Tree Approach to Vulnerability Classification

Sophie Engle, Sean Whalen, Damien Howard, and Matt Bishop
Department of Computer Science
University of California, Davis
[sjengle,shwhalen,djhoward,mabishop]@ucdavis.edu

Abstract

We present a classification scheme based on conditions which must hold for a vulnerability to exist. This scheme allows for vulnerabilities to fall into multiple classes without ambiguity, and enables analysts to focus on the causes of vulnerabilities. We use a tree-based approach to organize these conditions at different levels of abstraction.

1 Introduction

Several existing vulnerability classification schemes fulfill various needs of researchers, developers, and system administrators. However, current schemes have not made material progress in integrating policy or identifying unknown vulnerabilities. Our goal is to provide an unambiguous classification scheme to further progress in these areas.

A vulnerability classification scheme should satisfy several properties [7]. To illustrate these properties, consider a subset of playing cards (see figure 1). Let each card represent a vulnerability with the value, suit, color, and type of card representing conditions for the vulnerability to exist. Therefore the conditions represented by the four of spades (**4♠**) include having the value of four, belonging to the spade suit, being black in color, and being a number card (versus a face card).

Any classification scheme should allow vulnerabilities that arise from similar conditions to belong in the same class. For example, the four of spades, clubs, hearts, and diamonds (**4♠**, **4♣**, **4♥**, **4♦**) should all belong to the same class since each card has a value of four.

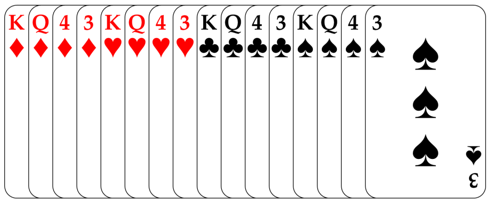

Figure 1: Subset of playing cards containing the king, queen, four, and three cards from all four suits.

However, vulnerabilities *may* fall into multiple classes. For example, the four of spades (**4♠**) also belongs to the class of spade cards with the king, queen, and three of spades (**K♠**, **Q♠**, **3♠**). Based on color, the four of spades also belongs to the class of black cards along with any spade or club card. Finally, it can also be viewed as belonging to the class of numbered cards, unlike the king or queen cards. The four of spades (**4♠**) belongs to all of these classes simultaneously. Figure 2 shows different classes based on the properties of value, suit, color, and type.

Classification should be primitive: determining if a vulnerability belongs to a particular class requires only a *yes* or *no* answer without ambiguity. As a result, each class is described by exactly one property. Consider the class of heart cards. To belong a card must have a heart (♥) suit. Other properties, such as value or type, do not matter. For example, it is easily determined that the four of spades (**4♠**) does not belong to the heart class while the three of hearts (**3♥**) does.

Finally, classification should be well-defined and based on measurable details. In the above examples, classification of cards is based on value, suit, color, or type. Classification should not be based on qualitative characteristics such as *high*, *low*, *lucky* or *unlucky*. Likewise, classification of vulnerabilities should not be based on social characteristics or motives.

2 Terminology

Many security and vulnerability terms have ambiguous or conflicting definitions, leading to confusion or misunderstanding. To avoid this, we explicitly define the terms used in this paper.

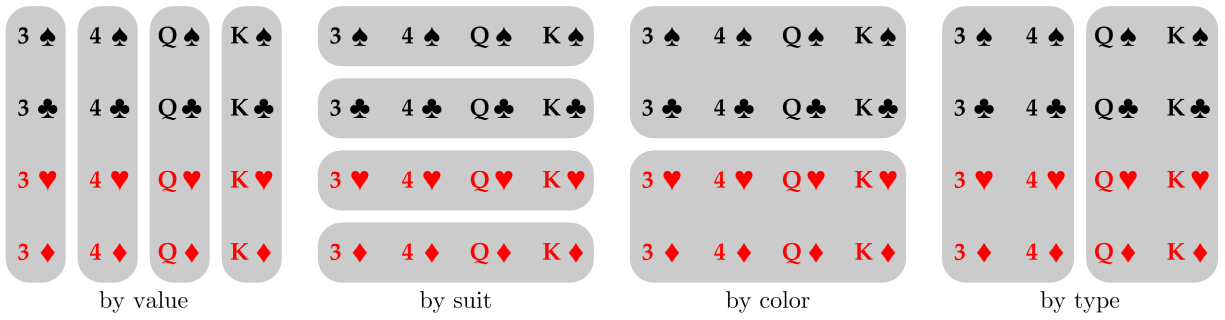

Figure 2: Different classes based on value, suit, color, and type.

A **security policy** is a partition of system states into allowed and disallowed states (see figure 3). A **vulnerability** is a set of transitions which take a system from an allowed state to a disallowed state, crossing the policy partition. The set of commands that implement a vulnerability is considered an **exploit**, while an **attack** is the *execution* of an exploit.

Each vulnerability defines exactly one vulnerable state. An allowed state from which a disallowed state may be reached is considered a **vulnerable state** (see figure 3). For example, if *user authenticated* is considered an allowed state but still permits policy to be breached, then this state is a vulnerable state. Once the transition from this state to a disallowed state is blocked, this state is no longer considered vulnerable. Attributes of vulnerable states are called **characteristics**, which are discussed next.

2.1 Characteristics

An attribute of a vulnerable state is considered a **characteristic**. For example, let the four of hearts (**4♠**) represent a vulnerable state. The value, suit, color, and type are all characteristics of the state. The **characteristic set** is the set of all attributes describing the vulnerable state.

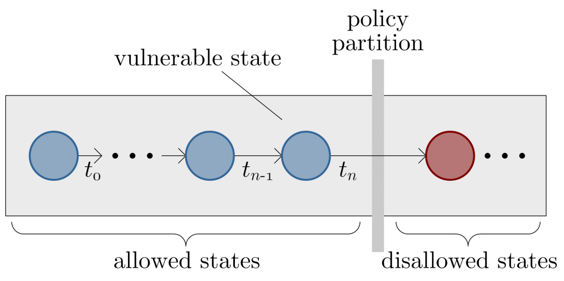

Figure 3: Representation of a vulnerable system. The vulnerability is the set of transitions t_0 to t_n.

We hypothesize that a large set of vulnerabilities can be described by a smaller set of characteristics. If true, detection and remediation of vulnerabilities based on the characteristic set may be a practical method of improving system security.

The characteristic set is **minimal** when it contains the minimum number of characteristics necessary for the vulnerability to exist. The characteristic set is **sound** when the characteristics are independent. The **basic characteristic set** is both sound and minimal.

We hypothesize that the basic characteristic set can be determined for any vulnerability. Removal of any characteristic from the basic set breaks the connectivity between the vulnerable and disallowed state, disabling the vulnerability. For example, consider a system with allowed states $\{$**K♣**, **Q♣**, **4♠**$\}$ and disallowed state $\{$**4♥**$\}$. Any transition between two cards which share an attribute is legal (similar to the game *crazy eights*). Thus **4♠** is a vulnerable state (see figure 4a) with a basic characteristic set $\{value = 4\}$. The set does not include suit as it does not effect the transition to **4♥**. Changing the value to 3 removes this vulnerability from the system (see figure 4b).

A set of characteristics C is considered **complete** with respect to a given system if the characteristic sets of all vulnerabilities in the system are composed of elements of C. Finding the complete characteristic set for a deck of cards is trivial (needing only value and suit to describe all cards), however this is rarely the case with computer systems.

We use the basic characteristic set of a vulnerability to classify that vulnerability. The characteristics themselves must be well-defined, and should be based on code, the environment, or other technical details.

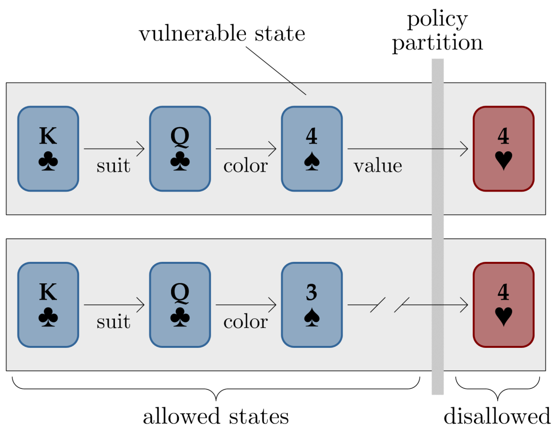

Figure 4: **a.** vulnerable system (top), **b.** secure system

In this paper, we focus on the classification scheme and methodology, and merely touch on specific characteristics themselves.

2.2 Characteristic Trees

Unfortunately, enumerating the state machine representation of a computer system is intractable. We need another way of representing vulnerabilities and characteristics. Therefore we use a tree to organize the characteristics of a system. This **characteristic tree** provides a hierarchical organization of the characteristic set, allowing for greater flexibility and depth than vulnerability classification based on characteristics alone.

Different layers in the tree represent characteristics at differing levels of abstraction. The characteristics themselves can be given at any level of abstraction, including low levels such as software or hardware implementation as well as at higher, design levels of abstraction. For extremely simple systems, the characteristics may be capable of describing system state attributes directly. For complex systems the characteristics must be more general.

Each node in the characteristic tree has associated type and subtype indicating its allowed interaction with other nodes. There are two main types: **abstract** and **detail**. For a deck of cards, abstract nodes may include *value* or *suit*, while *king* (**K**) or *spade* (♠) are more specific and would be considered detail nodes.

Abstract nodes provide hierarchical organization of the characteristics. There are two abstract subtypes, **container** and **category** nodes. Abstract container nodes are the most abstract and high-level nodes in the tree. These nodes may only have other abstract nodes as children. For example, *design* could be an abstract container node for characteristics describing design flaws. Likewise there could be an *implementation* node for implementation flaws.

On the other hand, abstract category nodes may contain only characteristic nodes (defined below) as children. These nodes provide the interface between abstract and detail nodes.

There are two subtypes of detail nodes, **characteristic** and **implementation** nodes. The characteristic node represents an actual characteristic of the vulnerability. The set of all characteristic nodes in the tree represent the characteristic set of the vulnerability. These nodes may contain implementation level details in the form of implementation nodes. Implementation nodes provide details for the parent characteristic only, and may not always be included.

For example, if the characteristic node is "*failure to check bounds*," the corresponding implementation node would contain information such as "*bounds of* `pass` *not checked in* `login` *program*." The type and subtype relationships are summarized in figure 5.

Two other restrictions are placed on the characteristic tree. First, abstract nodes must have at least one child node or the abstract node is removed from the tree. Hence abstract nodes are always internal nodes, and only detail nodes (either characteristic or implementation nodes) may be leaf nodes of the tree. This also means that only characteristics may have implementation-level details.

Finally, characteristic trees are always rooted. The root node of the tree will always be an abstract container node (called the root container). This node represents the vulnerability as a whole.

Figure 6 shows the relationships between different node types. Notice that because of these relationships, any path from the root to any leaf in the tree will include exactly one root container, one abstract category node, and one detail characteristic node.

type	subtype	child type
abstract	container	abstract (any)
abstract	category	characteristic
detail	characteristic	implementation
detail	implementation	(none)

Figure 5: Characteristic tree node types.

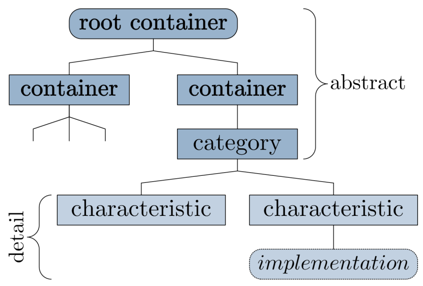

Figure 6: Relationships between node types.

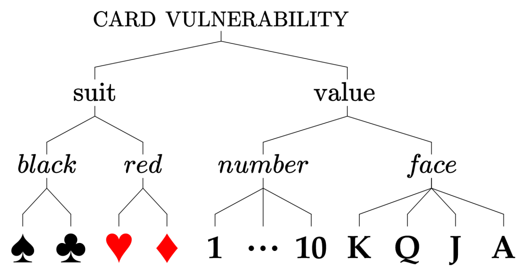

Figure 7: Complete characteristic tree for a deck of cards.

The **complete characteristic tree** hierarchically organizes the complete characteristic set for a given system. This tree provides an overall picture of the types of characteristics vulnerable states tend to have. Implementation nodes are never included in the complete characteristic tree, as they are specific to a particular vulnerability. Figure 7 provides an example complete characteristic tree for card characteristics.

Every vulnerability has a characteristic set *and* a corresponding characteristic tree. It is important to note that individual characteristic trees are not decision trees and do not represent a taxonomy of vulnerabilities. However, a taxonomy of vulnerabilities could be derived from a complete characteristic tree.

2.3 Symptoms

Recall that characteristics describe vulnerable states, which are a subset of allowed states (see figure 3). **Symptoms** are similar to characteristics, except they describe *disallowed* states reached by the vulnerability. Specifically, symptoms are those properties of the disallowed state which cause it to lay outside the policy partition. Symptoms describe the effect of exploiting the vulnerability on the system.

Since both characteristics and symptoms are attributes of system states, they are both described and handled in similar manners. For example, there are symptom sets and symptom trees with identical properties as characteristic sets and characteristic trees.

While the focus of this paper is classification based on characteristics, classification based on symptoms is also possible. In fact, symptoms may provide additional insight into the nature of vulnerabilities. The complete symptom tree may even help provide a taxonomy of vulnerability severity.

For example, not all buffer overflow vulnerabilities have severe consequences. Some overflows may crash the system, while others may lead to privilege escalation. Pairing the vulnerability characteristics and symptoms allow us to differentiate between these two overflows.

The pairing of characteristics and symptoms has other implications. By pairing characteristics with symptoms, we have a *requires-provides* relationship similar to those described in [20]. More on the relationship between our work and the requires-provides model is discussed in the future work section.

3 Classification

An unanticipated yet positive side effect of using characteristic trees for vulnerability classification is the simultaneous classification of the characteristics themselves. The next several sections discuss vulnerability classification, characteristic classification, taxonomies, and how classification is affected by the ability to form the complete characteristic set of modern systems.

3.1 Vulnerability Classification

Ideally, the complete characteristic set for a given system is determined before vulnerability classification takes place. This allows for creation of the complete characteristic tree, greatly simplifying the process of vulnerability classification. The tree is created using a top-down approach, repetitively dividing the characteristic set into hierarchical categories until the desired level of abstraction is achieved.

For example, let standard deck of cards represent a system. Let the following represent the complete characteristic set of this system:

$$\{\ \spadesuit,\ \clubsuit,\ \heartsuit,\ \diamondsuit,\ \mathbf{1},\ \mathbf{2},\ \cdots,\ \mathbf{10},\ \mathbf{K},\ \mathbf{Q},\ \mathbf{J},\ \mathbf{A}\ \}$$

These characteristics describe either *suit* or *value*. This becomes the first division of characteristics in the tree. Further inspection reveals that *suit* characteristics can be further divided into *black* or *red* suits. Similarly, *value* characteristics can be further divided into *number* or *face* values. This produces the complete characteristic tree shown in figure 7.

Once the complete characteristic tree is formed, vulnerability classification may occur. Classification is broken down into three major steps:

1. Define characteristic set for vulnerability.
2. Create characteristic tree.
3. Classify vulnerability.

There are a number of ways to define the characteristic sets of a vulnerability. One such method analyzes the steps taken to exploit the vulnerability [10]. After the characteristic set is defined, the characteristic tree is created. This can be done easily using a bottom-up approach since the characteristic tree for any vulnerability is simply a subtree of the complete characteristic tree. The process takes three steps:

1. *Identify relevant characteristic nodes in complete characteristic tree.* Recall that implementation nodes are not included in the complete characteristic tree. Therefore characteristic nodes are easily identified, as leaf nodes will always be characteristic nodes.

2. *Include the relevant characteristic nodes and all ancestors of those nodes.* This process can be seen as discarding all nodes and branches in the complete characteristic tree irrelevant to the current vulnerability.

3. *Add implementation nodes as necessary.* Any characteristic node may have one or more implementation nodes specific to the vulnerability.

For example, suppose the characteristics *heart* and *queen* (♥, **Q**) are required for a vulnerability to exist. In the tree the leaf nodes ♥ and **Q** are added

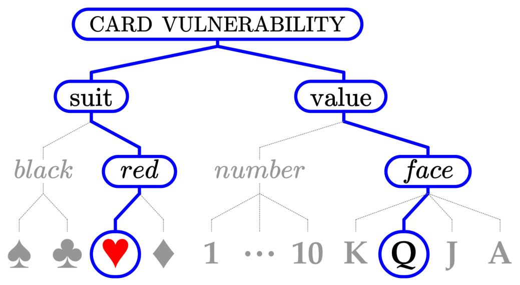

Figure 8: Characteristic tree for { ♥, **Q** }.

to the tree first. Then the ancestor nodes from the complete tree are included for both ♥ and **Q**. Therefore nodes *red* and *face* are added next, followed by *suit* and *value* before finally reaching the root node. Figure 8 illustrates how the tree for this vulnerability is derived.

Once the characteristic tree has been defined, classification is trivial. Each node in the characteristic tree represents a class (minus the root and implementation nodes), and the vulnerability belongs to all of these classes simultaneously. For the example given in figure 8, the vulnerability belongs to the class of red cards, heart cards, queen cards, and so on.

This is the simplest method of classification the characteristic trees provide. More complex classification uses logical operations to combine simple classes. For example, the class "royal flush" could be described as including only cards in the *face* class having the same suit characteristic.

3.2 Characteristic Classification

The characteristic tree not only provides the structure for classification of vulnerabilities, but also for classification of the individual characteristics themselves. This provides more depth and meaning to individual characteristics, and increases understanding of the impact certain characteristics may have.

Using the card example, the complete tree indicates that the queen characteristic (**Q**) indicates the face which also indicates value. On the other hand the heart characteristic (♥) indicates color and suit. The individual characteristics in this case now have two more levels of meaning with the tree than without.

4 Example: Characteristics

To illustrate characteristic sets, we examine the common problem of buffer overflows. Using work from [10] as a foundation, we break buffer overflows into two main types: data and executable buffer overflows. This breakdown is based on the observation that buffer overflows may occur in three different areas of process memory (data, stack, or heap) and aim to modify either variables, return addresses, or function pointers.

A **data buffer overflow** occurs when input overwrites existing data, causing the system to violate security policy. These overflows may occur in any area of process memory. A *direct* data buffer overflow directly modifies the value of some variable, whereas an *indirect* data buffer overflow modifies a pointer or causes incorrect data to be used.

An **executable buffer overflow** occurs when executable code is loaded into a buffer and eventually executed by altering either a return address or function pointer. A *direct* executable buffer overflow directly alters a return address, and hence must occur on the stack. An *indirect* executable buffer overflow does not involve modification of process state information. In most cases, these overflows modify a function pointer and may occur in any area of process memory.

4.1 Characteristics

Through the analysis in [10], we have isolated three characteristics common to all buffer overflows. This consists of an input, overflow, and modification characteristic as follows:

- `C_USR`: Program allows user to upload input of type `TYPE`. The value of `TYPE` may be `data`, `address`, or `instruction` depending on the specific overflow.
- `C_BUF`: User input exceeds bounds of associated buffer. This represents the actual buffer overflow, due to absent or incorrect bound checks on the user input.
- `C_MOD`: Modification of target `ELEM` allowed. The target `ELEM` may be a `variable`, `pointer`, `return_address`, or `function_pointer` depending on the specific overflow. This characteristic implies that this modification is not detected and countered.

Data buffer overflows involve user input of type `data` or `address` (for indirect overflows). The target of `C_MOD` is either a `variable` or a `pointer` to a variable. These overflows also require an additional characteristic describing policy violation:

- `C_POL`: Data value of `ELEM` affects process execution, causing policy violation. When `ELEM` is a pointer, the data value refers to the value resulting from following the pointer.

Executable buffer overflows involve user input of type `instruction`, `address`, or both. The target is always a `return_address` or `function_pointer`. In addition to the three main characteristics already discussed, these overflows have two other characteristics:

- `C_JMP`: Program can jump to the `MEM` in process memory. For direct executable overflows, `MEM` is always the `stack`. For indirect overflows, `PMEM` is usually the `heap`, but may sometimes be the `data` portion of process memory.
- `C_EXE`: Process can execute instructions stored in the `MEM`.

4.2 Classification

While we cannot create a complete characteristic tree for a system using these characteristics, we can fully represent a class of buffer overflow vulnerabilities. According to [10], all buffer overflows *must* have the first three characteristics (`C_USR`, `C_BUF` and `C_MOD`). In addition, data buffer overflows must also include the `C_POL` condition. Executable buffer overflows must include the `C_JMP` and `C_EXE` characteristics.

Therefore we can create complex classes (using logical operations combining simple classes) to describe all buffer overflow vulnerabilities. The general class of buffer overflows can be described as:

$$\texttt{C\_USR} \land \texttt{C\_BUF} \land \texttt{C\_MOD} \land \big(\texttt{C\_POL} \lor (\texttt{C\_JMP} \land \texttt{C\_EXE})\big)$$

Classes for data buffer overflows or executable buffer overflows can be expressed similarly. For example, data buffer overflows could be described as:

$$\texttt{C\_USR} \land \texttt{C\_BUF} \land \texttt{C\_MOD} \land \texttt{C\_POL}$$

Such that the following properties are true:

$$\begin{aligned} &\texttt{C\_USR}.\text{TYPE} = \texttt{data} \lor \texttt{address} \\ &\texttt{C\_MOD}.\text{ELEM} = \texttt{variable} \lor \texttt{pointer} \\ &\texttt{C\_POL}.\text{ELEM} = \texttt{C\_MOD}.\text{ELEM} \end{aligned}$$

4.3 Insights on Defense

Using these characteristics for buffer overflow vulnerabilities, we came across several insights. For example, by disabling the `C_BUF` characteristic all buffer overflow vulnerabilities are disabled.

Other defenses mostly focus on preventing executable buffer overflows, which are often more severe than data buffer overflows. These defenses focus on disabling those characteristics specific to these overflows. To negate the `C_MOD` characteristic, return addresses might be stored in memory areas write-protected from processes. Monitoring and countering the change of return addresses also negate this characteristic.

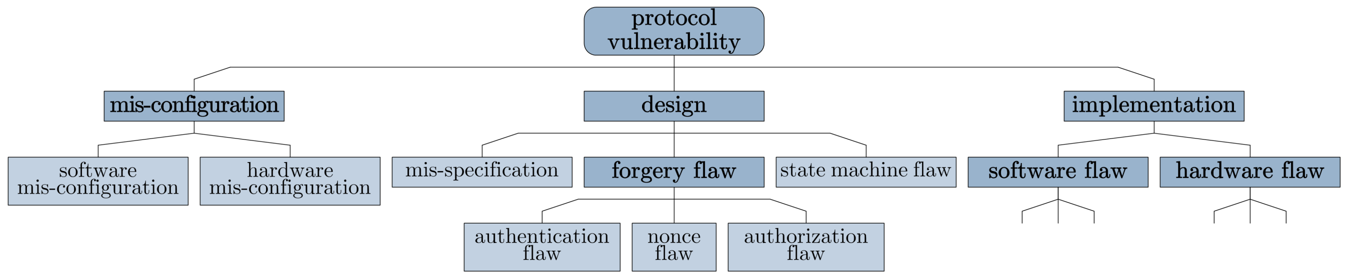

Figure 9: General characteristic tree for protocol vulnerabilities.

By understanding what characteristics these defenses disable, we also understand which types of buffer overflow vulnerabilities are prevented.

5 Example: Trees

We draw on an example from protocol vulnerabilities to illustrate characteristic trees [22]. We define the relevant characteristics, and present a specific instance of such a tree.

5.1 Characteristics

Several characteristics specific to protocol vulnerabilities have been identified, as shown in figure 9.

- Software/Hardware Mis-Configuration: Software configuration violates network policy.
- Mis-Specification: Intended semantics not captured by protocol specification. For example, mandating broken encryption schemes.
- Authentication Flaw: Protocol design verifies identities insufficiently or not at all.
- Nonce Flaw: Protocol design binds data to session non-uniquely or not at all.
- Authorization Flaw: Protocol design verifies permissions insufficiently or not at all.
- State Machine Flaw: Protocol specification tracks internal state insufficiently or not at all. Enables flooding and brute forcing.

The characteristics of software and hardware flaws are too numerous to enumerate here. We have defined a general set of symptoms, and are refining them for specific implementations.

5.2 Specific Instance

Figure 9 gives a generalized tree for protocol vulnerabilities, based on analysis of 24 protocol vulnerabilities. Specific vulnerabilities tend to be simple. For example, consider the Network Time Protocol version 2. A key index of 0 indicates signature verification should not be performed and the sender should not be trusted. However, the key index is not included in the hashing of the message, making it forgeable. An attacker could set the key index of another client to 0, effectively disabling that client's access to NTP election on a different host.

This vulnerability is captured with the tree in figure 10. This tree also includes the denial-of-service symptom that the vulnerability causes.

6 Classification Properties

Using characteristic trees allows us to create a classification scheme with properties beyond those we originally strived to achieve. The following sections discuss several properties we have encountered so far.

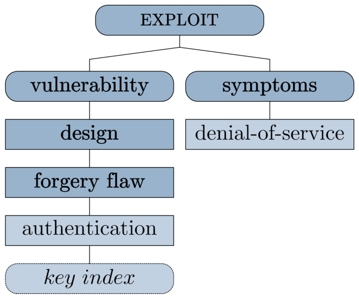

Figure 10: A network time protocol exploit.

6.1 Classification Requirements

Our simple tree-based approach to classification provides three of the main requirements for our classification scheme. The requirements, taken from the introduction, are as follows:

1. Any classification scheme should allow vulnerabilities that arise from similar conditions to belong in the same class.
2. Vulnerabilities may fall into multiple classes.
3. Classification should be primitive.
4. Classification should be well-defined.

Since the abstract nodes of the characteristic tree group similar characteristics together, vulnerabilities sharing similar characteristics will share the same abstract nodes. This satisfies the first requirement.

Also, each node in the tree represents a simple class. The vulnerability belongs to all these classes simultaneously, satisfying the second requirement.

If a node does not exist in the characteristic tree, the vulnerability does not belong to the corresponding class. Similarly, if the node exists the vulnerability does belong to that class. This simple existence test is unambiguous, and satisfies the third requirement.

The fourth requirement depends on the characteristics themselves. Without well-defined characteristics, classification cannot be considered well-defined. However, if the characteristics are well-defined, then the characteristic trees and classification are also well-defined. Hence our earlier statement that all characteristics must be well-defined.

6.2 Tree Properties

By applying a tree structure to characteristic sets we inherit the positive (and negative) properties of trees in general. Most importantly, the use of trees provide hierarchical organization, flexibility, and simplicity to our classification scheme.

The hierarchical organization allows us to provide multiple levels of abstraction as necessary to our characteristics. In many situations low levels of abstraction are necessary to provide the detail and assurance necessary for the environment. However, in situations where this is unnecessary these lower levels need not be included or analyzed to save time, space, and money.

The flexibility of trees allows us to adapt our characteristic trees to accommodate for new vulnerabilities. For example, in the absence of a predefined complete characteristic tree, we can define an initial generalized tree that we expand with new characteristics as more vulnerabilities are analyzed.

Finally, the simplicity of the tree structure allows us to focus on the characteristics of vulnerabilities themselves, and not on a complicated (or restricted) method of classifying them.

6.3 Obstacles

There is at least one significant downfall to this approach. Since characteristics are based on a particular system, one complete characteristic tree cannot be used for both Linux and Windows vulnerabilities unless the complete characteristic sets overlap. However, the abstract nodes are likely to overlap between complete trees, with the only differences being in the lower-level nodes more specific to the system.

The overlap of these characteristic sets would provide more insight to vulnerabilities in general. Eventually, it should be possible to build a high-level characteristic tree representing vulnerabilities in general without system dependencies.

7 Related Work

Our work is closest to the Program Analysis (PA) [5] classification scheme, as the PA notion of a "raw error pattern" is similar to our notion of a "characteristic." However, both the RISOS [1] and Program Analysis schemes do not take into account level of abstraction, and classify vulnerabilities using generic categories that lend themselves to ambiguity. Landwehr's scheme [18] focuses on the genesis, time of introduction, and location of vulnerabilities. Aslam [2, 3] classifies vulnerabilities using a decision tree, and does not allow vulnerabilities to fall into multiple classes.

Some classification schemes have specific goals. Endres presented a classification scheme from analyzing errors from a specific subset of programs [14]. However, the focus of the work was to determine what meaningful conclusions may be drawn from analysis of errors, and is not immediately extensible as a general vulnerability classification scheme.

DeMillo and Mathur present a grammar-based classification scheme of faults [13]. However, their work is specific to problems found in TEX. Weber presents a taxonomy of computer intrusions, but focuses on creating good classes for evaluating intrusion detection systems [21].

Numerous other classification schemes exist, differing from ours in perspective or approach. In his thesis, Krsul presents another classification scheme based on assumptions made by programmers [16, 17]. Cohen presented a classification scheme based on attack and defense [11, 12].

Howard's taxonomy provides a scheme with a different perspective and set of properties than ours [15]. The taxonomy presented focuses on computer incidents, and uses events, actions, targets, and attacks for classification. Our approach focuses more on the characteristics present that allow such actions or attacks to occur.

The attribute categorization scheme presented by Ostrand and Weyuker allows for development of new characteristics similar to our scheme [19]. However, the method presented requires further development to be widely applicable, and does not make use of trees to handle abstraction and classification.

8 Future Work

Determining the complete characteristic set for modern systems may not be a tractable task. Despite this, an initial **generalized characteristic tree** may be formed. The generalized tree defines only the most abstract levels of the characteristic tree, allowing for characteristics to be "plugged in" as vulnerability classification progresses. The initial generalized tree should eventually evolve into the complete characteristic tree as more characteristics and vulnerabilities are discovered.

We are attempting to populate a generalized characteristic tree using known protocol and host vulnerabilities. Doing this for multiple systems will help identify shared characteristics, leading to general remediation techniques. From this work we intend on creating a simple language for expressing vulnerabilities classified with our approach. This language will enable analysts to perform policy-based reasoning, and is one of the strengths of our system.

We plan to investigate meaningful visualization of our results. Currently the simple tree representation is useful only for the complete (or generalized) characteristic tree and individual vulnerabilities. We also hope to provide a method of representing multiple vulnerabilities simultaneously. Visualization may allow us to simply analysis, presentation, and discovery of new relationships between vulnerabilities.

So far little work has been done on the symptoms of vulnerabilities (discussed in section 2.3). However, the combination of characteristics and symptoms may have some of the most exciting implications. By using the *requires-provides* approach given by Jigsaw [20], our work may eventually extend to generating attack chains and automated tools.

9 Conclusion

Our aim is to provide a simple, yet flexible method of classifying vulnerabilities based on characteristics. We also strive to achieve the properties listed in section 6.1, which distinguishes our work from formal taxonomies [16]. This includes providing a well-defined, primitive classification scheme that allows for vulnerabilities to fall into multiple classes. We believe the scheme we have outlined in this paper satisfies this requirement.

We focused on using characteristics to describe vulnerabilities to achieve well-definedness. Applying a tree structure to these characteristics allowed us to handle multiple levels of abstraction. Finally, to allow vulnerabilities to fall into multiple classes, we decided against making this a decision tree. Instead, each node in the tree represents a simple class. The vulnerability then belongs to all these simple classes simultaneously. Allowing multiple levels of abstraction also gave us flexibility and organization to our characteristics.

By finding the complete characteristic set for a given system, we can also use these trees to come up with a taxonomy of vulnerabilities. While each complete characteristic set depends on a specific system, we believe this to be the best way of identifying vulnerabilities for that system. Furthermore, we may be able to discover and prevent previously unknown vulnerabilities by examining the system for individual characteristics from the complete set.

Much work using this scheme needs to be done, especially regarding host-based vulnerabilities. However, initial results in protocol vulnerabilities and buffer overflow vulnerabilities have been obtained. In both cases classification was not only practical, but gave us further insight into the nature and potential defenses of these vulnerabilities.

Our main hypothesis states that a large set of vulnerabilities can be described by a smaller set of characteristics. The correctness of this hypothesis determines whether our work simplifies the vulnerability analysis of a system. Regardless, understanding the conditions that create vulnerabilities will help us guard against their introduction and exploitation.

Acknowledgements

We gratefully acknowledge the support of the National Science Foundation under grant CCR-0311723 to the University of California at Davis.

References

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

[2] Aslam, Taimur, "A Taxonomy of Security Faults in the Unix Operating System," Masters Thesis, COAST Technical Report 95-09, Department of Computer Science, Purdue University, 1995.

[3] Aslam, Taimur, Ivan Krsul, and Eugene H. Spafford, "A Taxonomy of Security Faults," *Proceedings of the National Computer Security Conference*, COAST Technical Report 96-05, 1996.

[4] Basili, Victor R. and Barry T. Perricone, "Software Errors and Complexity: An Empirical Investigation," *Communications of the ACM*, vol. 27, no. 1, pp. 42-52, January 1984.

[5] Bisbey II, Richard and Dennis Hollingworth, "Protection Analysis: Final Report," Unclassified Report ISI/SR-78-13 for DTIC AD A056816, University of Southern California, Information Sciences Institute, May 1978.

[6] Bishop, Matt, *Computer Security: Art and Science*, Boston: Addison Wesley Professional, 2003.

[7] Bishop, Matt, "Vulnerability Analysis," *Proceedings of the Second International Symposium on Recent Advances in Intrusion Detection*, pp. 125-139, September 1999.

[8] Bishop, Matt, "A Taxonomy of UNIX and System Network Vulnerabilities," Technical Report CSE-95-8, Department of Computer Science, University of California at Davis, May 1995.

[9] Bishop, Matt and David Bailey, "A Critical Analysis of Vulnerability Taxonomies," Technical Report CSE-96-11, Department of Computer Science, University of California at Davis, September 1996.

[10] Bishop, Matt, Damien Howard, Sophie Engle, and Sean Whalen, "A Taxonomy of Buffer Overflows," (Pending), 2005.

[11] Cohen, Fred, "Information System Attacks: A Preliminary Classification Scheme," *Computers & Security*, vol. 16, no. 1, pp. 29-46, 1997.

[12] Cohen, Fred, "Information System Defenses: A Preliminary Classification Scheme," *Computers & Security*, vol. 16, no. 2, pp. 94-114, 1997.

[13] DeMillo, Richard and Aditya Mathur, "A Grammar Based Fault Classification Scheme and Its Application to the Classification of the Errors of TEX," Technical Report SERC-TR-165-P, Software Engineering Research Center, September 1995.

[14] Endres, Albert, "An Analysis of Errors and Their Causes in System Programs," *ACM SIGPLAN Notices*, vol. 10, no. 6, pp. 327-336, 1975.

[15] Howard, John Douglas, "An Analysis of Security Incidents on the Internet 1989 - 1995," Ph.D. Thesis, Carnegie Mellon University, 1998.

[16] Krsul, Ivan, "Software Vulnerability Analysis," Ph.D. Thesis, COAST Technical Report 98-09, Department of Computer Sciences, Purdue University, 1998.

[17] Krsul, Ivan, Eugene Spafford, and Mahesh Tripunitara, "Computer Vulnerability Analysis," COAST Technical Report 98-07, May 1998.

[18] Landwehr, Carl E., Alan R. Bull, John P. McDermott, and William S. Choi, "A Taxonomy of Computer Program Security Flaws," *ACM Computing Surveys*, vol. 26, no. 3, pp. 211-254, September 1994.

[19] Ostrand, Thomas J. and Elaine J. Weyuker, "Collecting and Categorizing Software Error Data in an Industrial Environment," *Journal of Systems and Software*, vol. 4, no. 4, pp. 289-300, November 1984.

[20] Templeton, S. J. and K. Levitt, "A Requires/Provides Model for Computer Attacks," *Proceedings of the New Security Paradigms Workshop 2000*, Cork Ireland, September 2000.

[21] Weber, Daniel, "A Taxonomy of Computer Intrusions," Masters Thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 1998.

[22] Whalen, Sean, Matt Bishop and Sophie Engle, "Protocol Vulnerability Analysis," Technical Report CSE-2005-04, Department of Computer Science, University of California, 2005.