple (s,o,a) where s is some subject, o is an object, and a is an action such that $\mathrm{CP}(s,o,a) \neq \mathrm{AP}(s,o,a)$.

For example, suppose Xander is allowed access to a file `readme.txt`. However, Yasmin runs a denial of service attack that blocks Xander's access. The result is:

$$\begin{aligned} \mathrm{CP}(\,\texttt{xander}, \texttt{readme.txt}, \mathrm{read}\,) &= \texttt{valid} \\ \mathrm{AP}(\,\texttt{xander}, \texttt{readme.txt}, \mathrm{read}\,) &= \texttt{invalid} \end{aligned}$$

In general, a policy violation occurs whenever adjacent levels of the hierarchy disagree.

Definition 2.3.10. Policy Violation. The tuple (s,o,a) where s is some subject, o is an object, and a is an action such that an inherent, configuration, or runtime policy violation exists.

Figure 1 summarizes the different levels of the policy hierarchy. By defining policy using this hierarchy, we remove our dependency on state space and capture the differences between the notion of *intent* (OP), *feasibility* (FP), *configuration* (CP), and *possibility* (AP). However, we have said nothing on the implementation of these functions. In fact, we rely on the following assumption:

Assumption 2.3.11. These functions may be approximated by an administrator or field expert.

Since the configured policy is given by the system configuration, it may be automated. However, given the nature of the OP and FP, these functions may not be automated. Also, the functionality of the actual policy is too complex in practice to automate.

If we could fully automate the decisions made by the AP, we would be able to easily find vulnerabilities. We know this is not the case. Instead, we must depend on experts to analyze systems for vulnerabilities and approximate the actual policy.

For example, an expert can use a tool such as CQUAL to search for user/kernel pointer bugs in source code [20]. Each potential bug can be throughly analyzed to determine if there is a security risk (i.e. potential policy violation). Some user/kernel bugs may result in crashing the system, resulting in an actual policy that denies access. Other user/kernel bugs have the potential for privilege escalation, resulting in an actual policy that grants access otherwise denied in the configured policy.

2.4. Defining Vulnerabilities

Using our policy hierarchy, we can precisely define the notion of a vulnerability. A vulnerability exists whenever a policy violation exists. Consider our earlier example where a denial of service caused:

$$\begin{aligned} \mathrm{CP}(\,\texttt{xander}, \texttt{readme.txt}, \mathrm{read}\,) &= \texttt{valid} \\ \mathrm{AP}(\,\texttt{xander}, \texttt{readme.txt}, \mathrm{read}\,) &= \texttt{invalid} \end{aligned}$$

Now consider the conditions that allow such violations to exist. For this example, the conditions include whatever attributes of the system allowed Yasmin to launch a denial of service attack. In an earlier example of an inherent policy violation, the user Xander was able to authenticate to the user account `yasmin`. The conditions in this situation describe the inability of FP to determine the human user authenticated on the system user account. In fact, there must always be some bug, configuration error, environmental fault, or technology constraint that allows policy violations to occur. This leads to the following definition:

Definition 2.4.1. Vulnerability. The tuple $V = (N, P)$ where N is the nonempty set of conditions that enables the nonempty set of policy violations P for some system x.

We are less interested in issues of policy representation or configuration mistakes than in issues where the implemented policy fails. Therefore, we define a more specific type of vulnerability:

Definition 2.4.2. Runtime Vulnerability. The vulnerability $R = (N, P)$ where N is the nonempty set of conditions that enables the nonempty set of runtime policy violations P for some system x.

Now we may describe our example more precisely. The runtime vulnerability in this case would be $R_1 = (N_1, P_1)$ where the set N_1 describes the conditions on the system that allowed the denial of service to occur, and $P_1 = \{(\texttt{xander}, \texttt{readme.txt}, \mathrm{read})\}$ describes the resulting runtime policy violation.

Notice that a runtime vulnerability is directly tied to a specific system and policy. The set of conditions describe properties of a specific system. Therefore the runtime vulnerability may not apply to systems with different hardware or software. Even on systems with identical hardware and software, the set of users (and hence policy violations) may be different. For example, the same set of conditions N_1 may cause the policy violation $(\texttt{yasmin}, \texttt{readme.txt}, \mathrm{read})$ on a different system.

We do not want to define a separate vulnerability for every possible system and every possible policy on that system. This brings us to the core of our model: characteristics and symptoms.

2.5. Defining Characteristics

The above definitions depend on the *perfect knowledge assumption*:

Assumption 2.5.1. The Perfect Knowledge Assumption. We know the following:

1. The nonempty set of systems $X = \{x_1, x_2, \ldots, x_n\}$.
2. For every system $x_i \in X$:
   (a) The set of all possible conditions $\mathbb{N}_{x_i}$
   (b) The set of all possible runtime policy violations $\mathbb{V}_{x_i}$
3. For any runtime vulnerability $R_{x_i} = (N_{x_i}, V_{x_i})$:
   (a) The set of conditions $N_{x_i} \subseteq \mathbb{N}_{x_i}$
   (b) The set of runtime policy violations $V_{x_i} \subseteq \mathbb{V}_{x_i}$

In reality, the perfect knowledge assumption is impractical. We want to analyze vulnerabilities for a general, loosely defined set of systems. For example, we may be interested in analyzing vulnerabilities in Windows XP, no matter what hardware configuration the system may have. We have already mentioned that enumeration of the possible conditions and runtime policy violations is also impractical. Therefore, we address the impracticality of the perfect knowledge assumption by adding a new layer of abstraction. We call this new abstraction a *characteristic*:

Definition 2.5.2. Characteristic. The set of sets of conditions $C = \{N_1, N_2, \ldots N_n\}$ where each $N_i \in C$ defines the set of conditions represented by C on system i.

Each system has its own set of conditions which describe it. A characteristic is an abstraction of related sets of conditions across different systems. We say that characteristic C *imposes* the set of conditions N_i on system i. For example, suppose our characteristic is `[password based authentication]`. This will result in different conditions on a Microsoft Windows system versus a Linux system.

Level	Policy Type	Domain Restrictions	Decision Type
1	OP: Oracle Policy	None	Should Be Allowed
2	FP: Feasible Oracle Policy	System-Defined	Should Be Allowed
3	CP: Configured Policy	System-Defined	Is Allowed
4	AP: Actual Policy	System-Defined	Is Possible

Figure 1. Our adaptation of the Unifying Policy Hierarchy. Each level has either different domain restrictions on the possible subjects, objects, and actions or makes different types of policy decisions.

However, characteristics still require enumeration of conditions, something we want to avoid. The *system oracle* helps alleviate this:

Definition 2.5.3. System Oracle. The function $\mathcal{F} : \mathbb{C} \times \mathbb{X} \rightarrow \mathbb{N}$ where $C \in \mathbb{C}$ is a characteristic description, $x \in \mathbb{X}$ is a system, and $N \in \mathbb{N}$ is a set of conditions such that $\mathcal{F}(C, x) = N$ indicates that C imposes the set of conditions N on system X.

Instead of attempting to find the set of conditions associated with a characteristic, we can focus on providing a description of that characteristic. The system oracle takes this description, and provides the associated set of conditions for a particular system. We can simplify the process even further with the *vulnerability indicator function*:

Definition 2.5.4. Vulnerability Indicator Function. The function $\mathcal{I} : \mathbb{R} \times \mathbb{C} \times \mathbb{X} \rightarrow \{\texttt{valid}, \texttt{invalid}\}$ where $R = (N, P) \in \mathbb{R}$ is a runtime vulnerability, $C \in \mathbb{C}$ is a characteristic, and $x \in \mathbb{X}$ is a system such that:

$$\mathcal{I}(R, C, x) = \begin{cases} \texttt{valid} & \text{if } \mathcal{F}(C, x) \subseteq N \\ \texttt{invalid} & \text{otherwise} \end{cases}$$

The vulnerability indicator function provides a yes or no answer to whether a given characteristic describes some subset of conditions for a given vulnerability on a specific system. Using the vulnerability indicator function, we never have to enumerate conditions of a system or of a characteristic.

The implementation of the system oracle and vulnerability indicator function still depend on conditions. It is here we bring in the expertise of those working in the field:

Assumption 2.5.5. The system oracle and vulnerability indicator function may be approximated by an expert in the field.

Given a well-defined characteristic, an expert should be able to determine if that characteristic applies to a particular system. In fact, we argue that this is precisely what occurs currently when analyzing vulnerabilities. Given a description of a vulnerability and its "properties," security experts decide if that vulnerability applies to particular systems and take appropriate action. By restating this phenomena in a precise model, we can see where ambiguity and judgement calls are introduced into the process of vulnerability analysis.

For example, consider the `login` vulnerability in UNIX Version 6. It was possible to overflow the input buffer for the password, overwriting the retrieved password hash used for validation. This allowed an adversary to input his/her own password and hash combination in the password buffer, overwriting the valid hash and gaining access to the system. We describe this data buffer overflow with four characteristics:

BUF1: The length of an input string may be longer than the destination buffer.

BUF10: The input string may contain data of the same type as the variable `var`.

BUF11: Input may modify the data of `var` without being countered.

BUF12: The variable `var` determines which execution path is taken at a future point in the execution of the process.

An expert with the UNIX operating system could examine the source code for the corresponding characteristics. The characteristic BUF1 captures the overflowing of the password input buffer. This is represented in the code by the lack of bounds checking for the length of the user-input password. Characteristics BUF10 and BUF11 capture the overwriting of the password hash. In the code, the program retrieves the password hash for the given account and stores it adjancent to the buffer for the user-input password. This allows the overflow of the user-input password to overwrite the hash. Finally, the login program uses this hash and compares it to the hash of the user-input password. Since the hash has been overwritten, the user is authenticated. This is functionality is captured by characteristic BUF12.

Notice the shift in focus from, for example, the Open Vulnerability and Assessment Language (OVAL) [4]. The OVAL Language focuses on determining the presence of vulnerabilities based on the configuration of a given system, including criteria such as the operating system version, presence of patches, installed applications, and so on. Our focus is on providing characteristics which describe aspects of the code, independent of version numbers.

Our approach is not unrelated however. Our characteristics may be used to determine which versions of an operating system are vulnerable. For example, the characteristics for the `login` vulnerability only existed in UNIX Version 6. Once that is determined, an OVAL Definition could be created for use by system administrators.

2.6. Defining Symptoms

We use *symptoms* to avoid having to enumerate all possible runtime policy violations.

Definition 2.6.1. Symptoms. An abstraction of runtime policy violations.

For example, let the symptom `[denial of service]` represent all policy violations (s, o, a) such that $\mathrm{CP}(s, o, a) = \texttt{valid}$ but $\mathrm{AP}(s, o, a) = \texttt{invalid}$. This captures the case when subject s has action a on object o blocked. The symptom `[privilege escalation]` could be defined similarly to represent all policy violations (s, o, a) such that $\mathrm{CP}(s, o, a) = \texttt{invalid}$ but $\mathrm{AP}(s, o, a) = \texttt{valid}$. This captures the case when subject s gains the ability to perform action a on object o.

We can also create symptoms which depend on the number or severity of violations that occur. For example, a system-wide denial of service could be defined for the case when every privilege which is valid in the CP becomes invalid in the AP. Likewise, escalation of privileges to root could be represented by the case where for a single subject, every invalid privilege in CP becomes valid in the AP.

For every discussion on characteristics, there is a similar discussion for symptoms. Most of our focus in this paper on characteristics, but symptoms play an essential part in our model.

2.7. Defining Abstractions

We are close to describing vulnerabilities in terms of characteristics and symptoms. However, we also need a meaningful way to compare vulnerability abstractions. The *universal characteristic set* provides the basis for such a comparison:

Definition 2.7.1. Universal Characteristic Set. The fixed, nonempty set $\mathbb{U}$ of characteristics under consideration.

The universal characteristic set does not necessarily include every possible characteristic. It is the set of characteristics tailored for a specific situation, and may grow or shrink as interests change. However, when this happens, the *maximal characteristic set* for a vulnerability must also change:

Definition 2.7.2. Maximal Characteristic Set. The set of characteristics $M_{\mathrm{c}} \subseteq \mathbb{U}$ for a runtime vulnerability $R = (N, P)$ on some system x such that for every characteristic $C \in \mathbb{U}$: if $\mathcal{I}(R, C, x) = \texttt{valid}$, then $C \in M$.

This set includes all characteristics from the universal set which apply to a specific runtime vulnerability. Therefore a runtime vulnerability abstraction is given by:

Definition 2.7.3. Runtime Vulnerability Abstraction. The tuple $A = (M_{\mathrm{c}}, M_{\mathrm{s}})$ where M_{c} is the maximal characteristic set and M_{s} is the maximal symptom set for some runtime vulnerability R.

Whenever the vulnerability abstractions are the same for any two runtime vulnerabilities under the same universal sets, we consider those vulnerabilities *equivalent*:

Definition 2.7.4. Runtime Vulnerability Equivalence Class. The set of runtime vulnerabilities $R = \{R_1, \ldots, R_n\}$ under a universal set $\mathbb{U}$ such that $A = (M_{\mathrm{c}}, M_{\mathrm{s}})$ is the runtime vulnerability abstraction for each pair $R_i, R_j \in R$.

For example, consider a simple buffer overflow which exists in multiple versions of a web server. The conditions and policy violations may change depending on the system in question, causing multiple runtime vulnerabilities. However, if every one of these runtime vulnerabilities have the same abstraction, then we say they belong to the same equivalence class. This allows us to treat runtime vulnerabilities across different systems as a single, more abstract, vulnerability.

With this we are finally able to describe vulnerabilities without requiring detailed knowledge of state space, conditions, or policy configurations. Most notions of a vulnerability used today are equivalent to our notion of a runtime vulnerability equivalence class. This is important because it preserves the established intuition of other models. In addition, the model presented here allows us to identify where and how ambiguity may be introduced into our vulnerability descriptions. This is key to being able to mitigate the negative impacts caused by ambiguity.

3. Tree Approach to Classification

Classification schemes have been used since the mid 1970s to better understand and analyze vulnerabilities. However, many of these classification schemes do not classify vulnerabilities. Instead, they focus only on errors, faults, flaws, or bugs that *tend* to lead to a vulnerability.

For example, many buffer overflow bugs lead to a vulnerability. Such vulnerabilities are then classified as *buffer overflow vulnerabilities*. However, not all buffer overflow bugs lead to vulnerabilities and some bugs lead to much more severe vulnerabilities than others. There must be some other characteristic missing from these classification schemes which allow a buffer overflow bug to become a serious security issue.

It is for this reason we propose a different approach: using our abstraction of a vulnerability into maximal characteristic and symptom sets as a foundation for classification. Using characteristics, we can capture more than just the originating software bug. Using symptoms, we can describe the difference between a vulnerability which leaks information versus one that leads to a privilege escalation.

Our classification framework consists of four main elements: the universal characteristic/symptom sets, the master classification tree, the maximal characteristic/symptom set for a runtime vulnerability abstraction, and finally the vulnerability classification tree.

3.1. Classification Components

We have already introduced the concepts of the universal characteristic set. The *master characteristic tree* builds on this set:

Definition 3.1.1. Master Characteristic Tree. A rooted tree $\mathbb{C}$ where each leaf node represents a characteristic and each parent node represents a class of characteristics.

There is a similar master symptom tree. Combined, these trees give us the *master classification tree*:

Definition 3.1.2. Master Classification Tree (MCT). A rooted tree $\mathbb{T} = \mathbb{C} \cup \mathbb{S}$ where $\mathbb{C}$ is the master characteristic subtree and $\mathbb{S}$ is the master symptom subtree.

The MCT provides all of the possible classes in our classification scheme. The *vulnerability classification tree* (VCT) provides the classification for any given vulnerability. The simplest way to define the VCT is as follows:

Definition 3.1.3. Vulnerability Classification Tree (VCT). A subgraph $\mathbb{V} \subseteq \mathbb{T}$ of the master classification tree for a runtime vulnerability abstraction $A = (M_c, M_s)$ such that if the characteristic node $c \in \mathbb{T}$ and $c \in M_c$ then $c \in \mathbb{V}$ along with its ancestors and if the symptom node $s \in \mathbb{T}$ and $s \in M_s$ then $s \in \mathbb{V}$ along with its ancestors.

Any characteristic that is both in the MCT and the maximal characteristic set is added to the VCT along with all of its ancestors. The same occurs for symptoms. The inner nodes of the VCT give the classes which describe the vulnerability. See figure 2 for an example of what the MCT and VCT look like.

This allows a vulnerability to belong to multiple classes at once. There will be classes which describe the characteristics of the vulnerability and classes which describe its possible symptoms. Some classes will even be abstractions of others, with the most abstract class being `runtime vulnerability`.

The process of determining if a node from the MCT is included in the VCT may be made more sophisticated. Our results section gives an example of this.

3.2. Classification Framework

We present a *framework* for classification. This provides how to classify a vulnerability based on its abstraction. However, to use this framework, a universal characteristic and symptom set must first be defined. Once we have these universal sets, we can begin building the master classification tree. This tree adds a hierarchy of abstraction on top of the universal sets.

At this point we can begin to classify vulnerabilities. We compare the maximal characteristic and symptom set for the vulnerability abstraction with our master classification tree. The nodes of the vulnerability classification tree provide the vulnerability's classification.

Given the perfect knowledge assumption, our classification is primitive and repeatable. Under this assumption, the system oracle and vulnerability indicator function perfectly determine the maximal sets for our vulnerability abstraction. From there, classification is automatic based on our master classification tree.

However, we cannot make the perfect knowledge assumption in practice. Even creating "meaningful" characteristics is a challenging obstacle. For example, `[machine has power]` is a valid characteristic, but in most cases it adds little to the vulnerability's classification.

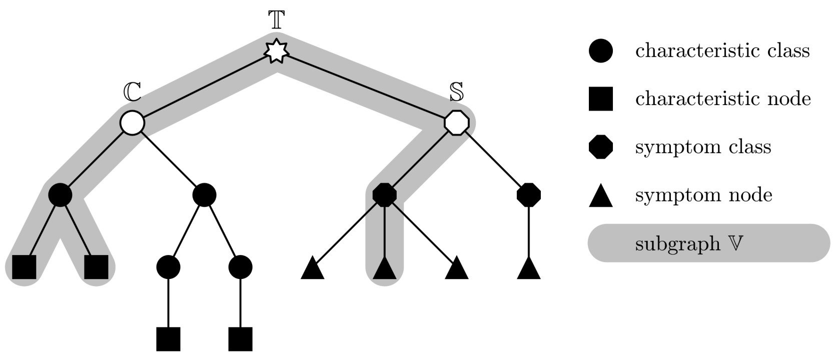

Figure 2. Representation of the master classification tree 𝕋 and vulnerability classification tree 𝕍.

Despite these obstacles, we can still strive for repeatability by making sure our characteristics are well-defined, making our approximation of the system oracle consistent. Therefore characteristic descriptions must include how to determine if the characteristic is applicable, and what point of view is being taken.

Classification is layered, and needs to be only as detailed as necessary. The framework is also extensible. As technologies improve we may add new characteristics to our universal set, and new sub-trees to our master classification tree. We may even leave certain classes undefined, to be filled in with specific characteristics at a later date. This flexibility allows the classification scheme to be built iteratively.

4. Classification Results

We have applied our model and framework to two areas: buffer overflow vulnerabilities and network protocol vulnerabilities [9, 31]. We present a summary our buffer overflow results here.

4.1. Buffer Overflow Classification

We were able to identify 13 characteristics and four different types of buffer overflow vulnerabilities. The core characteristic to any buffer overflow vulnerability is **BUF1**:

BUF1: The length of an input string may be longer than the destination buffer.

The characteristic BUF1 captures all buffer overflow bugs and exists in all buffer overflow vulnerabilities. However, a vulnerability requires more than BUF1 to violate policy. Consider the class of *executable buffer overflow* vulnerabilities. The idea is to overflow a buffer, often allocated on a stack, such that the return address or function pointer is altered to redirect the program flow to malicious code stored in the input payload. For example, the `fingerd` vulnerability exploited by the 1988 Internet Worm was an executable buffer overflow [1]. Successful exploitation of these vulnerabilities also require the following characteristics to exist:

BUF2: The input string many contain addresses.

BUF3: The input string may contain instructions.

BUF4: Input may modify the stored return address without being countered.

BUF5: The process may jump to memory on the stack.

BUF6: The process may execute instructions stored in the stack.

However, some executable buffer overflow vulnerabilities involve memory on a heap. Therefore we need to define additional characteristics:

BUF7: Input may modify the function pointer variable without being countered.

BUF8: The program may jump to the heap.

BUF9: The program may execute instructions stored in the heap.

Some vulnerabilities, like the `login` vulnerability described earlier, are *data buffer overflow* vulnerabilities. This introduces the following characteristics:

BUF10: The input string may contain data of the same type as the variable `var`.

BUF11: Input may modify the data of `var` without being countered.

BUF12: The variable `var` determines the future execution path taken by the process.

Finally, some data buffer overflows are indirect – changing the value of a pointer instead. The value of the variable referenced by the pointer then alters program flow. To capture this we need to add the following characteristics to our set:

BUF13: Input may modify the address stored in a variable `ptr` without being countered.

BUF14: The value of the variable pointed to by `ptr` determines the future execution path taken by the process.

The characteristics BUF1 through BUF14 make up our universal characteristic set. We use our universal characteristic set to create our master characteristic tree, as show in figure 3.

When we classify vulnerabilities using this tree, we use a more sophisticated version of the master classification tree. This introduces the concepts of a simple-class versus a super-class. A *simple-class* is a node in our tree which contains only characteristics. A *super-class* is a node which contains other simple-class or super-class nodes. For example, "direct executable" is a simple-class but "executable buffer overflow" is a super-class.

This distinction is important for determining our vulnerability classification tree. A simple-class is only included if all the contained characteristics are in the maximal characteristic set. Specifically, the simple-class C is only included in the VCT if $C \subset M_{\mathrm{c}}$ for a vulnerability abstraction $A = (M_{\mathrm{c}}, M_{\mathrm{s}})$. Then, any ancestor of the simple-class is added to the VCT. For the `login` data buffer overflow, the VCT would contain classes "buffer overflow vulnerability," "data buffer overflow," and "direct data."

We chose this approach because the characteristics for each simple-class are both necessary and sufficient for the buffer overflow to be exploitable [9]. For example, consider the `panic()` overflow in the Linux kernel [3]. In the implementation of `panic()` there is an unbounded `vsprintf()` call resulting in a buffer overflow on the stack. This overflow involves characteristics BUF1 through BUF3. However, the `panic()` function never returns making it unclear whether this overflow can result in a change of program flow. Since this overflow does not include all the characteristics of a direct executable buffer overflow, it is not considered a buffer overflow vulnerability.

This allows us to differentiate between bugs and vulnerabilities. It is always good practice to fix all known bugs. However, it may be unnecessary to immediately push out a major kernel revision over a bug versus a dangerous vulnerability.

We are also able to see which characteristics are common across different types of buffer overflows. Mant vulnerabilities require the input to contain intructions. This characteristic could be disabled by architectures which use segmentation to separate instructions and data.

Since these characteristics are designed to be necessary and sufficient, they also provide possible vectors for defense and prevention of buffer overflow vulnerabilities. This allows us to also classify the defenses against these vulnerabilities. BUF1 defenses include range-checking, bounds-checking, and hardware segmentation [29, 23, 25, 27]. Products like StackGuard and PointGuard are BUF4 and BUF7 defenses [13, 12].

This approach is compatible with other defense classifications. For example, the class of defenses characterized as "writing correct code" in Cowan's work maybe restated as BUF1 defenses [14].

5. Related Work

The definition of a vulnerability used by CVE is similar to our own in that it depends on policy, but separates the terms *universal vulnerability* from that of an *exposure* with a vague line separating which policies are more "universal" than others [2].

Other policy frameworks include those by Dobson and McDermind, Schneider, and Sterne [15, 26, 28]. However, we chose to adapt the Unifying Policy Hierarchy model due to its similar layered approach and formalism [10].

There are several attack models which are closely related to our work. For example, the work by Alves-Foss defines a "System Vulnerability Index" that combines a number of factors to determine how vulnerable a system is to attack. The work by Howard et al provides the concept of an "attack surface" and uses a three-dimensional model to study the vulnerability of a system. However, our focus is on modeling and classifying vulnerabilities, and less on measuring the vulnerability of a system. Fithen et al provide a graph-based methodology to analyze specific preconditions and postconditions, very similar to our approach [17]. Our model differs in that it focuses on multiple layers of abstraction while utilizing the Universal Turing machine model to allow vulnerabilities to be reduced to conditions and policy violations, or abstracted to characteristics and symptoms.

Many of the early classification schemes focus on bugs, faults, flaws, or errors which commonly lead to vulnerabilities. This includes the RISOS [5], PA [19], Landwehr [22] schemes as well as others [16, 24]. The Taxonomy of Security Faults by Aslam is a modern example of bug classification [6, 7]. Our work differs from these schemes in that we characterize more than just the underlying software bug. We try to also capture those conditions which turn a bug into a vulnerability, and the policy violations that result.

Other schemes, such as that by Cohen, Howard, and Weber focus less on system vulnerabilities and more on attacks, computer incidents, and intrusions [11, 18, 30]. We limit

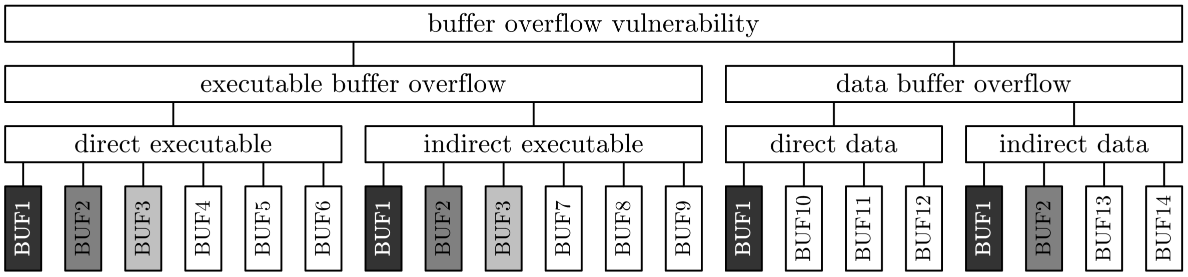

Figure 3. Example master characteristic tree for buffer overflow vulnerabilities.

our model and classification framework to runtime vulnerabilities, and do not currently attempt to characterize network attacks.

Krsul presents a formal vulnerability classification scheme based on the assumptions made by programmers [21]. His work also attempts to classify vulnerabilities, not errors. However, his approach and classification characteristics vary widely from our approach.

Our use of characteristics and goals for classification come directly from "Vulnerability Analysis: An Extended Abstract" by Bishop [8]. However, we attempt to further develop and formalize the notions provided in Bishop's previous work by providing not just a classification scheme, but also a precise model for vulnerability analysis in general.

6. Conclusion

Vulnerabilities will likely always exist, if for no other reason than fallible humans will misconfigure or make errors in the design, implementation, deployment, or maintenance of systems. This model and classification framework is an effort to increase our understanding of vulnerabilities, and how to defend against them.

We base our model on the components of a Universal Turing machine – and show how security may be defined in terms of states and transitions. However, the state diagram for a Universal Turing machine is often too complex to specify. Instead, we describe Turing machines in terms of high-level algorithms.

We use a similar approach to mitigate the complexity of state space. To do this, our model must first distinguish between vulnerabilities arising from imprecision introduced by the system model (inherent policy violations), errors in configuration (configuration policy violations), and errors in implementation (runtime policy violations) by using the Unifying Policy Hierarchy [10].

We then introduce characteristics and symptoms to describe runtime vulnerabilities. This allows us to describe vulnerabilities in a system- and policy-independent way.

Finally, we provide an extensible vulnerability classification framework and show its application to buffer overflow vulnerabilities. We discuss several of the necessary and sufficient characteristics for different types of buffer overflow vulnerabilities, and how their classification also provides us potential defense vectors.

As a result of our work, we better understand what differentiates bugs from vulnerabilities, where vulnerabilities arise in the system, where and how vulnerability analysis introduces ambiguity, and have a method of classifying vulnerabilities which also provide potential defenses against them.

References

[1] BSD fingerd buffer overflow vulnerability. Online at http://www.securityfocus.com/bid/2/info.

[2] Common vulnerabilities and exposures (CVE). Online at http://cve.mitre.org/about/terminology.html.

[3] Linux kernel panic() overflow. Online at http://osvdb.org/displayvuln.php?osvdb_id=7423.

[4] Open vulnerability and assessment language (OVAL). Online at http://oval.mitre.org/index.html.

[5] R. P. Abbott, J. S. Chin, J. E. Donnelley, W. L. Konigsford, S. Tokubo, and D. A. Webb. Security analysis and enhancements of computer operating systems. Technical Report NBSIR 76-1041, Institute for Computer Sciences and Technology at the National Bureau of Standards, April 1976.

[6] T. Aslam. A taxonomy of security faults in the unix operating system. Master's thesis, Purdue University, 1995. Available online at ftp://ftp.cerias.purdue.edu/pub/papers/taimur-aslam/aslam-taxonomy-msthesis.ps.Z.

[7] T. Aslam, I. Krsul, and E. H. Spafford. A taxonomy of security faults. In *Proceedings of the National Computer Security Conference*, 1996. Available online at ftp://ftp.cerias.purdue.edu/pub/papers/taimur-aslam/aslam-krsul-spaf-taxonomy.ps.

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

[9] M. Bishop, D. Howard, S. Engle, and S. Whalen. A taxonomy of buffer overlow vulnerabilities, 2006. Submitted and undergoing revision.

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

[11] F. Cohen. Information system attacks: A preliminary classification scheme. *Computers and Security*, 16(1):29–46, 1997. Available online at http://dx.doi.org/10.1016/S0167-4048(97)85785-9.

[12] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. Pointguard: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium*, pages 91–104, August 2003.

[13] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: Automatic adaptive detection and prevention of buffer overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, pages 63–77, January 1998.

[14] C. Cowan, P. Wagle, S. Beattie, and J. Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference and Expo (DISCEX)*, January 2000.

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

[16] A. Endres. An analysis of errors and their causes in system programs. *ACM SIGPLAN Notices*, 10(6):327–336, 1975. Available online at http://portal.acm.org/citation.cfm?id=390016.808455.

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

[18] J. D. Howard. *An Analysis of Security Incidents on the Internet 1989 - 1995*. PhD thesis, Carnegie Mellon University, 1998. Available online at http://www.cert.org/research/JHThesis/Start.html.

[19] R. B. II and D. Hollingworth. Protection analysis: Final report. Technical Report ISI/SR-78-13, Information Sciences Institute at the University of Southern California, May 1978. Available online at http://csrc.nist.gov/publications/history/bisb78.pdf.

[20] R. Johnson and D. Wagner. Finding user/kernel pointer bugs with type inference. In *Proceedings of the 13th USENIX Security Symposium*, pages 119–134, August 2004. Available online at http://www.cs.umd.edu/∼jfoster/cqual/.

[21] I. Krsul. *Software Vulnerability Analysis*. PhD thesis, Purdue University, 1998. Available online at ftp://ftp.cerias.purdue.edu/pub/papers/ivan-krsul/krsul-phd-thesis.pdf.

[22] C. E. Landwehr, A. R. Bull, J. P. McDermott, and W. S. Choi. A taxonomy of computer program security flaws. *ACM Computing Surveys*, 26(3):211–254, September 1994. Available online at http://doi.acm.org/10.1145/185403.185412.

[23] K. Lee and S. Chapin. Type-assisted dynamic buffer overflow detection. In *Proceedings of the 11th USENIX Security Symposium*, pages 81–88, 2000.

[24] T. J. Ostrand and E. J. Weyuker. Collecting and categorizing software error data in an industrial environment. *Journal of Systems and Software*, 4(4):289–300, November 1984. Available online at http://dx.doi.org/10.1016/0164-1212(84)90028-1.

[25] O. Ruwase and M. Lam. A practical dynamic buffer overflow detector. In *Proceedings of the 2004 Symposium on Network and Distributed System Security (NDSS)*, pages 159–169, February 2004.

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

[27] Z. Shao, Q. Zhuge, Y. He, and E. Sha. Defending embedded systems against buffer overflow via hardware/software. In *Proceedings of the 19th Annual Computer Security Applications Conference*, page 352, December 2003.

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

[29] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of the Network and Distributed System Security (NDSS) Symposium*, February 2000.

[30] D. Weber. A taxonomy of computer intrusions. Master's thesis, Massachusetts Institute of Technology, 1998. Available online at http://hdl.handle.net/1721.1/9861.

[31] S. Whalen, S. Engle, and M. Bishop. Protocol vulnerability analysis. Technical Report 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.