Measuring a System's Attack Surface

Pratyusa Manadhata **Jeannette M. Wing**

January 2004
CMU-CS-04-102

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This research is sponsored in part by the National Science Foundation under Grant No. CCR-0121547 and in part by the Defense Advanced Research Project Agency (DARPA) and the Army Research Office (ARO), under contract no. DAAD19-01-1-0485. Views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NSF, DARPA, ARO, or the US Government.

Keywords: Security metrics, attack, attack class, attack surface, threat modeling

Abstract

We propose a metric to determine whether one version of a system is relatively more secure than another with respect to the system's *attack surface*. Intuitively, the more exposed the attack surface, the more likely the system could be successfully attacked, and hence the more insecure it is. We define an attack surface in terms of the system's *actions* that are externally visible to its users and the system's *resources* that each action accesses or modifies. To apply our metric in practice, rather than consider all possible system resources, we narrow our focus on a "relevant" subset of resource types, which we call *attack classes*; these reflect the types of system resources that are more likely to be targets of attack. We assign payoffs to attack classes to represent likelihoods of attack; resources in an attack class with a high payoff value are more likely to be targets or enablers of an attack than resources in an attack class with a low payoff value. We outline a method to identify attack classes and to measure a system's attack surface. We demonstrate and validate our method by measuring the relative attack surface of four different versions of the Linux operating system.

1 Introduction

In recent years, there has been an alarming increase in the number of successful attacks on systems due to an increase in the number of vulnerabilities found and exploited by attackers. Industry has responded to these incidents by increasing the effort to make systems more secure and less vulnerable. In the future, the amounts of money, time, and effort spent by industry will continue to increase proportionally to the increased expectations and demands of their customers for more trustworthy systems. Our work is motivated by the questions faced by industry today: How has industry's effort made to make a system more secure paid off? Is the most recent release of a system more secure than the earlier ones? How can we quantify the results?

In this paper we propose a metric to compare the relative security of two versions of the same system. Rather than measure the absolute security of a system with respect to some yardstick, we measure its relative security: Given two versions, A and B, of a system we measure the security of A relative to B with respect to the system's *attack surface*. Intuitively, by decreasing the exposure of the system's attack surface, e.g., by eliminating system features, we make it more secure.

1.1 A New Metric

Today we commonly use two measurements to determine the security of a system: at the code level, we count the number of bugs found (or fixed from one version to the next), and at the system level, we count the number of times a system version is mentioned in CERT advisories [24], Microsoft Security Bulletins [29], MITRE Common Vulnerabilities and Exposures (CVEs) [31] etc. We argue in Section 7 why both measurements, while useful, are less than satisfactory. In this paper, we propose a new security metric based on the notion of attack surface. It is a metric that strikes at the *design level* of a system: above the level of code, but below the level of the entire system.

The *system actions* externally visible to the system's users together with the *system resources* accessed or modified by each action constitute the system's *attack surface*. Intuitively, the more actions available to a user or the more resources accessible through these actions, the more exposed the attack surface. The more exposed the attack surface, the more likely the system could be successfully attacked, and hence the more insecure it is. We can reduce the attack surface to decrease the likelihood of attack and make a system more secure.

Attacks carried out over the years, however, show that certain system resources are more likely to be opportunities, i.e., targets or enablers, of attack than others. For example, services running as the privileged user `root` in UNIX are more likely to be targets of attack than services running as non-root users. Files with full control (e.g., `rwxrwxrwx` in Unix) are more likely to be attacked than files with less generous permissions. Symbolic links are highly likely to be used as enablers in attacks. In Windows, applications, such as Internet Explorer and Outlook Express, with VBScript, JScript, or ActiveX controls enabled are more likely to be enablers of attack than if such scripting engines and controls were disabled. Our method of measuring a system's attack surface recognizes that not all system resources should be treated equally. We identify the system resources that are opportunities of attack by a given set of properties associated with the resources, and categorize them into *attack classes*. These properties reflect the *attackability* of a type of resource, i.e., some types of resources are more likely to be attacked than other types.

Given a set of attack classes, and given two versions of a system, we measure whether one is more secure relative to the other by comparing them with respect to the attack classes. There are different possible ways of doing this comparison. For example, for each version, we might count the number of running instances of each attack class (e.g., the number of services running as root and the number of open sockets), and compare each version's respective numbers for each attack class. We might further refine these counts by weighing instances of some classes more than instances of others, where weights represent the likelihoods of attack. We could use a *payoff function* to assign these likelihoods, e.g., to assign higher payoff to services running as `root` than those running as non-root.

Using our method, it is meaningful to compare two similar systems (e.g., two versions of Red Hat, or Red Hat and Debian) rather than two completely unrelated systems (e.g., Linux and Windows) because the unrelated systems would have different sets of attack classes. We emphasize this point in Section 7 when we compare our work on Linux to previous similar work on Windows [11].

1.2 Contributions and Roadmap

Our contributions in this paper are three-fold:

- In terms of a state machine model of the system, we present formal definitions of attack, attack surface, and attack class. Our definitions are new, and in particular more precise than found in earlier work [11].
- We outline a general method to measure the attack surface of any system. We explain what inputs the user must provide to use our method in practice.
- We demonstrate our method on the Linux operating system. We identify 14 attack classes and measure the attack surface of four different versions of Linux. Our results are consistent with the perceived security level of these four versions.

The rest of this paper is organized as follows. In Section 2, we introduce our state machine model and point out the key differences from other similar models. In Section 3, we present the formal definitions of attack, attack class, and attack surface. We explain our method of attack surface measurement in Section 4. We demonstrate the use of our method for the Linux operating system in Section 5. We discuss the pros and cons of our approach in Section 6 and compare it to related work in Section 7. We conclude in Section 8.

2 State Machine Model

We use a state machine to model the system, the threat (adversary) trying to attack the system, the administrator of the system, and the users in the system.

2.1 Informal Overview

The use of state machines is not new in security literature. For example, state machines are used for intrusion detection ([7] and [12]), and to model security policies ([2] and [17]). Our state machine, however, differs from standard state machines found in the literature as follows:

- We explicitly represent an access matrix in the state of the state machine, thereby allowing us to represent the set of principals (e.g., User and Administrator) explicitly.
- We represent the system itself as a separate entity in our model and not as a principal.
- We distinguish both the threat and the system administrator as system principals different from other system users.

These differences allow us to partition the set of actions of a state machine into pairwise-disjoint sets of actions. By tagging the actions with the executor of an action (i.e., a principal or the system) in a given execution sequence of actions, we can easily and succinctly define the notion of *attack*.

2.2 Formal Definition

A state machine M $= \langle S, I, A, T\rangle$ is a four-tuple where S is the set of states, I $\subseteq$ S is the set of initial states, A is the set of actions, and T is the transition relation.

We assume a set of potentially existing resources, *Resource*, partitioned into disjoint typed sets. The set of states, S, in a state machine ranges over the type 2^{State}, where the type *State* itself is defined as follows:
$State = Env \times Store \times Access\_Matrix$
$Env = Name \rightarrow Resource$
$Store = Resource \rightarrow Value$
$Access\_Matrix = Principal \times Resource \times Rights$

Given a state $\langle e, s, am\rangle \in S$, the environment e is a mapping of names to typed resources and the store s is a mapping of typed resources to their typed values. The access matrix am is a triple similar to Lampson's access matrix [13], where our principals are equivalent to Lampson's domains and our resources are equivalent to Lampson's objects. To be concrete in this paper we define $Principal$ = {Threat, Administrator, User}. The principal Threat is the adversary who attacks the system with some goal, Goal, in mind. We assume that we can represent the Goal of the Threat as a predicate over the resources in the system. The Administrator tries to protect the system and tries to prevent the Threat from achieving its Goal. The User tries to get some useful work done. For simplicity, we assume only one user in the system, but our model is general enough to handle multiple users. Access rights definitions are specific to the system being modeled. For instance, on the UNIX operating system, *Rights* = {r, w, x}. Representation of the access matrix as a separate entity in the state makes specifying the pre- and post-conditions of actions convenient. Finally, to distinguish between actions taken by a principal and those performed by the system, in our model we represent the system itself by a special entity System.

The action set A consists of the actions of the System, the Threat, the Administrator, and the User. Each action is specified by its pre- and post-conditions and is tagged to identify the executor of the action. $A = A_S \uplus A_T \uplus A_A \uplus A_U$, where A_S is the action set of the System; A_T, the action set of the Threat; A_A, the action set of the Administrator; and A_U, the action set of the User. $\uplus$ stands for disjoint union.

The transition relation T is defined as $T \subseteq S \times A \times S$. For any action $a \in A$, if *a.pre* and *a.post* are the pre- and post-conditions of a, we define the set of transition triples involving the action a to be $a.T = \{\langle x, a, x'\rangle : S \times A \times S \mid a.pre(x) \Rightarrow a.post(x, x')\}$. T is the union of all such sets, $a.T$, for each action $a \in A$. For an action $a \in A$, we define a function, *Res: predicate* $\rightarrow$ *Resource*, such that for each resource, r, appearing in the predicate, p, $r \in Res(p)$[1]. We define the corresponding function for an action, *Res: action* $\rightarrow$ *Resource* to collect all resources appearing in the pre- and post-conditions of the specification of action a. Formally, $Res(a) = Res(a.pre) \cup Res(a.post)$.

The System, the Threat, the Administrator, or the User can cause the state machine M to change its state by executing their respective actions. A *state transition*, $\langle x, a, x' \rangle$, is the execution of action a in state x resulting in the new state x'. A change of state of M involves the following observable behaviors: (1) addition, deletion or modification of a resource, and (2) modification of the entries of the access matrix. The addition or deletion of a resource changes a state's environment, store, and access matrix. The modification of an existing resource changes a state's store and possibly its access matrix.

3 Definitions and Examples

We use our state machine model to define formally notions of attack, attack surface, and attack class.

3.1 Attack

Let $\mathrm{M} = \langle S, I, A, T \rangle$ be the state machine representing the system under attack and Goal the state predicate characterizing the adversary's goal to be achieved in attacking the system.

Definition 1 *An* attack *is a finite sequence of action executions $a_1, .., a_i, .., a_n$ such that:*

- $\forall 1 \leq i \leq n \;.\; a_i \in A$;
- $a_1 \in A_T$;
- $\exists 1 < i \leq n \;.\; a_i \in A_S$; *and*
- *Goal is satisfied in the state reached by M after execution of a_n.*

An attack includes actions from the action sets of the System, the Threat, the Administrator, and the User. Since an attack is initiated by the Threat, the sequence starts with an action of the Threat. The sequence includes at least one action of the System to model the exploitation of some system vulnerability by the Threat in the attack. Finally, the adversary's goal should hold at the end of the attack.

To illustrate our formalism, let us first consider the specification of two actions: SEND_STRING$_T$ and PROCESS_STRING$_S$. Recall that a system state is a triple, $\langle e, s, am \rangle$, of an environment, store, and access matrix; in particular, when we write e (am) below, we mean the environment (access matrix) component of the state $\langle e, s, am \rangle$. For each action specification, we use a bar over a variable name, $\bar{x}$, to denote the resource itself, i.e., $e(x)$; an unprimed variable name, x, to denote the value of the resource, i.e., $s(e(x))$, and a primed variable name, x', to denote its value in the post-state, i.e. $s'(e'(x))$. We specify state changes explicitly: a resource named by a variable that remains unprimed in the post-state is assumed not to change.

[1]This function can be inductively defined over the syntax of the predicate language, which we intentionally do not fix in this paper. First-order logic or temporal logic would both be natural choices for a predicate language.

Below, we assume the type *channel* has functions *enqueue: channel* $\times$ *string* $\rightarrow$ *channel* and *dequeue: channel* $\rightarrow$ *string* to write data to and read data from the channel; it also has the function *empty: channel* $\rightarrow$ *boolean* that returns true if the channel is empty; false, otherwise. The type *process* has a function *display: process* $\times$ *string* $\rightarrow$ *unit* to print a string on the user's terminal and a function *x_load: process* $\times$ *string* $\rightarrow$ *executable* to extract a payload from an input string, returning a resource of type *executable*. The type *string* has a function *length: string* $\rightarrow$ *int* that returns the length of the string. We associate a function *eval: executable* $\rightarrow$ *unit* with every executable in the system; when invoked it results in the evaluation of the executable. Note that for an executable, E, the predicate *E.pre* $\Rightarrow$ *E.post* captures the effect of evaluating E.

action SEND_STRING$_T$(C: channel, I: string)
pre $\langle Threat, \bar{C}, rw\rangle \in am$
post $C' = enqueue(C, I)$

action PROCESS_STRING$_S$(C: channel,
P: process)
pre $\neg$ $empty(C)$
post $\exists I \,.\, I = dequeue(C) \wedge$
$(length(I) \leq 512 \Rightarrow display(P, I)) \wedge$
$(length(I) > 512 \Rightarrow$
$\exists E.(E = x\_load(P, I) \Rightarrow$
$E.pre \Rightarrow E.post))$

The effect of executing SEND_STRING$_T$ is to enqueue a string onto a channel. The effect of executing PROCESS_STRING$_S$ is to display a string if its length is ≤ 512 and execute the string's extracted payload, otherwise.

Now we give a hypothetical attack, where the Threat exploits a buffer overrun in a process P running in the system by sending through the communication channel C a string X whose length exceeds 512. (We give another example describing a real-life attack in the appendix. It also illustrates an attack with a User action.)

Informally, the adversary's goal is to execute some arbitrary code, *E: executable*, in the system; formally, we represent this Goal as the predicate *E.pre* $\Rightarrow$ *E.post*. The attack on the System consists of the following sequence of two action executions, where time runs down the page. We italicize the state predicates before and after the attack.

$$\{\exists\bar{C} : channel \in e \wedge \exists\bar{E} : executable \in e \,\wedge\, \exists\bar{P} : process \in e \wedge \exists\bar{X} : string \in e \,\wedge$$
$$\langle Threat, \bar{C}, rw\rangle \in am \,\wedge\, empty(C) \wedge length(X) > 512\}$$
$$\text{SEND\_STRING}_T(\text{C, X})$$
$$\text{PROCESS\_STRING}_S(\text{C, P})$$
$$\{\exists E : executable \,.\, E.pre \Rightarrow E.post\}$$

Since the length of the string X sent to SEND_STRING$_T$ is greater than 512, the last conjunct of the post-condition of PROCESS_STRING$_S$ holds; thus Goal is achieved at the end of this sample attack.

For a given attack, the target will appear as one of the resources of one of the system actions of the attack. In the example above, *Res(*PROCESS_STRING$_S$*)* = {I, C, E, P}, and the process P is the target of attack. In addition, the specification of (at least) one of the system actions in the attack reflects the vulnerability of the System exploited by the Threat. In the example, the post-condition of PROCESS_STRING_S reflects the vulnerability of the system, i.e., if the length of the input string I is greater than 512, the process P executes arbitrary code in I sent by the Threat. The intended (ideal) behavior for PROCESS_STRING_S is to display the input string I, no matter what its length.

3.2 Attack Surface

Definition 2 *The* attack surface *of the System is the pair,* $\langle A_S, \bigcup_{a \in A_S} Res(a) \rangle$*, where the first component is the set of system actions and the second is the collective set of resources,* $Res(a)$*, for each system action,* $a \in A_S$*.*

Note that each system action $a \in A_S$ can potentially be part of an attack and hence contributes to the attack surface. Even though a system action may not have appeared in any attack seen to date, it can be part of a future attack, exploiting vulnerabilities not yet discovered or fixed.

Similarly, by our definition, every system resource can potentially be part of an attack surface. In reality, however, not all system resources have the same likelihood of being a target or enabler of an attack. We use attack classes to capture this intuition.

3.3 Attack Class

To motivate our definition of attack class, consider the general resource type *service*. In practice, not all services have the same "attackability," i.e., likelihood of attack. We might want to distinguish between services running as `root` and services running as `non-root`. Or, while we might have a general resource type *file*, we might want to distinguish among files that allow full control, those that allow only read/write access, and those that allow only read access; and we may not care about the remaining types of files. These kinds of distinctions can vary across different kinds of systems; for example, in a medical databases we might try to gain access to patient records, but in a nuclear control system, we might try to gain access to the sensor and actuator processes. We want our attack surface metric to be applicable across this broad range of systems.

In order to characterize these distinctions formally, we use "properties of interest" that are relevant to a given system. In practice, these properties take into consideration the aspects of resources the Threat finds easy to exploit (e.g., weak access control on files). To be concrete, we assume that we are given a set of properties, each expressed as a predicate over resource types.

A set of properties, *Prop*, extends a given set of types, *Type*, by introducing new types, each a subtype of some type in *Type*. This extended type system defines a type hierarchy, where types are related by a subtype relation (e.g., see Liskov and Wing's behavioral notion of subtype [14]). For example, *nobody_account* is a subtype of *user_account* because an account of type *nobody_account* has all the properties of *user_account* with the further distinguishing property that its `user_id` is `nobody`. For types S and T, we write $S \leq T$ if S is a subtype of T. The subtype relation characterizes when properties of a resource of a type, T, are preserved by any of T's subtypes. Operationally, if $S \leq T$ then anywhere we expect a resource of type T we can substitute a resource of type S. In general, a type hierarchy is a forest of directed acyclic graphs, i.e., a type might have more than one parent.

Assume we have a type hierarchy relation, *Induce*, that takes as input a set of properties, *Prop*, and a set of types, *Type*, and yields both a new set of types, *SType*, and a subtype relation, $\leq$, such that:

- $SType \supset Type$. That is, *Induce* extends a given set of types with new ones.
- $\leq$ is a subtype relation on types in *SType*.
- $\forall S \in SType \backslash Type\ \exists T \in Type\ .\ S \leq T$. That is, each new type is a subtype of some existing type.
- $\forall p \in Prop$, $\exists S \in SType \backslash Type$ such that $\forall x : S\ .\ p(x)$. That is, each property plays a role in defining some new type.
- $\forall S \in SType \backslash Type$, $\exists p \in Prop$ such that $\forall x : S\ .\ p(x)$. That is, each new type derives from some property in *Prop*.

The relation *Induce* projected on its first type in its domain (*Prop*) and mapped onto the first type in its range (*SType*) is total and onto. Note that a property can be used to define more than one new type and a new type can be derived from more than one property. If *Induce*($\langle Prop, Type\rangle$, $\langle SType, \leq\rangle$) we say the set of properties *Prop induces* a type hierarchy on *Type*, producing a subtype relation, $\leq$, on types in *SType*.

Definition 3 *Given a set of properties, Prop, and a set of resource types, Type, let Type_Hierarchy be the subtype hierarchy induced by Prop on Type, i.e., Induce(*$\langle Prop, Type\rangle$, $\langle Type\_Hierarchy, \leq\rangle$*). The* attack classes *of a system are all the types in Type_Hierarchy that are leaf nodes, i.e., have no subtypes of their own.*

Note that by defining an attack class to be a type with no child node in the type hierarchy, we ensure that all attack classes are disjoint, and thus any count based on the numbers of elements in a class will not double count resources. (We could give a less restrictive definition of attack class, but then any straightforward counting method, e.g., based on the number of instances per class, should make sure instances are not double counted.)

The set of properties specified by the user captures how likely resources of a given type will be attacked. For example, two general resource types in the Linux operating system are *service* and *user_account*. We can use the predicate *service_running_as_root: service* $\rightarrow$ *boolean* to categorize the services into two attack classes: *service_running_as_root* and *service_running_as_non-root*. We can use the predicates *user_id: user_account* $\times$ *id* $\rightarrow$ *boolean* and *group_id: user_account* $\times$ *id* $\rightarrow$ *boolean* to categorize the user accounts into the attack classes *user_account_with_user_id=root_or_group_id=root*, *nobody_account*, and *all_other_accounts*. (Our type hierarchy for Linux presented in Section 5 is more elaborate.) In the Windows (as in Linux) operating system, one general resource type is *channel*. We can use the predicate *channel_protocol*: *channel* $\times$ *protocol* $\rightarrow$ *boolean* to categorize the resources of type channel into the attack classes *socket*, *RPC_end_point*, *named_pipe* and *all_other_channels*.

4 Attack Surface Measurement Method

In this section we outline a general method that can be used to identify the attack classes of a system and measure its attack surface. We base our method on our formal model and definitions described in earlier sections; we present it in a way so that it can be applied to any system.

4.1 Attack Surface Measurement

The attack surface of a system consists of the set of system actions A_S and the collective set of resources of each action $a \in A_S$. A naive but impractical way of measuring the attack surface is to enumerate the set of system actions of a given system and count the number of resources in each of the action's resource set. We describe below a more practical, yet meaningful way to measure the attack surface based on the attack classes of the system. Then, given two versions, A and B, of a system we compare their relative attack surface exposure with respect to the attack classes.

4.2 Method

Consider a system with a fixed set, A_S, of system actions, each specified in terms of pre- and post-conditions. In practice, a system's API serves as the set of system actions.

Step 1. Identify the resources that are potential targets of attack as $\bigcup_{a \in A_S} Res(a)$ from the given set of system actions A_S. Let *Type* be the set of types of all these resources.

Step 2. Given a set, *Prop*, of properties of interest over the resources, induce a type hierarchy over the set, *Type*, of resource types identified in Step 1. Every leaf node in this type hierarchy is an attack class of the system. Let *Attack_Class* be the set of attack classes.

Step 3. Define a payoff function *F: Attack_Class* $\rightarrow [0, 1]$ to assign payoffs to each attack class identified in Step 2.

Step 4. Choose some k attack classes from the attack classes identified in Step 2. (We discuss why we include this step below.)

Step 5. Compare the two versions of the system, A and B, with respect to these k attack classes to obtain their relative attack surface exposure.

Some notes on these steps in our method:

In Step 2, we need to rely on our knowledge of the system to state the properties of interest. They will differ from system to system and they may change over time based on our experience with a system as it evolves over time.

In Step 3, payoffs represent the likelihoods of attack. An attack class with a high payoff indicates that resources of that class are more likely to be attacked than resources of an attack class with a lower payoff. One naive way of assigning payoffs is to count the number of times a resource appears in the pre- and post-conditions of system actions; we would assign higher payoffs to the resources having higher counts. Another way of assigning payoffs is based on a system's reported history and we give higher payoffs to attack classes that appear in a greater number of vulnerability bulletins. A more sophisticated approach for defining a payoff function is to quantify the "damage" the adversary can effect if resources in a given attack class are compromised, e.g., in terms of cost to repair the system.

In Step 4, we acknowledge that measuring an attack surface in practice need not involve all the attack classes of the system. We might choose the top k attack classes of a system based on the payoffs assigned in Step 3. Or, more pragmatically (as we will see in Section 5), we might simply choose the k attack classes for which we have an automated means of counting their sizes.

In Step 5, there are many ways to use the attack classes to do the comparison. One simple way is to count the number of instances of each attack class in both versions and compare the numbers. The higher the count for a given attack class, the more the attack surface exposure is for that class. Another way is to incorporate the payoffs identified in Step 3 as weights in these counts; e.g., we could count the weighted sum of all instances in each attack class with the same immediate supertype, and compare the versions with respect to the supertype, rather than its individual attack class subtypes. In other words, the attack surface contribution of a resource type T with attack classes $S_1, S_2, .., S_k$ as its subtypes is given by $\sum_{i=1}^{k} n(S_i) \times w_i$, where $n(S_i)$ is the number of instance of the attack class S_i and w_i is the payoff assigned to S_i in Step 3. A more sophisticated comparison might take into account the interactions between various attack classes, e.g., we can compare the number of sockets opened by services running as `root` and ignore the other sockets in the system.

4.3 Reducing the Attack Surface

Our formal model and measurement method suggest ways in which we can reduce the exposure of an attack surface:

- Reduce the number of system actions.
- Remove a known or potential system vulnerability by strengthening the pre- and post-conditions of a system action $a \in A_S$, e.g., in a way that prevents the Goal of the Threat from ever being achieved.
- Eliminate an entire attack class.
- Reduce the number of instances of an attack class.

5 Linux Example

In this section, we describe the results of measuring the attack surface of four versions of the Linux operating system.

Step 1 of our method requires that we identify all resources of the system that are potential targets of attacks. Since it is impractical to enumerate the set of system actions for Linux, and then identify all possible resources each action might access or modify, we derived the set of resource types indirectly. We considered all resources that appear in the MITRE CVEs [31] as potential targets of attack and identified their types accordingly.

In Step 2, we defined a set of properties over the resource types identified in Step 1, and induced a type hierarchy over the extended set of resource types. Every leaf node in this type hierarchy is an attack class, resulting in 14 attack classes for Linux. Fig 1 depicts the type hierarchy and the 14 Linux attack classes. For example, in our type hierarchy, $\textit{nobody\_account} \leq \textit{user\_account} \leq \textit{all\_resource}$ and $\textit{symbolic\_link} \leq \textit{all\_resource}$. We use descriptive names for the types and subtypes to be suggestive of the properties we used to induce the hierarchy.

In Step 3, we used the history of attacks on Linux based on CVEs to assign payoffs to the attack classes identified in Step 2. We did not assign explicit numeric payoff values because we did not plan to use the numeric values in Steps 4 and 5. Instead, we assumed a higher payoff for an attack class if the resources of that attack class appear a greater number of times in the CVEs. Note that

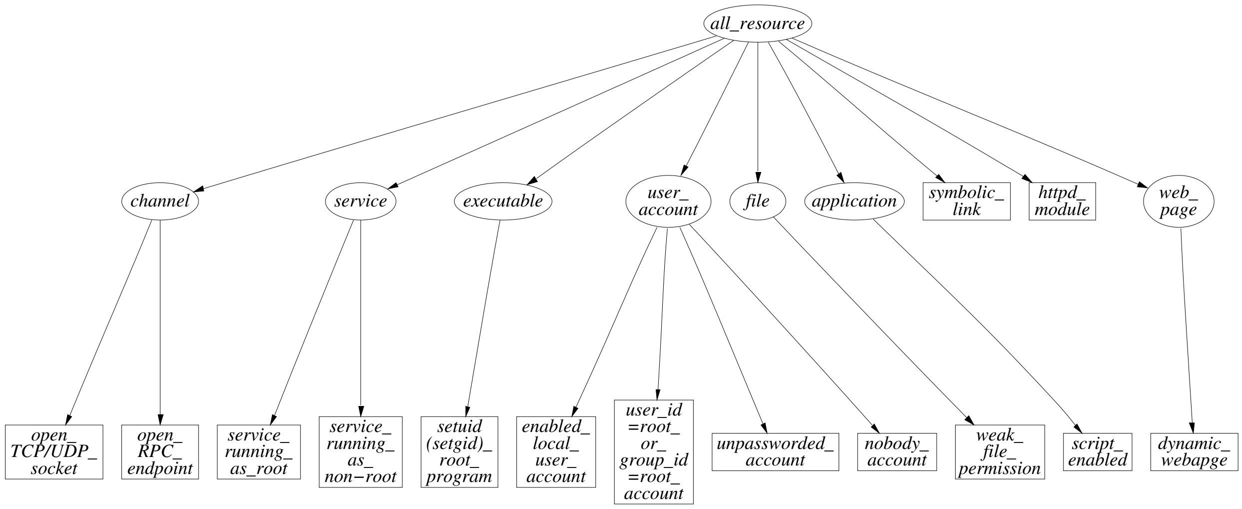

Figure 1: Linux Attack Classes: Attack Classes are represented by rectangular boxes.

counting the number of times a system appears in the CVEs is different from counting the number of times each attack class of the system appears in the CVEs. For example, consider two versions, A and B, of a system having ten attack classes and each appearing in 50 CVEs. Let two of the attack classes of version A appear in 25 CVEs each, ten of the attack classes of version B appear in 5 CVEs each, and the first two attack classes have lower payoffs compared to the remaining eight. Both A and B appear in same number of CVEs, but the attack surface exposure of B is more than that of A with respect to the ten attack classes.

In Step 4, we chose 11 attack classes for attack surface measurement out of the 14 identified in Step 2. We did not include three attack classes in our measurement since it was not possible to count the number of instances of each of them for all four versions of Linux. In Section 5.2, we explain in more detail why we omitted these three attack classes.

In Step 5, we counted the number of instances of each of the 11 attack classes for four versions of the Linux operating system and compared the numbers to get a relative measure of their attack surface.

5.1 Attack Classes

We identified the resources appearing in the publicly known vulnerabilities reported in the CVEs and CVE candidates list of MITRE [31]. We obtained further information about the vulnerabilities from the CERT Advisories [24], Debian Security Advisories [28], and Red Hat Security Advisories [33] referenced in the CVEs . We categorized the types of these resources into 14 attack classes. We describe below each attack class and give an example (CVE) of a vulnerability of a resource in that attack class.

5.1.1 Linux Attack Classes

open_TCP/UDP_socket: The services running on the system open TCP/UDP sockets and listen for client requests on them. Multiple sockets can be opened by a service and multiple services can share the same socket. This attack class is a subtype of the resource type *channel*. CVE-2001-0309 describes an attack involving open sockets— since the `inetd` daemon does not properly close sockets for internal services such as `daytime` and `echo`, an attacker can cause a denial-of-service attack by opening a series of connections to these services.

open_remote_procedure_call(RPC)_endpoint: Remotely accessible handlers for RPCs are registered in the system by RPC servers. This attack class is a subtype of the resource type *channel*. A remote attacker can exploit an integer overflow vulnerability in the `SunRPC xdr_array` function described in CVE-2002-0391 to execute arbitrary code on the system.

service_running_as_root: This attack class is a subtype of the resource type *service*. Examples of services running as root are `crond` and `telnetd`. CVE-1999-0192 describes a buffer overflow in the `telnetd` daemon which a remote attacker can exploit to gain `root` privilege on the system.

service_running_as_non-root: This attack class is a subtype of the resource type *service*. Examples of services running as non-root are `portpmap` and `rpc.statd`. CVE-2000-0666 describes a format string vulnerability in the service `rpc.statd` which a remote attacker can exploit to execute arbitrary code on the system.

setuid(setgid)_root_program: Setuid root programs are owned by `root` and execute in the context of `root` instead of the user who invokes them. This attack class is a subtype of the resource type *executable*. CVE-2000-0949 describes a heap overflow in the setuid root program `traceroute` which local users can exploit to execute arbitrary commands on the system.

enabled_local_user_account: This attack class is a subtype of the resource type *user_account*. Many of the attacks on Linux systems can be carried out only by local users. CVE-1999-0130 describes an exploit in which local users can gain `root` privilege by starting `Sendmail` in daemon mode.

user_id=root_or_group_id=root_account: This attack class is a subtype of the resource type *user_account* such that the `user id` or `group id` of the `user account` is `root (0)`. These accounts are potential targets of attack because of their enhanced privilege. CVE-2002-0875 describes a vulnerability in the daemon `fam` which unprivileged users can exploit to discover a list of files accessible to the `root` group.

unpassworded_account: This attack class is a subtype of the resource type *user_account* such that the `password` of the `user account` is set to `blank`. CAN-1999-0502 describes the presence of a unix account with default, null, blank or a missing password as a vulnerability of the system.

nobody_account: `Nobody` is a special user account created in the system. This attack class is a subtype of the resource type *user_account* such that the `user id` is set to `nobody`. CAN-2002-0424 describes a vulnerability in `efingerd` running as `nobody` which local users can exploit to gain `nobody` privilege by modifying their own `.efingerd` file and running `finger`.

weak_file_permission: This attack class is a subtype of the resource type *file* such that the access matrix entries of the file grant access rights to every user in the system. CVE-2001-1322 describes a vulnerability in `xinted` (runs with a default `umask` 0) which allows local users to read or modify files created by the programs running under `xinted` and not setting their safe `umask`.

script_enabled: This attack class is a subtype of the resource type *application* such that the applications are enabled to execute scripts. Examples of such applications are browsers and e-mail clients. CVE-2001-0745 describes a vulnerability in `Netscape` which a remote attacker can exploit to obtain sensitive user information via `Javascript`.

symbolic_link: This attack class consists of all resources of type *symbolic_link*. When a program running as `root` creates files in `/tmp` without checking for `symlink`, an attacker can create a symbolic link in `/tmp` before the program starts and hence can write to sensitive files. CVE-2000-0728 describes a vulnerability in the `xpdf` PDF viewer which local users can exploit to overwrite arbitrary files via symlink attack.

httpd_module: This attack class consists of all resources of type *httpd_module*. CAN-2003-0789 describes a vulnerability involving handling of CGI redirect in the `mod_cgid` module in `apache` which an attacker can exploit to view sensitive information.

dynamic_web_page: This attack class is a subtype of the resources type *web_page*. CVE-1999-0058 describes a buffer overflow vulnerability in the `php` program `php.cgi` which allows `shell` access to a remote attacker.

5.1.2 Attack Class Validation

We obtained 14 attack classes for Linux by identifying the resources that appear in MITRE CVEs. These 14 attack classes should be complete enough to cover vulnerabilities maintained in any other Linux vulnerability database. Thus, toward a partial validation of our 14 attack classes, we recasted some of the reported vulnerabilities of Linux available at the Bugtraq database [21] in terms of our attack classes. If our 14 attack classes are complete enough, then any vulnerability mentioned in Bugtraq should be covered by one of our attack classes.

For example, the vulnerability entry in the Bugtraq database with bugtraqid 7769 [22] describes a format string vulnerability, stack overflow, and file corruption in the `mod_gzip` module running in debug mode. The target of the attack involving the vulnerability is the resource `mod_gzip` and it is an instance of our Linux attack class *httpd_module*.

As another example, Bugtraqid 8732 [23] describes ASN.1 parsing vulnerabilities in `OpenSSL` which a remote attacker can exploit to cause a denial-of-service or to execute arbitrary code on the system. The resources that are the targets of this attack are the applications such as `ssh` that use OpenSSL. `ssh` is an instance of our Linux attack class *service_running_as_root*.

5.2 Attack Surface Measurements

We present the results of measuring the attack surface of following four versions of the Linux operating system.

- *Debian* is a Debian GNU/Linux 3.0r1 distribution obtained from Debian's website [27].
- *RH Default* is a Red Hat 9.0 Linux distribution obtained from Red Hat's website [32].
- *RH Facilities* is a customized Red Hat 9.0 Linux distribution installed by the Computing Facilities of CMU School of Computer Science [25].
- *RH Used* is an instance of *RH Facilities* after use by a graduate student for three months.

We took measurements for *Debian*, *RH Default*, and *RH Facilities* the very day each system was installed. We did not modify any of these three systems in any manner after installation. We took measurements for *RH Used* after three months of its installation. As described in Step 5 of Section 5, we counted the number of instances of each attack class in our measurement. The results of our measurements are shown in Table 1.

For the *weak_file_permission* attack class, we counted the number of file system objects with world-writable permission. We did not install any web server on the system running *Debian* and

Attack Class	Debian	RH Default	RH Facilities	RH Used
open_TCP/UDP_socket	15	12	40	41
open_remote_procedure_call(RPC)_endpoint	3	3	3	3
service_running_as_root	21	26	29	30
service_running_as_non-root	3	6	8	8
setuid(setgid)_root_program	54	54	72	72
enabled_local_user_account	21	25	33	34
user_id=root_or_group_id=root_account	0	4	3	3
unpassworded_account	0	0	2	2
nobody_account	1	1	1	1
weak_file_permission	7	7	21	37
script_enabled	1	2	2	2
symbolic_link	*	*	*	*
httpd_module	-	-	-	-
dynamic_web_page	-	-	-	-

Table 1: Attack surface measurement results

RH Default since the system running *RH Facilities* did not have a web server installed. Hence we did not count the numbers of instances of the attack classes *httpd_module* and *dynamic_web_page*. We did not count the number of instances of the attack class *symbolic_link* since it is impractical to determine whether the programs running as `root` check for `symlinks` before opening temporary files.

Our metric and method give us different ways to compare the security of different versions of a system:

- Default comparison: We compare the attack surfaces of *Debian* and *RH Default* to measure the relative security of different flavors (versions) of the system.
- Customized usage-based comparison: We compare the attack surfaces of *RH Default* and *RH Facilities* to observe the change in the security level of a system based on its customization.
- Time-based comparison: We compare the attack surfaces of *RH Facilities* and *RH Used* to monitor the security level of a system as it changes over time.

5.2.1 Debian vs. RH Default

As shown in Table 1, *RH Default* has higher counts in each of five attack classes, *Debian* has a higher count in one attack class, and both have the same counts in each of five attack classes. Hence the attack surface exposure of Red Hat is greater than that of Debian. Debian is perceived to be a more secure operating system and this is reflected in our measurement. We believe that even though the code base is the same for the two systems, design choices play an important role in making a system more or less secure.

5.2.2 RH Default vs. RH Facilities

As shown in Table 1, *RH Facilities* has higher counts in each of seven attack classes, *RH Default* has higher counts in one attack class, and both have the same counts in each of three attack classes. The attack surface exposure of the facilities distribution is more than that of the default distribution.

The facilities distribution is customized to make it more useful compared to the default distribution. For example, it has the `AFS` file system installed. It has services such as `lclaadmd, opshell, kopshell` and `terad` installed for remote management and network backup. Installing these features increases the counts for the attack classes *open_TCP/UDP_socket*, *service_running_as_root*, *enabled_local_user_account*, etc. Our results show that the attack surface exposure has increased with customization, thereby making the system less secure.

5.2.3 RH Facilities vs. RH Used

As shown in Table 1, *RH Used* has higher counts in each of four attack classes and both have the same counts in each of seven attack classes. The used version's attack surface exposure is greater than the initially installed version. The three-month use of the system increased the counts of the attack classes *open_TCP/UDP_socket*, *service_running_as_root*, *enabled_local_user_account*, and *weak_file_permission*. Our results show that the attack surface exposure has increased over time making the system less secure.

6 Discussion

In this section, first we make some qualifying remarks on our metric and Linux measurement results, and then we describe the advantages of our approach.

6.1 Caveats

We have some general caveats in using our attack surface metric in determining the relative security of different versions of a system.

- Our method measures the security of a *running* instance of a system. We are not measuring the system artifact (e.g., as manifest by its code), but rather a specific running version of it. Unlike a count of the number bugs in the code, it is a dynamic, not static measure.
- Our method measures the security of a system in a *given configuration*. A system typically has many settings; any combination of those settings yields a specific configuration.

Thus, it is important to realize that system's security level will change as its configuration changes over time. For example, initially a feature may be turned off but over time, the user might enable it, potentially increasing its attack surface, making it just as insecure as a system that initially has that feature turned on.

We also have caveats with respect to our specific results for Linux.

First, we chose 11 out of 14 attack classes in our attack surface measurement and our results should be interpreted in the context of these 11 attack classes. Suppose version A is more secure compared to a version B with respect to the 11 attack classes. If we were to include the remaining three attack classes in the measurement, version A may not be more secure, e.g., there may be higher counts for these classes for A than for B. Moreover if the payoffs for these three classes are higher than for the 11 we counted, and we weighed our counts by payoff, then A would look significantly worse than B.

Second, CMU School of Computer Science Computing facilities has replaced many standard services such as `telnetd` and `rshd` with local versions that use `Kerberos` for authentication and encryption in the distribution *RH Facilities*. These local versions are perceived to be more secure than the standard versions. Although we did not consider these security enhancements in our measurement, here is how we would: To account for the higher security level of Kerberized services, we would introduce a new predicate *kerberos_support: service* $\rightarrow$ *boolean* to allow us to distinguish whether a service uses `Kerberos`. We could use this predicate along with the previously defined predicate *service_running_as_root: service* $\rightarrow$ *boolean* to define four distinct subtypes (attack classes) of the type *service*: *kerberized_service_running_as_root*, *non-kerberized_service_running_as_root*, *kerberized_service_running_as_non-root*, and *non-kerberized_service_running_as_non-root*. We would then assign lower payoffs to the attack classes whose services use `Kerberos`.

6.2 Advantages

The use of attack surface as a security metric and our method of measuring the attack surface have the following advantages.

First, our metric is a relative measure of security. It is difficult to identify a yardstick for measuring a system's absolute security. Instead, we find it more practical and more useful to compare the security of two versions of a system with respect to a given set of attack classes. Our metric can be used to determine whether a new release of a system is more secure than an earlier version. By measuring the attack surfaces of different versions, system designers could potentially reduce the number of security patches that are released after the deployment of a new version in the field.

Second, our metric can be used to track the security level of the system over time by measuring the attack surface at regular intervals. We can observe the change in security level as different resources are turned on and off as required.

Finally, our method of measuring the attack surface leverages our knowledge of and experience with the system. Use of specific domain knowledge plays an important role in Steps 2-5 outlined in Section 4.2. Our method gives us the flexibility to refine upon our choice of properties that induce our type hierarchy, our assignment of payoffs, and our comparison method. We would make these refinements because of newly acquired knowledge and experience, changes in the threat model, or changes in technology.

7 Related Work

The use of attack surface as a security metric for any system is a novel idea. Michael Howard of Microsoft first introduced it informally for the Windows operating system. We compare our generalization of this metric and its application to Linux in Section 7.1 and then compare this metric to other security metrics in Section 7.2.

Attack Class	Windows	Linux
1	Open sockets	*open_TCP/UDP_socket*
2	Open RPC endpoints	*open_remote_procedure_call(RPC)_endpoint*
3	Services running as SYSTEM	*service_running_as_root*
4	Enabled accounts	*enabled_local_user_account*
5	Enabled accounts in admin group	*user_id=root_or_group_id=root_account*
6	Guest account enabled	*unpassworded_account*
7	Weak ACLs in FS	*weak_file_permission*
8	JScript enabled	*script_enabled*
9	Active Web handlers	*httpd_module*
10	Dynamic web pages	*dynamic_web_page*

Table 2: Comparison of Windows and Linux Attack Classes

7.1 Attack Surface Metric

Our work is inspired by Howard's Relative Attack Surface Quotient (RASQ) measurements for the Windows operating system [10] further elaborated by Pincus and Wing [11]. Howard, Pincus and Wing give a list of twenty attack classes for the Microsoft Windows operating system and compare seven versions of Windows [11]. The contributions of this paper as compared to the earlier work done for Windows [10, 11] are three-fold:

- We define the notion of attack, attack surface, and attack class more formally and in terms of a different state machine model. The significant differences in our *state machine model* are in making the access matrix explicit and in distinguishing the system as an entity different from its principals. The significant contributions in our *definitions* are in further dividing types of resources into attack classes by introducing a type hierarchy and distinguishing among the attack classes based on their attackability.

- In Section 4, we present a *method* for applying our metric so that others can use the notion of attack surface for any system. The method requires identifying resources that are potential targets of attacks and identifying interesting properties of the resources to characterize their attackability. We also allow users to specify a payoff function for attack classes, to help determine what attack classes to use for comparing two versions of a system.

- We apply our method and metric to *Linux*. The concrete contributions are in identifying the 14 attack classes for Linux and in measuring the attack surface of four different versions of Linux.

We compare the attack classes of Windows and Linux in Table 2. There exists a one-to-one mapping between ten attack classes of Windows and Linux. The remaining ten attack classes of Windows have no corresponding equivalents in Linux. Similarly the remaining four attack classes of Linux have no corresponding equivalents in Windows.

7.2 Other Security Metrics

Many have done work in the area of detection of bugs at the code level [8, 9, 19, 20]. Using bug counts as a security metric has the following disadvantages: (1) the bug detection process may miss some bugs and may raise false positives, and (2) equal importance is given to all bugs, even though some bugs are easier to exploit than others.

Many organizations, such as CERT [24] and MITRE [31], and websites, such as SecurityFocus [21], track vulnerabilities found in various systems. Counting the number of times a system appears in these bulletins is not an ideal metric because it ignores the specific system configuration that gave rise to the vulnerability, and it does not capture a system's future attackability.

Our approach lies in between these two approaches: It is at a higher level abstraction than the code level, implicitly giving importance to bugs based on ease of exploit. It is at a lower level of abstraction than the entire system, linking vulnerabilities to specific system configurations.

Browne et al. [5] give a mathematical model to reflect the rate at which incidents involving exploits of vulnerability are reported to the CERT. Beattie et al. [3] give a model for finding the appropriate time for applying security patches to a system for optimal uptime. Both of these studies focus on vulnerabilities with respect to their discovery, exploitation and remediation over time, rather than a single system's collective points of vulnerability.

There has been some work done in the area of quantitative modeling of the security of a system. Brocklehurst et al. [4, 15] measure the operational security of a system by estimating the effort spent by an attacker to cause a security breach in the system and the reward associated with the breach. Alves-Foss et al. [1] use the System Vulnerability Index—obtained by evaluating factors such as system characteristics, potentially neglectful acts and potentially malevolent acts—as a measure of computer system vulnerability. Voas et al. [18] propose the minimum-time-to-intrusion (MTTI) metric based on the predicted period of time before any simulated intrusion can take place. MTTI is a relative metric that allows the users to compare different versions of the same system. Ortalo et al. [16] model the system as a privilege graph [6] exhibiting its vulnerabilities and estimate the effort spent by the attacker to attack the system successfully, exploiting these vulnerabilities. The estimated effort is a measure of the operational security of the system. These works focus on the vulnerabilities of a system as a measure of its security, where as we use the notion of the attackability of various resources of the system as a measure of its security.

8 Conclusions

Our state machine model is general enough to model the behavior of the system, the threat, the administrator and the users on the system. Our attack surface measurement method can be applied to any system. Our application of our metric and method to Linux give results that confirm perceived beliefs about the relative security of four versions of the Linux operating system.

Measurement of security, both quantitatively or qualitatively, has been a long-standing challenge to the community. We view our work as a first step towards a meaningful and practical metric for security measurement (e.g., see CRA Grand Challenge # 3 [26]). We believe that the best way to begin is to start counting what is countable and then use the resulting numbers in a qualitative manner. We believe that our understanding over time would lead us to more meaningful and useful quantitative metrics for security measurement.

9 Acknowledgments

We thank Michael Howard and Steve Lipner for their encouragement to formalize and generalize Mike's original work on the Relative Attack Surface Quotient. We also thank Jon Pincus for his enthusiastic collaboration on related work carried out at Microsoft. Both Mike and Jon gave us seeds for how to think about applying the attack surface metric to Linux.

References

[1] J. Alves-Foss and S. Barbosa, *Assessing Computer Security Vulnerability*, ACM SIGOPS Operating Systems Review 29,3 (1995) p. 3-13.

[2] Lujo Bauer, Jarred Ligatti and David Walker, *More Enforceable Security Policies*, Workshop on Foundations of Computer Security (2002).

[3] Steve Beattie, Seth Arnold, Crispin Cowan, Perry Wagle, Chris Wright and Adam Shostack, *Timing the Application of Security Patches for Optimal Uptime*, Proceedings of LISA'02:16th Systems Administration Conference (2002).

[4] S. Brocklehurst, B. Littlewood, T. Olovsson, and E. Johsson, *On Measurement of Operational Security*, Proceedings of the 9th Annual Conference on Computer Assurance (1994).

[5] Hilary Browne, John McHugh, William Arbaugh and William Fithen, *A Trend Analysis of Exploitations*, IEEE Symposium on Security and Privacy (2001).

[6] M. Dacier and Y. Deswarte, *Privilege Graph: An extension to the Typed Access Matrix Model*, Proceedings of the Third European Symposium on Research in Computer Security (1994).

[7] S.T. Eckmann, G. Vigna, and R.A. Kemmerer, *STATL: An Attack Language for State-based Intrusion Detection*, Journal of Computer Security 10,1/2 (2002) p. 71-104.

[8] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem, *Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions*, Symposium on Operating System Design and Implementation (2000).

[9] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf, *Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code*, ACM Symposium on Operating Systems Principles (2001).

[10] Michael Howard, *Fending Off Future Attacks by Reducing Attack Surface*, `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncode/html/secure02132003.asp` (2003).

[11] Michael Howard, Jon Pincus and Jeannette M. Wing, *Measuring Relative Attack Surfaces*, Proceedings of Workshop on Advanced Developments in Software and Systems Security (2003).

[12] K. Ilgun, R.A. Kemmerer, and P.A. Porras, *State Transition Analysis: A Rule-Based Intrusion Detection Approach*, IEEE Transactions on Software Engineering 21,3 (1995) p. 181-199.

[13] B. Lampson, *Protection*, ACM Operating Systems Review 8,1 (1974) p. 18-24.

[14] Barbara H. Liskov and Jeannette M. Wing, *A Behavioral Notion of Subtyping*, ACM Transactions on Programming Languages and Systems 16,6 (1994) p. 1811-184.

[15] B. Littlewood, S. Brocklehurst, N. Fenton, P. Mellor, S. Page, and D. Wright, *Towards Operational Measures of Computer Security*, Journal of Computer Security 2,2/3 (1993) p. 211-229.

[16] R. Ortalo, Y. Deswarte, M. Kaâniche, *Experimenting with Quantitative Evaluation Tools for Monitoring Operational Security*, IEEE Transactions on Software Engineering 25,5 (1999) p.633-650.

[17] Fred B. Schneider, *Enforceable Security Policies*, ACM Transactions on Information and System Security 3,1 (2000) p. 30-50.

[18] J. Voas, A. Ghosh, G. McGraw, F. Charron, and K. Miller, *Defining an Adaptive Software Security Metric from a Dynamic Software Failure Tolerance Measure*, Proceedings of the 11th Annual Conference on Computer Assurance (1996).

[19] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Ai, *A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities*, Network and Distributed System Security Symposium (2000).

[20] Xiaolan Zhang, Antony Edwards, Trent Jaeger, *Using CQUAL for Static Analysis of Authorization Hook Placement*, Proceedings of the Usenix Security Symposium (2002).

[21] Bugtraq,
`http://www.securityfocus.com/archive/1`

[22] Bugtraq id=7769,
`http://www.securityfocus.com/bid/7769`

[23] Bugtraq id=8732,
`http://www.securityfocus.com/bid/8832`

[24] CERT Advisories,
`http://www.cert.org/advisories/`

[25] CMU SCS Computing Facilities,
`http://www.cs.cmu.edu/~help`

[26] Computing Research Associates Grand Challenges in Trustworthy Computing,
`http://www.cra.org/Activities/grand.challenges/security/home.html` (2003).

[27] Debian,
`http://www.debian.com`

[28] Debian Security Information,
`http://www.debian.org/security/`

[29] Microsoft Security Bulletins,
`http://www.microsoft.com/technet/security/current.asp`

[30] Microsoft Security Bulletin MS02-005,
http://www.microsoft.com/technet/treeview/default.asp?url=/technet/security/bulletin/MS02-005.asp (2002).

[31] MITRE CVEs,
http://www.cve.mitre.org

[32] Red Hat,
http://www.readhat.com

[33] Red Hat Security advisories,
http://www.redhat.com/support/errata/index.html

Microsoft Security Bulletin MS02-005

To show the generality of our formal model and specification approach, we describe an attack sequence exploiting a Windows vulnerability. It is one of the vulnerabilities described in the February 11, 2002 Microsoft Security Bulletin MS02-005 [30]. The processing of an HTML document having an embedded object by the MSHTML parser involves a buffer overrun. The adversary can exploit this vulnerability to execute arbitrary code in the security context of the user.

In the specifications of the actions below, we assume the type *web_server* has a function *make_doc: web_server* $\times$ *emb_obj* $\rightarrow$ *web_page* to create a web page with an embedded object and a function *add_page: web_server* $\times$ *web_page* $\rightarrow$ *web_server* to add a page to the web server. The type *web_page* has a function *get_obj: web_page* $\rightarrow$ *emb_obj* which returns the object embedded in the web page. The type *emb_obj* has a function *length: emb_obj* $\rightarrow$ *int* that returns the length of the embedded object. The type *browser* has a function *dload: browser* $\times$ *web_server* $\rightarrow$ *web_page* to download a web page from a web server. It also has a function *get_security_zone: browser* $\times$ *web_server* $\rightarrow$ *security_zone* to get the security zone to which the web server is mapped on the system. The type *security_zone* has a function *r_activex: security_zone* $\rightarrow$ *boolean* that returns true if the user has enabled the option to run ActiveX controls in the security zone; false, otherwise. The type *parser* has a function *display: parser* $\times$ *emb_obj* $\rightarrow$ *unit* to display the embedded object and a function *x_load: parser* $\times$ *emb_obj* $\rightarrow$ *executable* to extract the payload from the embedded object.

action CREATE_DOCUMENT$_T$(W: web_server, X: emb_obj)
pre *true*
post $W' = add\_page(W, make\_doc(W, X))$

action DOWNLOAD$_U$(W: web_server,
B: browser): D: web_page
pre *true*
post $D = dload(B, W) \wedge \bar{D} \in e'$

action PARSE$_S$(M: parser, P: web_page,
Z: security_zone)
pre *true*
post $\exists X \;.\; X = get\_obj(P) \wedge r\_activex(Z) \Rightarrow$
$[(length(X) \leq 512 \Rightarrow display(M, X)) \wedge$
$(length(X) > 512 \Rightarrow$
$\exists E \;.\; (E = x\_load(M, X) \Rightarrow$
$E.pre \Rightarrow E.post))]$

The effect of executing CREATE_DOCUMENT$_T$ is to create a web page with an embedded object on the web server. The effect of DOWNLOAD$_U$ is to download a web page from a web server. The effect of PARSE$_S$ is to display the embedded object in the web page being parsed if the length of the embedded object is ≤ 512 and execute the object's extracted payload, otherwise.

Now we give an example of attack on the System exploiting the buffer overrun. We assume that the user has mapped the web server WS to security zone Z on the system, and has enabled the option to run ActiveX controls in zone Z. Informally, the adversary's goal is to execute some arbitrary code, *E: executable*, in the system; formally, we represent this Goal as the predicate *E.pre* $\Rightarrow$ *E.post*. The attack on the System consists of the sequence of three action executions.

$$\{\exists\overline{WS} : web\_server \in e \wedge \exists\overline{IE} : browser \in e \wedge \exists\overline{MSHTML} : parser \in e \wedge$$
$$\exists\overline{Z} : security\_zone \in e \wedge \exists\overline{X} : emb\_obj \in e \wedge length(X) > 512 \wedge$$
$$Z = get\_security\_zone(IE, WS) \wedge r\_activex(Z)\}$$
$$\text{CREATE\_DOCUMENT}_T(\text{WS, X})$$
$$\text{D} = \text{DOWNLOAD}_U(\text{WS, IE})$$
$$\text{PARSE}_S(\text{MSHTML, D, Z})$$
$$\{\exists E : executable\ .\ E.pre \Rightarrow E.post\}$$

Since the length of the object, X, embedded in the web page is greater than 512, the second conjunct of the post-condition of PARSE_S holds; thus Goal of the adversary is achieved at the end of the attack.