Measuring Relative Attack Surfaces

Michael Howard[1] Jon Pincus[2] Jeannette M. Wing[3]

August 2003
CMU-CS-03-169

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

We propose a metric for determining whether one version of a system is more secure than another with respect to a fixed set of dimensions. Rather than count bugs at the code level or count vulnerability reports at the system level, we count a system's *attack opportunities*. We use this count as an indication of the system's "attackability," likelihood that it will be successfully attacked. We describe a system's *attack surface* along three abstract dimensions: targets and enablers, channels and protocols, and access rights. Intuitively, the more exposed the system's surface, the more attack opportunities, and hence the more likely it will be a target of attack. Thus, one way to improve system security is to reduce its attack surface.

To validate our ideas, we recast Microsoft Security Bulletin MS02-005 using our terminology, and we show how Howard's Relative Attack Surface Quotient for Windows is an instance of our general metric.

[1]Windows Security Management, Microsoft Corporation, Redmond, WA 98052.
[2]Microsoft Research, Microsoft Corporation, Redmond, WA 98052.
[3]Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213. Work done while on sabbatical at Microsoft Research.

Keywords: Security metrics, attacks, vulnerabilities, attack surface, threat modeling.

1 Introduction

Given that security is not an either-or property, how can we determine that a new release of a system is "more secure" than an earlier version? What metrics should we use and what things should we count? Our work argues that rather than attempt to measure the security of a system in absolute terms with respect to a yardstick, a more useful approach is to measure its "relative" security. We use "relative" in the following sense: Given System A, we compare its security *relative to* System B, and we do this comparison with respect to a given number of yardsticks, which we call *dimensions*. So rather than say "System A is secure" or "System A has a measured security number N" we say "System A is more secure than System B with respect to a fixed set of dimensions."

In what follows, we assume that System A and System B have the same operating environment. That is, the set of assumptions about the environment in which System A and System B is deployed is the same; in particular, the threat models for System A and System B are the same. Thus, it helps to think of System A and System B as different versions of the same system.

1.1 Motivation

Our work is motivated by the practical problem faced in industry today. Industry has responded to demands for improvement in software and systems security by increasing effort[1] into creating "more secure" products and services. How can industry determine if this effort is paying off and how can we as consumers determine if industry's effort has made a difference?

Our approach to measuring relative security between systems is inspired by Howard's informal notion of *relative attack surface* [How03]. Howard identified 17 "attack vectors," i.e., likely opportunities of attack. Examples of his attack vectors are open sockets, weak ACLs, dynamic web pages, and enabled guest accounts. Based on these 17 attack vectors, he computes a "measure" of the attack surface, which he calls the Relative Attack Surface Quotient (RASQ), for seven running versions of Windows.

We added three attack vectors to Howard's original 17 and show the RASQ calcuation for five versions of Windows in Figure 1. The bar chart suggests that a default running version of Windows Server 2003 is much more secure than previous versions with respect to the 20 attack vectors. It also illustrates that the attack surface of Windows Server 2003 increases only marginally when IIS is enabled—in sharp contrast to Windows NT 4.0, where enabling IIS (by installing the "Option Pack") dramatically increased the RASQ, and to Windows 2000, where IIS is enabled by default[2]. As will be discussed in Section 6.3, these differences in RASQ are consistent with anecdotal evidence for the relative security of different Windows platforms and configurations.

1.2 A New Metric: Attackability

Two measurements are often used to determine the security of a system: at the code level, a count of the number of bugs found (or fixed from one version to the next); and at the system level, a count of the number of times a system (or any of its versions) is mentioned in the set of Common Vulnerabilities and Exposures (CVE) bulletins [MIT], CERT advisories [CER], etc.

Rather than measure code-level or system-level vulnerability, we consider a different measure, somewhat in between, which we call *attack opportunity*, or "attackability" for short. Counting the number of bugs found (or fixed) misses bugs that are not found (or fixed), perhaps the very one that

[1] For example, Microsoft's *Trustworthy Computing Initiative*, started in January 2002.

[2] NT 4.0 measurements were taken on a system where Service Pack 6a had been installed; NT 4.0 with IIS enabled, with both Service Pack 6a and the NT 4.0 Option Pack installed. IIS stands for Internet Information Server.

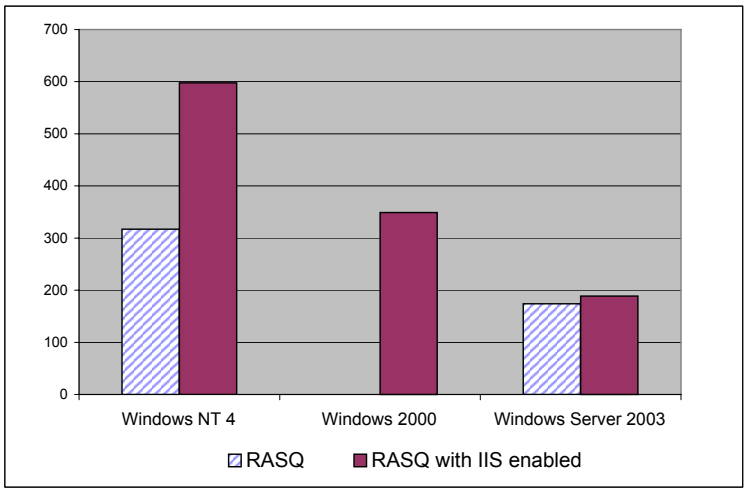

Figure 1: Relative Attack Surface Quotient of Different Versions of Windows [How03]

is exploited; it treats all bugs alike when one might be easier to exploit than another, or the exploit of one may result in more damage than the exploit of another. Instead, we want a measure—at a higher abstraction level—that gives more weight to bugs that are more likely to be exploited. Counting the number of times a system version appears in bulletins and advisories ignores the specifics of the system configuration that give rise to the exploit: whether a security patch has been installed, whether defaults are turned off, whether it always runs in system administrator mode. Instead, we want a measure—at a lower abstraction level—that allows us to refer to very specific states (i.e., configurations) of a system. Given this intermediate viewpoint, we propose that there are certain system features that are more likely than others to be opportunities of attack. The counts of these "more likely to be attacked" system features determine a system's attackability.

Further, we will categorize these attack opportunities into different abstract *dimensions*, which together define a system's *attack surface*. Intuitively, the more exposed the system's surface, the more attack opportunities, and hence the more likely it will be a target of attack. Thus, one way to improve system security is to reduce its attack surface.

Suppose now we are given a fixed set of dimensions and a fixed set of attack opportunities (i.e., system features) for each dimension. Then with respect to this fixed set of dimensions of attack opportunities, we can measure whether System A is "more secure" than System B.

In our work, we use state machines to model Systems A and B. Our abstract model allows Systems A and B to be any two state machines, each of which interacts with the same state machine model of its environment, i.e., threat model. In practice, it is more useful and more meaningful to compare two systems that have some close relationship, e.g., they provide similar functionality, perhaps through similar APIs, rather than two arbitrary systems. The abstract dimensions along which we compare two systems are derived directly from our state machine model: *process and data resources* and the *actions* that we can execute on these resources. For a given attack, which we define to be a sequence of action executions, we distinguish *targets* from *enablers*: *targets* are processes or data resources that an adversary aims to control, and *enablers* are all other processes and data resources that are used by the adversary to carry out the attack successfully. The adversary obtains control over these resources through communication *channels* and *protocols*. Control is subject to the constraints imposed by a system's *set of access rights*. In summary, our attack surface's three

dimensions are: targets and enablers, channels and protocols, and access rights. Attackability is a measure of how exposed a system's attack surface is.

1.3 Contributions and Roadmap

We use a state machine formal framework to support three main contributions of this paper:

- The notion of a system's *attack surface.*
- A new *relative measure of security*, attackability.
- A model for vulnerabilities as differences between intended and actual behavior, in terms of pre-conditions and post-conditions (Section 2.2).

Our "relative" approach has the advantage that security analysts are more willing and able to give relative rankings of threats and relative values to risk-mitigation controls, than absolute numbers [But03]. We also avoid the need to assign probabilities to attacks.

We view our work as only a first step toward coming up with a meaningful, yet practical way of measuring (relative) security. By no means do we claim to have identified "the right" or "all" the dimensions of an attack surface. Indeed, our use of the word "dimensions" is only meant to be suggestive of a surface; our dimensions are not orthogonal. We hope with this paper to spark a fruitful line of new research in security metrics.

In Section 2 we present our formal framework and then in Section 3 we explain our abstract dimensions of a system's attack surface. To illustrate these ideas concretely, in Section 4 we recast Microsoft Security Bulletin MS02-005 in terms of our concepts of targets and enablers. In Section 5 we give an abstract attack surface measurement function. Again, to be concrete, in Section 6 we revisit Howard's RASQ metric in terms of our abstract dimensions. In Section 7 we discuss how best to apply and not to apply the RASQ approach. We close with a review of related work in Section 8 and suggestions for future work in Section 9.

2 Terminology and Model

Our formal model is guided by the following three terms from *Trust in Cyberspace* [Sch91]:

- A *vulnerability* is an error or weakness in design, implementation, or operation.
- An *attack* is the means of exploiting a vulnerability.
- A *threat* is an adversary motivated and capable of exploiting a vulnerability.

We model both the system and the threat as state machines, which we will call *System* and *Threat*, respectively. A state machine has a set of states, a set of initial states, a set of actions, and a state transition relation. We model an attack as a sequence of executions of actions that ends in a state that satisfies the adversary's goal, and in which one or more of the actions executed in an attack involves a vulnerability.

2.1 State Machines

A *state machine*, $M = \langle S, I, A, T \rangle$, is a four-tuple where S is a set of states, $I \subseteq S$ is a set of initial states, A is a set of actions, and $T = S \times A \times S$ is a transition relation. A state $s \in S$ is a mapping from typed *resources* to their typed *values*:

$$s : Res_M \rightarrow Val_M$$

Of interest to us are state resources that are *processes* and *data*. A *state transition*, $\langle s, a, s' \rangle$, is the execution of action a in state s resulting in state s'. A change in state means that either a new resource is added to the mapping, a resource was deleted, or a resource changes in value. We assume each state transition is atomic.

An *execution* of a state machine is the alternating sequence of states and action executions:

$$s_0 \ a_1 \ s_1 \ a_2 \ s_2 \ \ldots \ s_{i-1} \ a_i \ s_i \ \ldots$$

where $s_0 \in I$ and $\forall i > 0. \langle s_{i-1}, a_i, s_i \rangle \in T$. An execution can be finite or infinite. If finite, it ends in a state.

The *behavior* of a state machine, M, is the set of all its executions. We denote this set *Beh(M)*. A state s is *reachable* if either $s \in I$ or there is an execution, $e \in Beh(M)$, such that s appears in e.

We will assume that actions are specified by pre- and post-conditions. For an action, $a \in A$, if $a.pre$ and $a.post$ denote a's pre- and post-condition specifications, we can then define the subset of the transition relation, T, that involves only action a as follows:

$$a.T = \{\langle s, a, s' \rangle : S \times A \times S \mid a.pre(s) \Rightarrow a.post(s, s')\}$$

We model both the system under attack and the threat (adversary) as state machines:

$$System = \langle S_S, I_S, A_S, T_S \rangle$$
$$Threat = \langle S_T, I_T, A_T, T_T \rangle$$

We partition the resources of a state machine, M, into a set of *local resources* and a set of *global resources*, $Res_M = Res_M^L \uplus Res_M^G$. We define the *combination* of the two state machines, $ST = System \bowtie Threat$, by merging all the corresponding components[3]:

- $S_{ST} \subseteq 2^{Res_{ST} \rightarrow Val_{ST}}$
- $I_{ST} = I_S \cup I_T$
- $A_{ST} = A_S \cup A_T$
- $T_{ST} = T_S \cup T_T$

We identify the global resources of S and the global resources of T such that $Res_{ST}^G = Res_S^G = Res_T^G$ and so $Res_{ST} = Res_S^L \uplus Res_T^L \uplus Res_{ST}^G$. Finally, $Val_{ST} = Val_S \cup Val_T$. We extend the definitions of executions, behaviors, etc. in the standard way.

An adversary targets a system under attack to accomplish a goal:

$$System\text{-}Under\text{-}Attack = (System \bowtie Threat) \times Goal$$

[3]There are more elegant formulations of composing two state machines; we use a simple-minded approach that basically merges two state machines into one big one. In the extreme, if the local resources sets are empty, then the two machines share all state resources; if the global resource set is empty, they share nothing. Thus our model is flexible enough to allow communication through only shared memory, only message passing, or a combination of the two.

where *Goal* is formulated as a predicate over states in S_{ST}. Note that we make explicit the goal of the adversary in our model of a system under attack. Example goals might be "Obtain root access on host H" or "Deface website on server S." In other contexts, such as fault-tolerant computing, *Threat* is synonymous with the system's "environment." Thus, we use *Threat* to model environmental failures, due to benign or malicious actions, that affect a system's state.

Intuitively, the way to reduce the attack surface is to ensure that the behavior of *System* prohibits *Threat* from achieving its *Goal*.

2.2 Vulnerabilities

Vulnerabilities can be found at different levels of a system: implementation, design, operational, etc. They all share the common intuition that something in the actual behavior of the system deviates from the intended behavior. We can capture this intuition more formally by comparing the difference between the behaviors of two state machines. Suppose there is a state machine that models the intended behavior, and one that models the actual behavior:

$$Intend = \langle S_{Int}, I_{Int}, A_{Int}, T_{Int} \rangle$$
$$Actual = \langle S_{Act}, I_{Act}, A_{Act}, T_{Act} \rangle$$

We define the vulnerability difference set, *Vul*, to be the difference in behaviors of the two machines:

$$Vul = Beh(Actual) - Beh(Intend)$$

An execution sequence in *Vul* arises from one or more differences between some component of the state machine *Actual* and the corresponding component of *Intend*, i.e., differences between the corresponding sets of (1) states (or more relevantly, reachable states), (2) initial states, (3) actions, or (4) transition relations. We refer to any one of these kinds of differences as a *vulnerability*. Let's consider each of these cases:

1. $S_{Act} - S_{Int} \neq \varnothing$

   If there is a difference in state sets then there are some states that are defined for *Actual* that are not intended to be defined for *Intend*. The difference may be due to (1) a resource that is in a state in *Actual*, but not in *Intend* or (2) a value allowed for resource in *Actual* that is not allowed for that resource in *Intend*. (A resource that is not in a state in *Actual*, but is in *Intend* is ok.) The difference may not be too serious if the states in the difference are not reachable by some transition in T_{Act}. If they are reachable, then the difference in transition relations will pick up on this vulnerability. However, even if they are not reachable, it means that if any of the specifications for actions changes in the future, we must be careful to make sure that the set of reachable states in *Actual* is a subset of that of *Intend*.

2. $I_{Act} - I_{Int} \neq \varnothing$

   If there is a difference in initial state sets then there is at least one state in which we can start an execution when we ought not to. This situation can arise if resources are not initialized when they should be, they are given incorrect initial values, or when there are resources in an initial actual state but not in any initial intended state.

3. $A_{Act} - A_{Int} \neq \varnothing$

   If there is a difference in action sets then there are some actions that can be actually done that are not intended. These actions will surely lead to unexpected behavior. The difference will show up in the differences in the state transition relations (see below).

4. $T_{Act} - T_{Int} \neq \varnothing$

If there is a difference in state transition sets then there is at least one state transition allowed in *Actual* that should not be allowed according to *Intend*. This situation can arise because either (i) the action sets are different or (ii) the pre-/post-conditions for an action common to both action sets are different.

More precisely, for case (ii) where $A_{Act} = A_{Int}$, consider a given action $a \in A_{Int}$. If $a.T_{Act} - a.T_{Int}$ is non-empty then there are some states either in which we can execute a in *Actual* and not in *Intend* or which we can reach as a result of executing a in *Actual* and not in *Intend*. Let $a_{Act}.pre$ and $a_{Int}.pre$ be the pre-conditions for a in *Actual* and *Intend*, respectively, and similarly for their post-conditions. In terms of pre- and post-conditions, *no difference* can arise if

- $a_{Act}.pre \Rightarrow a_{Int}.pre$ and
- $a_{Act}.post \Rightarrow a_{Int}.post$.

Intuitively, if the "actual" behavior is stronger than the "intended" then we are safe.

Given that *Actual* models the actual behavior of the system, then our system combined with the *Threat* machine looks like:

$$\textit{System-Under-Attack} = (\textit{Actual} \bowtie \textit{Threat}) \times \textit{Goal}$$

as opposed to

$$\textit{System-Under-Attack} = (\textit{Intend} \bowtie \textit{Threat}) \times \textit{Goal}$$

again with the expectation that were *Intend* implemented correctly, *Goal* would not be achievable.

In this paper we focus our attention at implementation-level vulnerabilities, in particular, differences that can be blamed on an action's pre-condition or post-condition that is too weak or incorrect. A typical example is in handling a buffer overrun. Here is the intended behavior, for a given input string, s:

$$\text{length(s)} \leq 512 \Rightarrow \text{"process normally"} \wedge \text{length(s)} > 512 \Rightarrow \text{"report error and terminate"}$$

If the programmer forgot to check the length of the input, the actual behavior might instead be

$$\text{length(s)} \leq 512 \Rightarrow \text{"process normally"} \wedge \text{length(s)} > 512 \Rightarrow \text{"execute extracted payload"}$$

Here "execute extracted payload" presumably has an observable unintended side effect that differs from just reporting an error.

2.3 Attacks

An attack is the "means of exploiting a vulnerability" [Sch91]. We model an attack to be a sequence of action executions, at least one of which involves a vulnerability. More precisely, an attack, k, either starts in an unintended initial state or reaches an unintended state through one of the actions executed in k. In general, an attack will include the execution of actions from both state machines, *System* and *Threat*.

The difference between an arbitrary sequence of action executions and an attack is that an attack includes either (or both) (1) the execution of an action whose behavior deviates from the intended (see previous section) or (2) the execution of an action, $a \in A_{Act} - A_{Int} (\neq \varnothing)$. In this second case, the set of unintended behaviors will include behaviors not in the set of intended behaviors since $A_{Act} \neq A_{Int}$.

For a given attack, k, the *means of an attack* is the set of all actions in k and the set of all process and data resources accessed in performing each action in k. These resources include all global and local resources accessed by each action in k and all parameters passed in as arguments or returned as a result to each action executed in k.

3 Dimensions of an Attack Surface

We consider three broad dimensions to our attack surface:

- *Targets and enablers.* To achieve his goal, the adversary has in mind one or more targets on the system to attack. An *attack target*, or simply *target*, is a distinguished process or data resource on *System* that plays a critical role in the adversary's achieving his goal. We use the term *enabler* for any accessed process or data resource that is used as part of the means of the attack but is not singled out to be a target.

- *Channels and protocols.* *Communication channels* are the means by which the adversary gains access to the targets on *System*. We allow both message-passing and shared-memory channels. *Protocols* determine the rules of interaction among the parties communicating on a channel.

- *Access rights.* These rights are associated with each process and data resource of a state machine.

Intuitively, the more targets, the larger the attack surface. The more channels, the larger the attack surface. The more generous the access rights, the larger the attack surface.

We now look at each of these dimensions in turn.

3.1 Targets and Enablers

Targets and enablers are resources that an attacker can use or coopt. There are two kinds: processes and data. Since it is a matter of the adversary's goal that determines whether a resource is a target or enabler, for the remainder of this section we use the term targets to stand for both. In particular, a target in one attack might simply be an enabler for a different attack, and vice versa.

Examples of process targets are browsers, mailers, and database servers. Examples of data targets are files, directories, registries, and access rights.

The adversary wants to control the target: modify it, gain access to it, or destroy it. Control means more than ownership; more generally, the adversary can use it, e.g., to trigger the next step in the attack. Consider a typical worm or virus attack, which follows this general pattern:

Step 1: Ship an executable—treated as a piece of data—within a carrier to a target machine.

Step 2: Use an enabler, e.g., a browser, to extract the payload (the executable) from the carrier.

Step 3: Get an interpreter to execute the executable to cause a state change on the target machine.

where the attacker's goal, achieved after the third step, may be to modify state on the target machine, to use up its resources, or to set it up for further attacks.

The prevalence of this type of attack leads us to name two special types of data resources. First, *executables* is a distinguished type of data resource in that they can be interpreted (i.e., evaluated). We associate with executables one or more eval functions, *eval: executable* $\rightarrow$ *unit*.[4] Different *eval* functions might interpret the same executable with differing effects. Executables can be targets and controlling such a target includes the ability to call an *eval* function on it. The adversary would do so, for example, for the side effect of establishing the pre-condition of the next step in the attack.

[4] Writing the return type of *eval* as *unit* is our way, borrowed from ML, to indicate that a function has a side effect.

Obvious example types of *eval* functions include browsers, mailers, applications, and services (e.g., Web servers, databases, scripting engines). Less obvious examples include application extensions (e.g., Web handlers, add-on dll's, ActiveX controls, ISAPI filters, device drivers), which run in the same process as the application; and helper applications (e.g., CGI scripts), which run in a separate process from the application.

Carriers are our second distinguished type of data resource. Executables are embedded in carriers. Specifically, carriers have a function *extract_payload: carrier* $\rightarrow$ *executable*. Examples of carriers include viruses, worms, Trojan horses, and email messages.

Part of calculating the attack surface is determining the types and numbers of instances of potential process targets and data targets, the types and numbers of instances of eval functions for executables that could have potentially damaging side effects; and the types and numbers of instances of carriers for any executable.

3.2 Channels and Protocols

A channel is a means of communicating information from a sender to a receiver (e.g., from an attacker to a target machine). We consider two kinds of channels: message-passing (e.g., sockets, RPC connections, and named pipes) and shared-memory (e.g., files, directories, and registries). Channel "endpoints" are processes.

Associated with each kind of channel is a *protocol*, the rules of exchanging information. For message-passing channels, example protocols include ftp, RPC, http, and streaming. For shared-memory, examples include protocols that might govern the order of operations (e.g., a file has to be open before read), constrain simultaneous access (e.g., multiple-reader/single-writer or single-reader/single-writer), or prescribe locking rules (e.g., acquire locks according to a given partial order).

Channels are data resources. A channel shared between *System* and *Threat* machines is an element of Res^G_{ST} in the combination of the two machines. In practice, in an attack sequence, the *Threat* machine might establish a new message-passing channel, e.g., after scanning host machines to find out what services are running on port 80.

Part of calculating the attack surface is determining the types of channels, the numbers of instances of each channel type, the types of protocols allowed per channel type, the numbers and types of processes at the channel endpoints, the access rights (see below) associated with the channels and their endpoints, etc.

3.3 Access rights

We associate *access rights* with all resources. For example, for data that are text files, we might associate read and write rights; for executables, we might associate execute rights. Note that we associate rights not only with files and directories, but also with channels (since they are data resources) and channel endpoints (since they are running processes).

Conceptually we model these rights as a relation, suggestive of Lampson's orginal access control matrix [Lam74]:

$$Access \subseteq Principals \times Res \times Rights$$

where $Principals = Users \cup Processes$, $Res = Processes \cup Data$, and *Rights* is left uninterpreted. (*Res* is the same set of resources introduced in Section 2.) For example, in Unix, *Rights* = {read, write, execute}, in the Andrew file system, *Rights* = {read, lookup, insert, delete, write, lock,

administer}, and in Windows there are eighteen different rights associated with files and directories alone; and of course not all rights are appropriate for all principals or resources. More generally, to represent conditional access rights, we can extend the above relation with a fourth dimension, $Access \subseteq Principals \times Res \times Rights \times Conditions$, where *Conditions* is a set of state predicates.

There are shorthands for some "interesting" subsets of the *Access* relation, e.g., accounts, trust relationships, and privilege levels, that we usually implement in practice, in lieu of representing the *Access* relation as a matrix.

- *Accounts* represent principals, i.e., users and processes. Thus, we view an account as shorthand for a particular principal with a particular set of access rights. Accounts can be data or process targets.

  There are some special accounts that have default access rights. Examples are well-known accounts such as guest accounts, and accounts with "admin" privileges. These typically have names that are easy to guess.

  Part of calculating the attack surface is determining the number of accounts, the number of accounts with admin privileges, and the existence and number of guest accounts, etc. Also, part of calculating the attack surface is determining for each account if the tightest access rights possible are associated with it.

- A *trust relationship* is just a shorthand for an expanded access rights matrix. For example, we might define a specific trust relation, $Tr \subseteq Principals \times Principals$, where network hosts might be a subset of *Principals*. Then we might define the access rights for principal p_1 to be the same as or a subset of those for principal p_2 if $Tr(p_1, p_2)$. We could do something similar to represent the "speaks for" relation of Lampson, Abadi, Burrows, and Wobber [LABW92]. In both cases, by modeling access rights as a (flat) ternary relation, however, we lose some information: the structural relationship between the two principals (A trusts B or A speaks for B). We choose, however, to stick to the simpler access rights matrix model because of its prevalence in use.

- *Privilege levels* map a principal to a level in a total or partial order, e.g., $none < user < root$. Associated with a given level is a set of access rights. Suppose we have a function, *priv_level*: $Principals \rightarrow$ {none, user, root}, then the rights of principal p would be those associated with *priv_level(p)*.

Reducing the attack surface with respect to access rights is a special case of abiding by the Principle of Least Privilege: Grant only the relevant rights to each of the principals who are allowed access to a given resource.

4 Security Bulletins

To validate our general attack surface model, we described a dozen Microsoft Security bulletins [Mic] using our terminology [PW03]. The one example we present here illustrates how two different attacks can exploit the same vulnerability via different channels.

The Microsoft Security Bulletin MS02-005, posted February 11, 2002, reports six vulnerabilities and a cumulative patch to fix all of them. We explain just the first (see Figures 2 and 3). The problem is that the processing of an HTML document (a web page sent back from a server or HTML email) that embeds another object involves a buffer overrun. Exploiting this buffer overrun vulnerability lets the adversary run arbitrary code in the security context of the user.

We now walk through the template which we use for describing these bulletins.

First we specify the vulnerability as the difference in actual from intended behavior for an action. Here the action is the processing by MSHTML (the HTML renderer on Microsoft Windows 2000 and Windows XP) of an HTML document D in a security zone Z. The intended pre-condition is "true," i.e., this action should be allowed in all possible states. However, due to a missing validation check of the action's input, the actual pre-condition is that the length of the object, X, embedded in D, should be less than or equal to 512 bytes.

The intended post-condition is to display the embedded object as long as the ability to run ActiveX Controls is enabled for zone Z. The actual post-condition, due to the non-trivial pre-condition, is that if the length of X is longer than 512 bytes, then the executable E extracted from X is evaluated for its effects. By referring to the pre- and post-conditions of E, i.e., E.pre and E.post, we capture E's effects as if it were evaluated; this makes sense only for a resource that is an executable, and thus has an *eval* function defined for it. Note that most executables, when evaluated, will simply crash the MSHTML process.

After describing the vulnerability, we give a series of sample attacks, each of which shows how the vulnerability can be exploited by the adversary. Before giving some sample attacks for MS02-005a, we explain the parts in our template that we use to describe each attack.

- The goal of the attack.
- A resource table showing for each resource (data or process) involved in the attack whether it serves as a carrier ("Y" means "yes; a blank, "no"), a channel (if so, "MP" means message-passing; "SM" means shared-memory; and a blank means it is not a channel), or a target or enabler ("T" means it is a target; "E", an enabler).
- The pre-condition for the attack. Each clause is a conjunct of the pre-condition.
- The attack itself, written as a sequence of actions. The action exploiting the vulnerability is in boldface. (More formally, we would specify each action with pre- and post-conditions. For the attack to make sense, the pre-condition of the attack should imply the pre-condition of the first action in the attack, the post-condition of the ith action should imply the pre-condition of the $i+1$st action, and the post-condition of the last action should imply the post-condition of the attack.)
- The post-condition for the attack. This post-condition corresponds to the adversary's goal, i.e., the reason for launching the attack in the first place. It should imply the goal (see first item above).

Let's now return to our example. Since MSHTML is used by both the browser and the mailer, we give two sample attacks, each exploiting the same vulnerability just described.

In the first attack (Figure 2), the adversary's goal is to run arbitrary code on the client. As indicated by the resource table for Attack 1, he accomplishes his goal by using the web server and the client browser as enablers. The server-client web connection is the message-passing channel by which the attack occurs. The HTML document is the carrier of the payload and the MSHTML process is the target of attack.

The pre-condition for the attack is that the victim should have requested a web page from the adversary and should have enabled for zone Z the option to run ActiveX Controls, and that the adversary's site is mapped to zone Z on the victim's machine. The attack itself is the sequence of three actions: the web server sends an HTML document D with an ill-formed embedded object to

Action Vulnerability: MSHTML processes HTML document D in zone Z.

Intended precondition: true
Actual precondition: D contains `<EMBED SRC=X>` $\Rightarrow$ length(X) $\leq$ 512
Intended postcondition: (one of many clauses)
D contains `<EMBED SRC=X>` $\wedge$ "Run ActiveX Controls" is enabled for Z $\Rightarrow$ display(X)
Actual postcondition: (one of many clauses)
D contains `<EMBED SRC=X>` $\wedge$ "Run ActiveX Controls" is enabled for Z $\Rightarrow$
[(length(X) $>$ 512 $\wedge$ extract_payload(X) = E) $\Rightarrow$ (E.pre $\Rightarrow$ E.post)
$\wedge$ length(X) $\leq$ 512 $\Rightarrow$ display(X)]

Attack 1: Web server executes arbitrary code on client.
Goal: Enable execution of arbitrary code on client.

Resource Table

Resource	*Carrier*	*Channel*	*Target/Enabler*
HTTPD web server (process)			E
server-client web connection C (data)		MP	E
browser B (process)			E
HTML document D (data)	Y		E
MSHTML (process)			T

Preconditions

- Victim requests a web page from adversary's site S.
- Victim's machine maps site S to zone Z.
- Victim's machine has "Run ActiveX Controls" security option enabled for zone Z.
- Adversary creates HTML document D containing an embed tag `<EMBED X>`, where length(X) $>$ 512 and extract_payload(X) = E.

Attack Sequence

1. Web server sends document D to browser B over connection C.
2. B passes D to MSHTML in zone Z.
3. **MSHTML processes D in zone Z.**

Postconditions

- Arbitrary, depending on the payload.

Figure 2: Microsoft Security Bulletin MS02-005a: Cumulative Patch for Internet Explorer (I)

Attack 2: Mail-based attack (HTML email) executing arbitrary code on client.
Goal: Enable execution of arbitrary code on client.

Resource Table

Resource	*Carrier*	*Channel*	*Target/Enabler*
HTTPD web server (process)			E
server-client mail connection C (data)		MP	E
Outlook Express OE (process)			E
HTML mail message M (data)			E
HTML document D (data)	Y		E
MSHTML (process)			T

Preconditions

- Victim able to receive mail from attacker.
- Victim's HTML email is received in zone Z.
- Victim's machine has "Run ActiveX Controls" security option enabled for zone Z.
- Adversary creates HTML document D containing an embed tag `<EMBED X>`, where length(X) $>$ 512 and extract_payload(X) = E.
- Adversary creates mail message M with D included, where Z $\neq$ Restricted Zone.

Attack Sequence

1. Adversary sends HTML message M to victim via email.
2. Victim views (or previews) M in OE.
3. OE passes D to MSHTML in zone Z .
4. **MSHTML processes D in zone Z.**

Postconditions

- Arbitrary, depending on the payload.

Figure 3: Microsoft Security Bulletin MS02-005a: Cumulative Patch for Internet Explorer (II)

the client browser; the browser passes D to the MSHTML process; the MSHTML processes D as specified in the vulnerability. The post-condition of the attack is the effect of running the embedded executable.

In the second attack (Figure 3), the adversary's goal is the same and the vulnerability is the same. The means of attack, however, are different. Here, the enablers are an HTML mail document and the mailer process, i.e., Outlook Express. Note that people usually consider Outlook Express to be the target, but in fact, for this attack, it is an enabler. The channel, carrier, and target are the same as for the first attack.

The pre-condition is different: the victim needs to be able to receive mail from the attacker and HTML email received is in zone Z that is not the restricted zone. The attack is a sequence of four actions: the web server sends an HTML document D with an ill-formed embedded object to the victim via email; the victim views the HTML document in the mailer process, i.e., Outlook Express; the mailer process sends D to MSHTML in zone Z; and finally, the MSHTML processes D as specified in the vulnerability. The post-condition is as for the first attack, i.e., the effect of running the embedded executable.

5 Analyzing Attack Surfaces

We use our broad dimensions of targets and enablers, communication channels and protocols, and access rights to guide us in deciding (1) what things to count, to determine a system's attackability; (2) what things to eliminate or reduce, to improve system security; and (3) how to compare two versions of the same system. In this section we consider briefly the first two items; Section 6 gives a detailed concrete example of all three.

5.1 Measuring the Attack Surface

We can define a measure of the system's attack surface to be some function of the targets and enablers, the channels associated with each type or instance of a target and enabler, the protocols that constrain the use of channels, and the access rights that constrain the access to all resources.

$$surf = f(targets, enablers, channels, protocols, access\ \ rights)$$

In general, we can define the function f in terms of additional functions on targets, enablers, channels, and access rights to represent relationships between these (e.g., the constraints imposed by protocols on channels, and the constraints imposed by access rights on all resources), or weights of each type (e.g., to reflect that certain types of targets are more critical than others or to reflect that certain instances of channels are less critical than others).

We deliberately leave f uninterpreted because in practice what a security analyst may want to measure may differ from system to system. Moreover, defining a precise f in general, even for a given system, can be extremely difficult. We leave the investigation of what different types of metrics are appropriate for f for future work. In Section 6 we give a very simplistic f.

5.2 Reducing the Attack Surface

The concepts underlying our attack surface also give us a systematic way to think about how to reduce it. We can eliminate or reduce the number of (1) types or instances of targets, processes, enablers, executables, carriers, eval functions, channels, protocols, and rights; (2) types or instances

of vulnerabilities, e.g., by strengthening the actual pre- or post-condition to match the intended; or (3) types or instances of attacks, e.g., through deploying one or more security technologies.

Principles and rules of thumb that system administrators and software developers follow in making their systems more secure correspond naturally to our concepts. For example, the tasks specified in "lockdown instructions" for improving security of a system frequently include eliminating data and process targets and strengthening access rights. Consider these examples:

Colloquial	Formal
Turn off macros.	Eliminate an eval function for one type of data.
Block attachments in Outlook.	Avoid giving any executable (data) as an argument to an eval function.
Secure by default.	Eliminate entire types of targets, enablers, and channels; restrict access rights.
Check for buffer overrun.	Strengthen the post-condition of the actual behavior to match that of the intended behavior.
Validate your input.	Strengthen the pre-condition of the actual behavior to match that of the intended behavior.
Change your password every 90 days.	Increase the likelihood that the authentication mechanism's pre-condition is satisfied.

6 An Example Attack Surface Metric

Howard identified a set of 17 RASQ vectors [How03] and defined a simple attack surface function to determine the relative attack surface of seven different versions of Windows. In Section 6.1 we present 20 attack vectors: Howard's original 17 plus 3 others we added later. In Section 6.2 we present his RASQ calculation for all 20 attack vectors in detail. In Section 6.3 we analyze his RASQ results: we confirm observed behavior reflecting user experience and lockdown scenarios, but also we point out additional missing elements.

6.1 Attack Vectors for Windows

Howard's original 17 RASQ vectors [How03] are shown as the first 17 in Figure 4. Upon our[5] initial analysis of his work, we noted that he had not considered enablers, such as scripting engines. Thus, we subsequently added three more attack vectors, shown in italics. Figure 4 shows how we map the 20 attack vectors into our terminology of channels, process targets, data targets, process enablers, and access rights.

We describe each in more detail below.

1. Open sockets: TCP or UDP sockets on which at least one service is listening. Since one service can listen on multiple sockets and multiple services can listen on the same socket, this attack vector is a channel type; the number of channels is independent of the number of services.

2. Open RPC endpoints: Remotely-accessible handlers registered for remote procedure calls with the "endpoint manager." Again, a given service can register multiple handlers for different RPC interfaces.

3. Open named pipes: Remotely-accessible named pipes on which at least one service is listening.

[5] Pincus and Wing

4. Services: Services installed, but not disabled, on the machine. (These are equivalent to daemons on UNIX systems.)

5. Services running by default: Services actually running at the time the measurements are taken. Since our measurements are taken when the system first comes up, these are the services that are running by default at start-up time.

6. Services running as SYSTEM: Services configured to log on as LocalSystem (or System), as opposed to LocalService or some other user. (LocalSystem is in the administrators group.)

7. Active Web handlers: Web server components handling different protocols that are installed but not disabled (e.g., the W3C component handles http; the nntp component handles nntp).

8. Active ISAPI filters: Web server add-in components that filter particular kinds of requests. ISAPI stands for Internet Services Application Programming Interface; it enables developers to extend the functionality provided by a web server. An ISAPI filter is a dynamic link library (.dll) that uses ISAPI to respond to events that occur on the server.

9. Dynamic web pages: Files under the web server root other than static (.html) pages. Examples include .exe files, .asp (Active Server Pages) files, and .pl (Perl script) files.

10. Executable vdirs: "Virtual Directories" defined under the web server root that allow execution of scripts or executables stored in them.

11. Enabled accounts: Accounts defined in local users, excluding any disabled accounts.

12. Enabled accounts in admin group: Accounts in the administators group, excluding any disabled accounts.

13. Null sessions to pipes and shares: Whether pipes or "shares" (directories that can be shared by remote users) allow anonymous remote connections.

14. Guest account enabled: Whether there exists a special "guest" account and it is enabled.

15. Weak ACLs in FS: Files or directories that allow "full control" to everybody. "Full control" is the moral equivalent of UNIX rwxrwxrwx permissions.

16. Weak ACLs in Registry: Registry keys that allow "full control" to everybody.

17. Weak ACLS on shares: Directories that can be shared by remote users that allow "full control" to everybody. Even if one has not explicitly created any shares, there is a "default share" created for each drive; it should be protected so that others cannot get to it.

18. VBScript enabled: Whether applications, such as Internet Explorer and Outlook Express, are enabled to execute Visual Basic Script.

19. Jscript enabled: As for (18), except for Jscript.

20. ActiveX enabled: As for (18), except for ActiveX Controls.

20 RASQ Attack Vectors	Formal
Open sockets	channels
Open RPC endpoints	channels
Open named pipes	channels
Services	process targets
Services running by default	process targets, constrained by access rights
Services running as SYSTEM	process targets, constrained by access rights
Active Web handlers	process targets
Active ISAPI Filters	process targets
Dynamic Web pages	process targets
Executable vdirs	data targets
Enabled accounts	data targets
Enabled accounts in admin group	data targets, constrained by access rights
Null sessions to pipes and shares	channels
Guest account enabled	data targets, constrained by access rights
Weak ACLs in FS	data targets, constrained by access rights
Weak ACLs in Registry	data targets, constrained by access rights
Weak ACLs on shares	data targets, constrained by access rights
VBScript enabled	*process enabler*
Jscript enabled	*process enabler*
ActiveX enabled	*process enabler*

Figure 4: Mapping RASQ Attack Vectors into Our Formalism

6.2 Attack Surface Calculation

In Howard's calculation, the attack surface area is the sum of independent contributions from a set of channels types, a set of process target types, a set of data target types, a set of process enablers, all subject to the constraints of the access rights relation, A.

$$surf^A = surf^A_{ch} + surf^A_{pt} + surf^A_{dt} + surf^A_{pe}$$

This simple approach has a major advantage in that it allows the categories to be measured independently. This simplification comes at a cost. For example, since interactions between services and channels are not considered, Howard's RASQ calculation fails to distinguish between sockets opened by a service running as administrator and (less attackable) sockets opened by a service running as an arbitrary user.

Figure 5 gives a table showing each of the four terms in detail. Each term takes the form of a double summation: for each type (of channel types, *chty*, process target types, *ptty*, data target types, *dtty*), and process enabler types, *pety*, for each instance of that type, a *weight*, ω, for that instance is added to the total attack surface. For a given type, τ, we assume we can index the instances per type such that we can refer to the ith instance by τ_i. For *weight* functions, ω, that are conditional on the state of the instance (e.g., whether or not an account is default), we use the notation $(cond, v_1, v_2)$ where the value is v_1 if *cond* is true and v_2 if *cond* is false.

For channels, access control is factored into the weights in one very limited case: Howard gives a slightly lower weight to named pipes compared to the other channels because named pipes are not generally accessible over the Internet. An alternate, more general approach to modeling this

$surf^A_{ch} = \sum_{c \in chty} \sum_{i=1}^{\lvert c \rvert} \omega(c_i)$	
chty	$\omega(c_i)$
socket	1.0
endpoint	0.9
namedpipe	0.8
nullsession	0.9

$surf^A_{pt} = \sum_{p \in ptty} \sum_{i=1}^{\lvert p \rvert} \omega(p_i)$	
ptty	$\omega(p_i)$
service	$0.4 + def(p_i) + adm(p_i)$
webhandler	1.0
isapi	1.0
dynpage	0.6

where $def(p_i) = (default(p_i), 0.8, 0.0)$
$adm(p_i) = (run\_as\_admin(p_i), 0.9, 0.0)$

$surf^A_{dt} = \sum_{d \in dtty} \sum_{i=1}^{\lvert d \rvert} \omega(d_i)$	
dtty	$\omega(d_i)$
account	$0.7 + adg(d_i) + gue(d_i)$
file	$(weakACL(d_i), 0.7, 0.0)$
regkey	$(weakACL(d_i), 0.4, 0.0)$
share	$(weakACL(d_i), 0.9, 0.0)$
vdir	$(executable(d_i), 1.0, 0.0)$

where $adg(d_i) = (d_i \in \text{AdminGroup}, 0.9, 0.0)$
$gue(d_i) = (d_i.\text{name} = \text{“guest”}, 0.9, 0.0)$

$surf^A_{pe} = \sum_{e \in pety} \sum_{e_i \in \{IE,OE\}} \omega(e_i)$	
pety	$\omega(e_i)$
vbscript	$(app\_executes\_vbscript(e_i), 1.0, 0.0)$
jscript	$(app\_executes\_jscript(e_i), 1.0, 0.0)$
activex	$(app\_executes\_activex(e_i), 1.0, 0.0)$

where IE = Internet Explorer
OE = Outlook Express

Figure 5: Howard's Relative Attack Surface Quotient Metric

situation would be to calculate a "local attack surface" and "remote attack surface," each of which is appropriate for different threats.

For process targets, the weight function for services makes use of the access rights relation explicitly by referring to whether a service is a default service or if it is running as administrator.

The influence of the access rights relation is the most obvious for data targets, since it is used to determine whether an account is in a group with administrator privileges and whether it is a guest account. Note that we view an account as a shorthand for a subset of the access rights, i.e., a particular principal with a particular set of rights. Access rights is also used to determine the value of *weakACL* on files, registry keys, and shares. The predicate *weakACL* is true of its data target if all principals have all possible rights to it, i.e.,"full control".

The weights for process enablers are the count of the number of applications that enable a particular form of attack. Here, we consider only two applications, Internet Explorer and Outlook Express; in general, we would count others. Script-based attacks, for example, may target arbitrary process or data targets, but are enabled by applications that process script embedded in HTML documents. Malicious ActiveX components can similarly have arbitrary targets, but any successful attack is enabled by an application that allows execution of the potentially malicious component.

Our reformulation of Howard's original model shows that there are only 13 types of attack targets, rather than 17; in addition, there are 3 types of enablers.

6.3 Analysis of Attack Surface Calculation

The results of applying these specific weight functions for five different versions of Windows are shown in Figure 1. As mentioned in the introduction, the two main conclusions to draw are that *with respect to the 20 RASQ attack vectors* (1) the default version of a running Windows Server 2003 system is more secure than the default version of a running Windows 2000 system, and (2) a running Windows Server 2003 with IIS installed is only slightly less secure than a running Windows Server 2003 without IIS installed.

While it is too early to draw any conclusions about Windows Server 2003, the RASQ numbers are consistent with observed behavior in several ways:

- Worms such as Code Red and Nimda spread through a variety of mechanisms. In particular, Windows NT 4.0 systems were at far greater risk of being successfully attacked by these worms if the systems were installed with IIS than if they were not. This observation is consistent with the increased RASQ of this less secure configuration.
- Windows 2000 security is generally perceived as being an improvement over Windows NT 4.0 security [Wee01]; the differences in RASQ for the two versions in a similar configuration (i.e., with IIS enabled) reflect this perception.
- Conversely, Windows 2000 (unlike Windows NT 4.0) is shipped with IIS enabled by default, which means that the default system is actually *more* likely to be attacked. This observation is consistent with anecdotal evidence that many Windows 2000 users (including one author of this paper) affected by Code Red and Nimda had no idea they were actually running IIS.

As a sanity check, we also measured the RASQ in two "lockdown" configurations: applying IIS security checklists to both NT 4.0 with IIS [Tec01a] and Windows 2000 [Tec00]. Since the tasks specified in the lockdown instructions include disabling services, eliminating unnecessary accounts, and strengthening ACLs, the RASQ unsurprisingly decreases: on Windows NT 4.0, from 598.3 in the default configuration to 395.4 in the lockdown configuration; on Windows 2000, from 342.2 in the default to 305.1. These decreases are consistent with users' experience that systems in lockdown

configurations are more secure; for example, such configurations were not affected by the Code Red worm [Tec01b].

Our set of 20 attack vectors still misses types and instances, some of which also need more complex weight functions:

- For channels, some IPC mechanisms were not counted; for example COM is counted if DCOM is enabled, but otherwise it is not.
- For process targets, we did not handle executables that are associated with file extensions that might execute automatically (i.e., "auto-exec") or be executed mistakenly by a user. Also, we did not count ActiveX controls themselves as process targets, only as process enablers, i.e., whether applications such as IE and OE were set up to invoke them.
- The model treats all instances of each type the same, whereas some instances should probably be weighted differently. For example, a socket over which several complex protocols are transmitted should be a bigger contributor to the attack surface than a socket with a single protocol; and port 80 is well-known attack target that should get a higher weight than other channel endpoints.
- Just as for process targets that are services, for other types of process targets the weight function should take into consideration the privileges of the account that the process is executing as. For example, for versions of IIS $\leq$ 5.0, ISAPI filters always run as System, but in IIS 6.0, they run as Network Service by default.

These missing attack opportunities and refined weight functions suggest potential enhancements to Howard's RASQ model and the attack surface calculation.

7 Discussion of the RASQ Approach

We have some caveats in applying the RASQ approach naively:

- Obtaining numbers for individual attack vector classes is more meaningful than reading too much into an overall RASQ number. It is more precise to say that System A is more secure than System B because A has fewer services running by default rather than because A's RASQ is lower than B's. After all, summing terms with different units does not "type check". For example, if the number of instances in one attack vector class is N for System A and 0 for System B, but for a different attack vector class, the number is 0 for System A and N for System B, then all else being equal, the systems would have the same RASQ number. Clearly, the overall RASQ number does not reflect the security of either A or B with respect to the two different attack vector classes.
- The RASQ numbers we presented are computed for a given configuration of a running system. When an RASQ number is lower for System A than System B because certain features are turned off by default for System A and enabled by default for System B, that does not mean that System A is inherently "more secure"; for example, as the owner of System A begins to turn features on over time it can become just as insecure as System B. On the other hand, if 95% of deployed systems are always configured as System A initially (e.g., features off by default) and remain that way forever, then we could say in some global sense that we are "more secure" than if those systems were configured as System B.

- Do not compare apples to oranges. It is tempting to calculate an RASQ for Windows and one for Linux and then try to conclude one operating system is more secure or more attackable than the other. This would be a big mistake. For one, the set of attack vectors would be different for the two different systems. And even if the sets of attack vectors were identical, the threat models differ.

Rather, a better way to apply the RASQ approach for a given system is first to identify a set of attack vectors, and then for each attack vector class, compute a meaningful metric, e.g., number of running instances per class. Comparing different configurations of the same system per attack vector class can illuminate poor design decisions, e.g., too many sockets open initially or too many accounts with admin privileges. When faced with numbers that are too high or simply surprising, the system engineer can then revisit these design decisions.

8 Related Work

To our knowledge the notion of "attackability" as a security metric is novel. At the code level, many have focused on counting or analyzing bugs (e.g., [CYC+01, Gra90, LI93, SC91]) but none with the explicit goal of correlating bug count with system vulnerability.

At the system level, Browne et al. [BMAF01] define an analytical model that reflects the rates at which incidents are reported to CERT. Follow-on work by Beattie et al. [BAC+02] studies the timing of applying security patches for optimal uptime based on data collected from CVE entries. Both empirical studies focused on vulnerabilities with respect to their discovery, exploitation, and remediation over time, rather than on a single system's collective points of vulnerability.

Finally, numerous websites, such as Security Focus [Foc], and agencies, such as CERT [CER] and MITRE [MIT], track system vulnerabilities. These provide simplistic counts, making no distinction between different types of vulnerabilities, e.g., those that are more likely to be exploited than others, or those relevant to one operating system over another. Our notion of attackability is based on separable types of vulnerabilities, allowing us to take relative measures of a system's security.

9 Future Work

Our state machine model is general enough to model the behavior an adversary attacking a system. We identified some useful abstract dimensions such as targets and enablers, but we suspect there are others that deserve consideration. In particular, if we were to represent *configurations* more explicitly, rather than as just states of the system (in particular the resources and access rights), then we can more succinctly define what it means for a process to be running by default or whether an account is enabled.

Further into the future we imagine a "dial" on the workstation display that allows developers to determine if they have just increased or decreased the attack surface of their code. We could flag design errors or design decisions that tradeoff performance for security. For example, consider a developer debating whether to open up several hundred sockets at boot-up time or to open sockets on demand upon request by authenticated users. For a long-running server, the first approach is appealing because it improves responsiveness and is a simpler design. However, even a simple attack surface calculation would reveal a significant increase in the server's attackability; this potential security cost would need to be balanced against the benefits.

Measuring security, quantitatively or qualitatively, has been a long-standing challenge to the community. The need to do so has recently become more pressing. We view our work as a first step

in revitalizing this research area. We suggest that the best way to begin is to start counting what is countable; then use the resulting numbers in a qualitative manner (e.g., doing relative comparisons). Perhaps over time our understanding will then lead to meaningful quantitative metrics.

References

[BAC+02] Steve Beattie, Seth Arnold, Crispin Cowan, Perry Wagle, Chris Wright, and Adam Shostack. Timing the application of security patches for optimal uptime. In *2002 LISA XVI*, pages 101–110, November 2002.

[BMAF01] Hilary Browne, John McHugh, William Arbaugh, and William Fithen. A trend analysis of exploitations. In *IEEE Symposium on Security and Privacy*, May 2001. CS-TR-4200, UMIACS-TR-2000-76.

[But03] Shawn Butler. *Security Attribute and Evaluation Method*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 2003.

[CER] CERT. CERT/CC Advisories. http://www.cert.org/advisories/.

[CYC+01] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallen, and Dawson Engler. An empirical study of operating systems errors. In *ACM Symposium on Operating Systems Principles*, pages 73–88, October 2001.

[Foc] Security Focus. http://www.securityfocus.com/vulns/stats.shtml.

[Gra90] J. Gray. A census of tandem system availability between 1985 and 1990. *IEEE Transactions on Software Engineering*, 39(4), October 1990.

[How03] Michael Howard. Fending Off Future Attacks by Reducing the Attack Surface, February 2003. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncode/html/secure02132003.asp.

[LABW92] Butler Lampson, Martin Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: Theory and practice. *ACM TOCS*, 10(4):265–310, November 1992.

[Lam74] Butler Lampson. Protection. *Operating Systems Review*, 8(1):18–24, January 1974.

[LI93] I. Lee and R. Iyer. Faults, symptoms, and software fault tolerance in the tandem GUARDIAN operating system. In *Proceedings of the International Symposium on Fault-Tolerant Computing*, 1993.

[Mic] Microsoft Security Response Center. Security Bulletins. http://www.microsoft.com/technet/treeview/?url=/technet/security/current.asp?frame=true.

[MIT] MITRE. Common Vulnerabilities and Exposures. http://www.cve.mitre.org/.

[PW03] Jon Pincus and Jeannette M. Wing. A Template for Microsoft Security Bulletins in Terms of an Attack Surface Model. Technical report, Microsoft Research, 2003. in progress.

[SC91] M. Sullivan and R. Chillarge. Software defects and their impact on system 118 availability. In *Proceedings of the International Symposium on Fault-Tolerant Computing*, June 1991.

[Sch91] Fred B. Schneider. *Trust in Cyberspace*. National Academy Press, 1991. CSTB study edited by Schneider.

[Tec00] Microsoft TechNet. Secure Internet Informations Services 5 Checklist, June 2000. http://www.microsoft.com/technet/security/tools/chklist/iis5chk.asp.

[Tec01a] Microsoft TechNet. Microsoft Internet Information Server 4.0 Security Checklist, July 2001. http://www.microsoft.com/technet/security/tools/chklist/iischk.asp.

[Tec01b] Microsoft TechNet. Microsoft Security Bulletin MS01-033, June 2001. http://www.microsoft.com/technet/security/bulletin/MS-01-033.asp.

[Wee01] Information Week. Windows 2000 Security Represents a Quantum Leap, April 2001. http://www.informationweek.com/834/winsec.htm.