An Attack Surface Metric

Pratyusa Manadhata **Jeannette M. Wing**

July 2005
CMU-CS-05-155

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

We propose a metric to determine whether one version of a software system is more secure than another with respect to the system's *attack surface*. Rather than count bugs at the code level or count vulnerability reports at system level, we measure a system's *attackability*, i.e., how likely the system will be successfully attacked. We define the *attack surface* of a system in terms of the system's attackability along three abstract dimensions: method, data, and channel. Intuitively, the larger the attack surface, the more likely the system will be attacked, and hence the more insecure it is. We demonstrate the use of the attack surface metric by measuring and comparing the attack surface of two versions of a hypothetical IMAP server.

This research was sponsored by the Army Research Office under contract no. DAAD190110485 and DAAD19-02-1-0389, the National Science Foundation under grant no. CCR-0121547 and CNS-0433540, and the Software Engineering Institute through a US government funding appropriation. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the sponsoring institutions, the U.S. Government or any other entity.

Keywords: Security metric, entry point, exit point, attackability, attack class, attack surface, attack surface metric

1 Introduction

Measurement of security, both qualitatively and quantitatively, has been a long standing challenge to the research community, and is of practical import to industry today. Industry has responded to demands for improvement in software security by increasing effort [1] into creating "more secure" products and services. But how can industry determine if this effort is paying off? Our work is motivated by the questions faced by industry today: 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?

1.1 A New Metric

In this paper, we propose a metric to determine whether one version of a system is more secure than another. Rather than measure the absolute security of a system, we measure its relative security: Given two versions, A and B, of a system, we measure whether version A is more secure than version B with respect to their *attack surface*. We do not use the *attack surface metric* to determine whether a version of a system is absolutely good or bad, rather to determine whether one version of a system is relatively better or worse than another.

Intuitively, a system's attack surface is the ways in which the system will be successfully attacked. We define the attack surface of a system in terms of the system's *resources*. An attacker uses a system's resources to attack a system, hence a system's resources contribute to the system's attack surface. Intuitively, the more resources available to the attacker, the more exposed the attack surface. The more exposed the attack surface, the more ways the system can be attacked, and hence the more insecure it is.

Given two versions, A and B, of a system, we compare their attack surface to determine whether one is more secure than another. The attack surface measurements might be incomparable because of the way we define attack surface along multiple dimensions. We can, however, use the attack surface measurements along with the knowledge of the usage scenario of the system to determine whether version A is more secure than version B. We illustrate this point with an example in Section 5.3.

1.2 Contributions and Roadmap

Our contributions in this paper are the following:

- We introduce the *entry point and exit point framework* to identify the resources that contribute to a system's attack surface.
- We introduce the notion of *attackability* to determine the attack surface contribution of a resource. We define the attackability of a resource as a *cost-benefit ratio* to the attacker.
- We introduce the notion of *attack classes* for the convenience in defining the attack surface of a system. We present formal definitions of attack classes, and attack surface in terms of a state machine model of a system. We outline a general method to measure the attack surface of a system.

[1]For example, Microsoft's *Trustworthy Computing Initiative*.

The rest of the paper is organized as follows. We compare our approach to related work in Section 2. We introduce the entry point and exit point framework in Section 3. We present the definitions of attackability, attack classes and attack surface in Section 4. We illustrate the use of our attack surface metric in Section 5. We discuss possible avenues of future work in Section 6. We summarize in Section 7.

2 Related Work

Michael Howard of Microsoft informally introduced the notion of attack surface for the Windows operating system [12]. Pincus and Wing [13] further elaborated, and we [19] formalized Howard's Relative Attack Surface Quotient (RASQ) measurements for the Windows operating system. We compare our work with prior work on attack surface in Section 2.1, and with other security metrics in Section 2.2.

2.1 Attack Surface Metric

In our prior work, we define the attack surface of a system 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 [19]. Every system action can potentially be part of an attack, and hence contributes to attack surface. Similarly, every system resource also contributes to attack surface. Intuitively, the more actions available to a user or the more resources accessible through these actions, the more exposed the attack surface. Rather than consider all possible system resources, we narrow our focus on a *relevant* subset of resource types. Attacks carried out over the years show that certain system resources are more likely to be used in an attack than others. Hence we do not treat all system resources equally. We categorize the system resources into *attack classes* based on a given *set of properties* associated with the resources. 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. We use the notion of attack class to distinguish between resources with different attackability. These attack classes together constitute the attack surface of a system.

We measured the attack surface of four different versions of the Linux operating system [19]. Howard et al. measured the attack surface of seven different versions of the Windows operating system [13]. The results of both the Linux and Windows measurements confirm perceived beliefs about the relative security of the different versions. We realized, however, that our attack surface measurement method suffers from two major drawbacks: (1) there is no systematic way to identify the relevant subset of system resources that can be used in an attack, and (2) there is no systematic way to identify the set of properties associated with each resource. Hence it is difficult to identify the attack classes of a system. In our Linux attack surface measurement work, we used the history of attacks on Linux to identify the relevant subset of resources. Similarly, we relied on our knowledge of the Linux operating system to identify the properties of interest. Howard et al.'s Windows attack surface measurement method suffers from a similar drawback: there is no systematic way of identifying the attack classes of the Windows operating system. Howard et al. used the history of the attacks on Windows to identify twenty attack classes.

Our current work is motivated by the above mentioned difficulties in identifying the attack classes of a system. We define the attack surface of a system in terms of the attackability of the system's resources. We use the entry point and exit point framework to identify the relevant subset of resources that contribute to the attackability of a system. We determine the attackability of each resource using a cost-benefit ratio to the attacker. We group the resources into attack classes based on their attackability. The attackability of these attack classes constitute the attack surface of a system.

2.2 Other Security Metrics

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 [7], Microsoft Security Bulletins [15], MITRE Common Vulnerabilities and Exposures (CVEs) [16] etc. We argue that both measurements, while useful, are less than satisfactory.

At the code level, many have focused on counting and analyzing bugs (e.g., [8], [11], [17], [22]). Using bug counts as a security metric, however, 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 [7] and MITRE [16], and websites, such as SecurityFocus [5], 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. Our attack surface metric strikes at the *design level* of a system: above the level of code, but below the level of the entire system.

Butler [6] uses multi-attribute risk assessment method to obtain a prioritized list of threats to an organization. The method ranks threats such as scanning, DDoS, and password nabbing based on threat frequency and expected outcome. The method focuses on threats to an organization rather than the software systems used in the organization.

Browne et al. [4] give a mathematical model to reflect the rate at which incidents involving exploits of vulnerability are reported to the CERT. Beattie et al. [2] 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. [3, 18] 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. [24] 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. [21] model the system as a privilege graph [10] 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. The approach of using vulnerabilities as a measure of security fails to capture a system's future attackability as it misses out on future vulnerabilities. This approach also does not take into account a system's design. Our approach focuses on a system's design, and hence can be used by system designers and developers to improve the security of a system.

3 Entry Point and Exit Point Framework

Informally, *entry points* of a system are the ways through which data "enters" the system from its environment, and *exit points* are the ways through which data "exits" from the system to its environment. We know from the past that many attacks on software systems require the attacker to either send data into the system, or receive data from the system. Hence the entry points and the exit points of a system act as the entry points of attack on the system. We believe that the identification of the entry points and exit points of a system is the important first step towards the detection and prevention of such attacks. The entry point and exit point framework is a formal framework to define the entry points and exit points of a system. We use the framework to identify the resources that contribute to a system's attack surface. We discuss possible future uses of the framework in Section 6. We introduce a model of a system and its environment in Section 3.1. We use the model to define the entry points and exit points. We give some auxiliary definitions (*interaction* in Section 3.2 and *relies on* in Section 3.3) that we use later to define entry points in Section 3.4 and exit points in Section 3.5.

3.1 Model

We consider a model with a set, S, of systems, a user, U, and a data store, D. For a given system, $s \in S$, we define its environment, $E_s = \langle U, D, T\rangle$, to be a three-tuple where U is the user, D is the data store, and $T = S \setminus \{s\}$, is the set of systems excluding s. We are interested in identifying the entry points and exit points of every system $s \in S$. The system s interacts with its environment E_s, hence we define the entry points and exit points of s with respect to E_s. Figure 1 shows a system, s, and its environment, $E_s = \langle U, D, \{s_1, s_2, \}\rangle$. We model every system $s \in S$ as a *state*

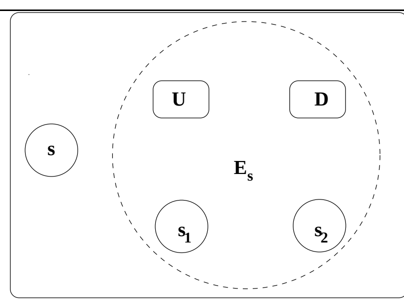

Figure 1: A system and its environment.

machine. A state machine, $\mathsf{M} = \langle \mathsf{S}, \mathsf{I}, \mathsf{M}, \mathsf{T}\rangle$, is a four tuple where S is a set of states, $\mathsf{I} \subseteq \mathsf{S}$ is a set of initial states, M is a set of methods, and $\mathsf{T} \subseteq \mathsf{S} \times \mathsf{A} \times \mathsf{S}$ is a transition relation. A state $\mathsf{s} \in \mathsf{S}$ is a mapping of the *variables* to their *values*: s: $\mathsf{Var} \rightarrow \mathsf{Val}$. A state transition, $\mathsf{s} \times \mathsf{a} \times \mathsf{s}'$, is the invocation of a method m in state s resulting in state $\mathsf{s'}$. We specify the actions by *pre* and *post* conditions.

A method of a system receives *arguments* as input and returns *result* as output. We model the access control list of the methods as a special state variable, $\mathsf{acl} \in \mathsf{domain(s)}$. acl is a list of tuples, $\langle m\,, P\rangle$, where m is a method and P is a set of entities (other systems in S or the user U) that has the *invocation privilege* on m.

Every system has a set of *communication channels*. The channels of a system s are the means by which the user U or any system $s_1 \in T$ communicates with s. We model each channel as a special state variable in $\mathsf{domain(s)}$. Each channel c has a *type* associated with it. The type of a channel specifies the rules of communication. We model the access control list of the channels as a special state variable, $\mathsf{acl}_c \in \mathsf{domain(s)}$. acl_c is a list of tuples, $\langle c\,, P\rangle$, where c is a channel and P is a set of entities (other systems in S or the user U) that can communicate with s using the channel c. We assume that we know the set of channels of every system $s \in S$. In practice, however, we extract this information from the codebase of s.

We say an entity e *can connect* to a system s if s has a channel c such that $\langle c\,, P\rangle \in \mathsf{acl}_c$ and $e \in P$. We say an entity, e, *can invoke* a method m of s if e can connect to s, and e has the invocation privilege on m. Given a system-environment pair $\langle s,\, E_s\rangle$, a method m of s is an *external method* if the user U or the methods of any system $s_1 \in T$ can invoke m. s's API is simply the set of its external methods. A method m is an *internal method* if only methods of s can invoke m. We say "a system s invokes system s_1's API" to mean a method of s invokes an external method of s_1.

The user U and the data store D are global with respect to the systems in S. We model the data store D as a separate entity to allow sharing of data among all the systems in the environment. The data store D is a set of typed *data items*. Specific examples of data items are files, cookies, registry entries, and user accounts. Every data item $d \in D$ has a set of methods based on its type. We categorize these methods into the generic classes of read, write, create and delete methods. A method can belong to more than one class. We say an entity "reads (writes etc.)" a data item d if the entity invokes a method from the read (write etc.) class of d. Every data item d also has an access control list, acl, of its methods. acl is a list of tuples, $\langle m\,, P\rangle$, where m is a method of d and P is a set of entities (systems in S or the user U) that has the invocation privilege on m. Every data item d also has a set of channels, and an access control list, acl_c, of the channels. We say an entity p *can connect* to a data item d if d has a channel c such that $\langle c\,, P\rangle \in \mathsf{acl}_c$ and $p \in P$. We say an entity e *can read (write etc.)* a data item d if e can connect to d, and e has invocation privilege on a method from the read (write etc.) class of d.

For simplicity we assume only one user U present in the environment. U represents the adversary who attacks the systems in S. U can invoke the external methods of the systems in S and the methods of the data items in D. We say "U invokes a system s's API" to mean U invokes an external method of s.

Every system $s \in S$ *relies on* a subset of systems of S in order to function in an expected manner. s's methods invoke the APIs of these systems. s's methods also read, write, create and delete data items in the data store D. We assume, for every method m of s, we know the APIs that m invokes, and the data items that m reads, writes, inserts or deletes. In practice, however, we extract this information from the code base of s.

In the following section, we use the above model of a system s and its environment E_s to define s's *entry points* and *exit points* with respect to the given environment E_s.

3.2 Interaction Relations

The system s interacts with the entities present in its environment E_s. Data items flow into and out of the system s as a result of these interactions. The definitions of the interaction relations formalize our intuition of data flow from the environment into the system s. The *input interaction relation* captures all interactions that enable the user U (or the attacker) to send data into the system s. Similarly, the *output interaction relation* captures all interactions that enable the user U (or the attacker) to receive data from the system s.

3.2.1 Input Interaction Relation

Figure 2 shows different types of input interactions. If a system s input-interacts with a system s_1, then data flows from s into s_1 (Fig 2(a)). If the user U input-interacts with a system $s \in S$, then data flows from U into s (Fig 2(c)). The input interaction relation is transitive, but neither reflexive nor symmetric.

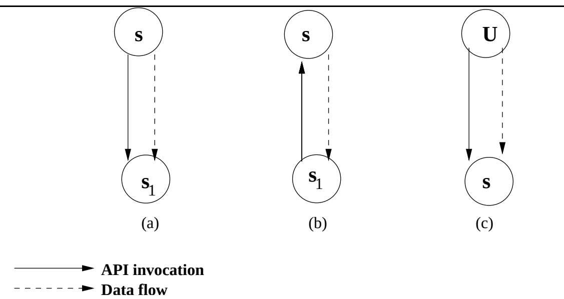

Figure 2: Examples of input interactions.

(1) The system s *can directly input-interact with system* $s_1 \in T$ if (a) s can invoke s_1's API, and pass data items as input to s_1's API (Fig 2(a)), or (b) s_1 can invoke s's API and receive data items as result returned from s's API (Fig 2(b)).

The system s *can indirectly input-interact with system* $s_2 \in T$ if there exists a system $s_1 \in T$ such that (a) s can directly input-interact with s_1 and s_1 can directly input-interact with s_2, or (b) s can directly input-interact with s_1 and s_1 can indirectly input-interact with s_2, or (c) s can indirectly input-interact with s_1 and s_1 can indirectly input-interact with s_2.

Definition 1. *The system s* input-interacts with system $s_1 \in T$ *if s can input-interact directly or indirectly with s_1.*

(2) The user U *can directly input-interact with the system* s if U can invoke s's API and pass data items as input to s's API (Fig 2(c)).

The user U *can indirectly input-interact with system* s if (a) there exists a data item d such that s reads d from the data store D and U can write d to the data store D, or (b) there exists a system $s_1 \in T$ such that U can directly input-interact with s_1 and s_1 input-interacts with s, or (c) U can indirectly input-interact with s_1 and s_1 input-interacts with s.

Definition 2. *The user U* input-interacts with system s *if U can directly or indirectly input-interact with s.*

3.2.2 Output Interaction Relation

Figure 3 shows different types of output interactions. If a system s output-interacts with a system s_1, then data flows from s_1 into s (Fig 3(a)). If the user U output-interacts with a system $s \in S$, then data flows from s into U (Fig 3(c)). The output interaction relation is transitive, but neither reflexive nor symmetric.

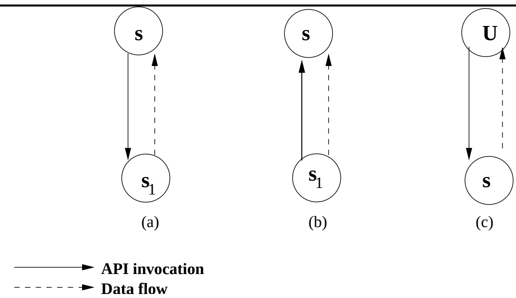

Figure 3: Examples of input interactions.

(1) The system s *can directly output-interact with system* $s_1 \in T$ if (a) s can invoke s_1's API, and receive data items as result returned from s_1's API (Fig 3(a)), or (b) s_1 can invoke s's API and pass data items as input to s's API (Fig 3(b)).

The system s *can indirectly output-interact with system* $s_2 \in T$ if there exists a system $s_1 \in T$ such that (a) s can directly output-interact with s_1 and s_1 can directly output-interact with s_2, or (b) s can directly output-interact with s_1 and s_1 can indirectly output-interact with s_2, or (c) s can indirectly output-interact with s_1 and s_1 can indirectly output-interact with s_2.

Definition 3. *The system s* output-interacts with system $s_1 \in T$ *if s can output-interact directly or indirectly with s_1.*

(2) The user U *can directly output-interact with the system* s if U can invoke s's API and receive data items as result returned from s's API (Fig 3(c)).

The user U *can indirectly input-interact with system* s if (a) there exists a data item d such that s writes d to the data store D and U can read d from the data store D, or (b) there exists a system $s_1 \in T$ such that U can directly output-interact with s_1 and s_1 output-interacts with s, or (c) U can indirectly output-interact with s_1 and s_1 output-interacts with s.

Definition 4. *The user U* output-interacts with system s *if U can directly or indirectly output-interact with s.*

3.3 Relies On Relation

The system s depends on many other systems in its environment E_s to function in an expected manner. The definition of the relies on relation formalizes this dependency between s and other systems in its environment. Relies on relation is transitive, but neither reflexive nor symmetric.

(1) The system s *directly relies on* a system $s_1 \in T$ if s invokes s_1's API.

(2) The system s *indirectly relies on* a system $s_2 \in T$ if there exists a system $s_1 \in T$ such that (a) s directly relies on s_1 and s_1 directly relies on s_2, or (b) s directly relies on s_1 and s_1 indirectly relies on s_2, or (c) s indirectly relies on s_1 and s_1 indirectly relies on s_2.

Definition 5. *The system s* relies on *a system $s_1 \in T$ if s directly or indirectly relies on s_1.*

3.4 Entry Points

The methods of the system s that receive data items from s's environment as input are s's entry points . A method m of s can receive data directly or indirectly from the environment.

Definition 6. *A method m* receives a data item d directly *if (a) m receives d as input or, (b) m reads d from the data store, or (c) m invokes a system s_1's API and receives d as result returned from s_1's API.*

Definition 7. *A method m* receives a data item d indirectly *if a method m_1 of s receives d directly and passes d as input to m.*

The systems on which s relies on can also pass data into s, and hence act as s's entry points.

Definition 8. *A* direct entry point *of the system s is a method m of s such that (a) the user U can invoke m (Figure 4(a)), or (b) a system $s' \in T$ can invoke m, and U input-interacts with s' (Figure 4(b)), or (c) m reads a data item d from the date store and the user U can write d to the data store D (Figure 4(c)), or (d) m invokes the API of a system s' on which s relies, and the user U input-interacts with s' (Figure 4(d)).*

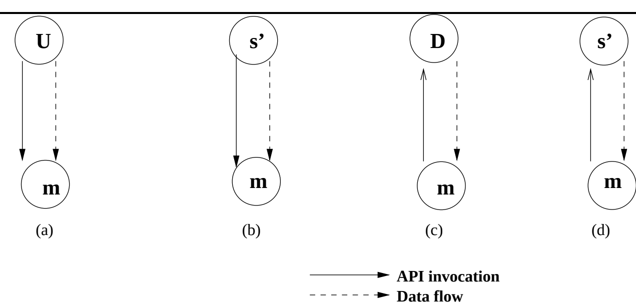

Figure 4: Direct Entry Point.

Definition 9. *An* indirect entry point *of the system s is a method m of s such that (1) m is not a direct entry point of s, and (2) a direct entry point m_1 of s invokes m, and passes data items received by it to m as input (Figure 5(a)), or (3) an indirect entry point m_2 of s invokes m, and passes data items received by it to m as input (Figure 5(b)).*

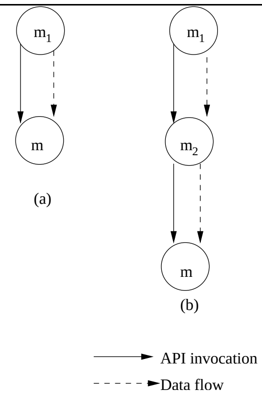

Figure 5: Indirect Entry Point.

Definition 10. *The set of* system entry points *of the system s is the disjoint union of the set of direct entry points and the set of indirect entry points of s.*

Definition 11. *The set of* infrastructure entry points *of the system s is the union of the set of direct entry points of all systems s' such that s relies on s', and the user U input-interacts with s'.*

3.5 Exit Points

The methods of a system s that return data items as result to s's environment are s's exit points. The systems on which s relies can also receive data from s, and hence act as s's exit points. A method m of s can be both an entry point and an exit point of s.

Definition 12. *A* direct exit point *of the system s is a method m of s such that (a) the user U can invoke m, and receive data items as result returned from m (Figure 6(a)), or (b) a system $s' \in T$ can invoke m and receive data items as result returned from m, and U output-interacts with s' (Figure 6(b)), or (c) m writes a data item d to the data store and the user U can read d from the data store D (Figure 6(c)), or (d) m invokes the API of a system s' on which s relies and passes data items as input to s''s API, and the user U output-interacts with s' (Figure 6(d)).*

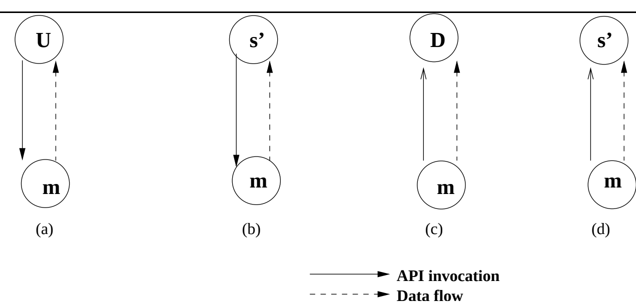

Figure 6: Direct Exit Point.

Definition 13. *The set of* system exit points *of the system s is the set of direct exit points of s. Notice that a system does not have indirect exit points.*

Definition 14. *The set of* infrastructure exit points *of the system s is the union of the set of direct exit points of all systems s' such that s relies on s', and the user U output-interacts with s'.*

3.6 Untrusted Data

A successful attack on the system s requires the user U (or the attacker) to send (receive) data into (from) s. U can send (receive) data into (from) s directly by invoking a method of s, or indirectly using the *persistent* data items of the global data store D. The methods of s act as entry points of attack in the former case, whereas the persistent data items act as the entry points in the latter case. For example, s might read a persistent data item d from the data store D after U writes d to the data store. Similarly, U might read a persistent data item d from the data store D after s writes d to the data store. Hence the persistent data items enable the user U to send (receive) data into (from) s, and act as the entry points of attack. The user U (or the attacker) can use a persistent data item d in an attack if U can read or write d. Hence those persistent data items of s that U can read or write are the untrusted data items. The user U (or the attacker) can attack the system s if s has untrusted data items.

Definition 15. *An* untrusted data input *of the system s is a persistent data item d such that a direct entry point of s receives d, and the user U can write d to the data store D.*

Definition 16. *An* untrusted data output *of the system s is a persistent data item d such that a direct exit point of s writes d to the data store D, and the user U can read d from the data store D.*

Definition 17. *The set of* untrusted data items *of the system s is the disjoint union of the set of untrusted data inputs and the set of untrusted data outputs of s.*

3.7 Open Channels

A successful attack on the system s requires the user U (or the attacker) to connect to s using a communication channel. Recall from Section 3.1 that U can connect to s using a channel c if the tuple $\langle c, U\rangle$ belongs to the channel access control list,acl_c, of s. The channels of s that U can use to connect to s directly or indirectly are the open channels. The user U (or the attacker) can attack the system s if s has open channels.

Definition 18. *An* open channel *of a system s is a channel c of s such that (a) the user U can connect to s through c, or (b) a system $s' \in T$ can connect to s through c, and U either input-interacts or output-interacts with s.*

4 Attack Surface Definitions

In this section, we give the definitions of attackability, attack class, and attack surface in terms of our model introduced in Section 3.1. We define the attack surface of a system in terms of resources. A resource of a system is either a method of the system, a channel of the system, or a data item in the global data store. We identify the accessible subset of resources that contribute to attack surface using the entry point and exit point framework introduced in Section 3. We define the attackability of a resource using a cost-benefit ratio in Section 4.2. We give the definition of attack class in Section 4.3 and attack surface in Section 4.4. We introduce our attack surface measurement method in Section 4.5.

4.1 Accessible Resources

In our model, some resource of a system might not contribute to the attack surface of the system. Only those resources of a system that the attacker can use in an attack contribute to the system's attack surface. A resource's access rights determine whether the attacker can use the resource in an attack. A successful attack requires the attacker to connect to the system, invoke the system's methods, and send (receive) data into (from) the system. We consider only *entry point attacks* and *exit point attacks*, i.e., attacks that require the attacker to either send data into the system or receive data from the system. Hence the attacker uses a channel in an attack if the attacker can connect to the system through the channel. Similarly, the attacker uses a method in an attack if the attacker can send (receive) data items into (from) the system by invoking the method, and the attacker uses a data item in an attack if the attacker can send (receive) the data item into (from) the system.

We use the entry point framework to identify the relevant subset of resources that contribute to a system's attack surface. By definition, the set of system entry points (exit points) are the methods that the attacker can invoke to send (receive) data into (from) the system. The set of open channels are the channels that the attacker can use to connect to the system. The set of untrusted data items are the data items that the attacker can send (receive) into (from) the system. Hence the set of system entry points and system exit points, the set of open channels, and the set of untrusted data items are the resources that contribute to the system's attack surface.

4.2 Attack Weights of Resources

In our model, not all resources are equally likely to be used in an attack, hence not all resources contribute equally to a system's attack surface. Some resources are more likely to be used in an attack than others, e.g., a method running as `root` is more likely to be attacked than a method running as `non-root`. The attackability of a resource is the likelihood of the resource being used in an attack. We define the attackability of a resource in terms of a cost-benefit ratio to the attacker in using the resource in an attack. The benefit to the attacker is in terms of the *damage potential* of a resource, i.e., how much damage the attacker can do to the system by using a resource in an attack. The cost to the attacker is in terms of the *effort* required to use a resource, i.e., the effort the attacker spends to acquire the necessary access rights in order to be able to use a resource in an attack. The higher the benefit, the higher the attackability, and the higher the cost, the lower the attackability.

We define damage potential and effort in terms of the attributes of a resource. The definitions depend on the type of the resource, i.e., method, channel, or data. We define damage potential and effort based on our intuition and experience of a system. We leave for future work a more formal characterization and justification of our notion of damage potential and effort.

4.2.1 Damage Potential

In our model, the attacker uses a method in an attack by exploiting a vulnerability in the method. The attacker gains the same *privilege* as the method by exploiting a vulnerability in the method. For example, the attacker gains `root` privilege by exploiting a buffer overrun vulnerability in a method running as `root`. The attacker can cause damage to the system after gaining `root` privilege. We propose that the the damage that the attacker can do to the system depends on the method's privilege. So the damage potential of a method depends on the method's privilege.

In our model, the attacker uses communication channels to connect to a system, and send (receive) data to (from) a system. A channel's *type* imposes restriction on the data exchange allowed using the channel, e.g., a `socket` allows raw bytes to be exchanged where as a `RPC end point` does not. We informally propose that the damage the attacker can do to a system using a channel depends on the restriction imposed by the channel's type. Hence the damage potential of a channel depends on the channel's type.

In our model, the attacker uses persistent data items to indirectly send (receive) data into (from) a system. A persistent data item's *type* imposes restriction on the data exchange, e.g., a `file` can contain executable code whereas a `registry entry` can not. The attacker can send executable code into the system by using a `file` in an attack, but the attacker can not do the same using a `registry entry`. We informally propose that the damage the attacker can do to a system using a persistent data item depends on the restriction imposed by the data item's type. Hence the damage potential of a persistent data item depends on the data item's type.

4.2.2 Effort

The attacker can use a resource in an attack if the attacker has required *access rights*. The attacker spends effort to acquire these access rights. The effort required to acquire an access right level of a system is proportional to the difference between the access right level and the lowest access right level of the system. Hence the effort the attacker needs to spend to use a resource in an attack depends on the access rights of the resource.

4.2.3 Attackability

We define the attackability of a resource in terms a cost-benefit ratio.

Definition 19. *The attackability of a resource, r, with damage potential, dp, and access rights, ar, is* $\frac{dp}{ar}$.

Hence the attackability of a method is $\frac{privilege}{access\,rights}$, the attackability of a channel is $\frac{type}{access\,rights}$, and the attackability of a data item is $\frac{type}{access\,rights}$. Note that the cost-benefit ratio is a symbolic ratio rather than a numeric ratio.

4.3 Attack Class

An attack class is a set of methods, or a set of channels, or a set of data items such that each member of the set has the same attackability. We assume we have a system, s, and its environment, $E_s = \langle U, D, T \rangle$. We use entry point analysis to identify a set, M, of system entry points and system exit points, a set, C, of open channels, and a set, I, of untrusted data items.

Definition 20. *A* system attack class *of the system s is a subset of the set M such that every method belonging to the subset has the same privilege and access rights.*

Definition 21. *A* channel attack class *of the system s is a subset of the set C such that every channel belonging to the subset has the same type and access rights.*

Definition 22. *A* data attack class *of the system s is a subset of the set I such that every data item belonging to the subset has the same type and access rights.*

Definition 23. *The* attackability, ac(A), of an attack class A *is the total attackability of the members of A.*

4.4 Attack Surface

We define the attack surface of a system in terms of the system's attackability, i.e., the total attackability of the system's resources. Each resource of a system that belongs to an attack class contributes to the total attackability of a system. The attackability contributions of a method, however, is different than the attackability contribution of a data item or a channel. The numeric values of the attackability of a method and the attackability of a channel might be same, but the units of attackability are different. Hence we compute the total attackability of a system along three dimensions: method, channel, and data . The total attackability along a dimension is the sum of the attackability of all the attack classes along that dimension. The attackability of a system along these three dimensions define the system's attack surface.

We assume we have a system, s, and its environment, $E_s = \langle U, D, T \rangle$. We use entry point analysis to identify a set, M, of system entry points and exit points, a set, C, of open channels, and a set, I of untrusted data items. We group the methods in M into the set, SAC, of system attack classes. We group the channels in C into the set, CAC, of channel attack classes. We group the data items in I to the set, DAC, of data attack classes.

Definition 24. *The* system attackability, SA, *of the system s is the total attackability of the attack classes $\in SAC$.*

Definition 25. *The* channel attackability, CA, *of the system s is the total attackability of the attack classes $\in CAC$.*

Definition 26. *The* data attackability, DA, *of the system s is the total attackability of the attack classes $\in DAC$.*

Definition 27. *The* attack surface *of the system s is the three-tuple $\langle SA, CA, DA \rangle$.*

4.5 Attack Surface Measurement Method

In this section we outline a general method to measure the attack surface of a system. We base our method on our formal model and definitions described in earlier sections. We assume we have a a system, s, and its environment, $E_s = \langle U, D, T \rangle$.

1. We identify the set, M, of system entry points and system exit points, the set, C, of open channels, and set, I, of untrusted data items of s.

2. We group the methods in M into the set, SAC, of system attack classes, the channels in C into the set, CAC, of channel attack classes, and the data items in I to the set, DAC, of data attack classes based on their attackability.

3. We compute the total system attackability, SA, the total channel attackability, CA, and the total data attackability, DA.

4. The attack surface of s is $\langle SA, CA, DA \rangle$.

5 An Example Attack Surface Metric

In the previous sections, we have outlined a generic qualitative method for determining the attackability of a resource, and the attack surface of a system. We describe a specific quantitative method for computing the attackability of a resource in Section 5.1. Our quantitative method is one of several possible ways of quantifying attackability. We use the attackability computation described in Section 5.1 to quantitatively measure the attack surface of a system in Section 5.2. We demonstrate the use of our attack surface metric in Section 5.3.

5.1 Attackability Computation

We define simple metrics to measure the damage potential and the effort so that we can determine the attackability of a resource. A simple such metric would be a natural number. We describe a method of assigning numeric values to damage potential and effort. Since damage potential and effort depend on attributes of a method or a resource, we assign numeric values to the following attributes: the privilege levels of methods, the types of channels, the types of data items, and the access right levels. In practice, the users of our attack surface metric will assign these numeric values based on their experience of a system.

We assume there is a total ordering, TO_p, among the set, P, of privilege levels of the methods. It is natural to associate a total order among the set of privilege levels, e.g., `root` > `non-root`. The higher the privilege, the higher the damage potential.

We assume there is a total ordering, TO_c, among the set, C, of types of the channels. Mehta et al. [20] define four types of communication channels in their taxonomy of software connectors: `procedure call, event, data access,` and `stream`. We assume the following order: `stream` > `procedure call` > `data access` > `event`. The higher the channel type, the higher the damage potential.

We assume there is a total ordering, TO_t, among the set, T, of types of the persistent data items. Specific examples of persistent data items are `files`, `database records`, `cookies` and `registry entries`. We assume the following total order: `files` > `database records` > `cookies` > `registry entries`. The higher the type, the higher the damage potential.

We further assume there is a total ordering, TO_a, among the set, A, of access right levels. It is natural to associate a total order among the set of access right levels, e.g., `authenticated` > `unauthenticated`. The higher the access rights, the higher the effort.

We define a function $BM : P \rightarrow N$ to assign numeric values to privilege levels, a function $BD : T \rightarrow N$ to assign numeric values to types of data items, a function $BC : C \rightarrow N$ to assign numeric values to types of channels, and a function $C : A \rightarrow N$ to assign numeric values to access right levels. We assign numeric values in accordance to the total ordering such that the following hold.

$$\begin{array}{c} \forall p_1, p_2 \in P.(p_1\, TO_p\, p_2) \Longrightarrow (BM(p_1) \,\leq\, BM(p_2)) \\ \forall t_1, t_2 \in T.(t_1\, TO_t\, t_2) \Longrightarrow (BD(t_1) \,\leq\, BD(t_2)) \\ \forall c_1, c_2 \in C.(c_1\, TO_c\, c_2) \Longrightarrow (BC(c_1) \,\leq\, BC(c_2)) \\ \forall a_1, a_2 \in A.(a_1\, TO_e\, a_2) \Longrightarrow (C(a_1) \,\leq\, C(a_2)) \end{array}$$

Definition 28. *The* attackability, ac(m), of a method m *with privilege* p *and access rights* a *is* $\frac{BM(p)}{C(a)}$.

Definition 29. *The* attackability, ac(d), of a data item d *of type* t *and access rights* a *is* $\frac{BD(t)}{C(a)}$.

Definition 30. *The* attackability, ac(c), of a channel c *with type* t *and access rights* a *is* $\frac{BC(t)}{C(a)}$.

Definition 31. *The* attackability, ac(AC), of an attack class A *is* $\sum\limits_{e \in A} ac(e)$.

We consider a system s with the set {`root`, `non-root`} of privilege levels; the set {`stream`, `procedure call`} of types of channels; the set {`file`, `registry`} of data item types; and the set {`unauthenticated`, `authenticated`} of access rights. We assume the following total orders.

`non-root` < `root`
`procedure call` < `stream`
`registry` < `file`
`unauthenticated` < `authenticated`

We define the functions BM, BC, BD, and C as given below. We use these numeric values to

BM(`non-root`) = 1	BC(`procedure call`) = 1	BD(`registry`) = 1	C(`anonymous`) = 1
BM(`root`) = 4	BC(`stream`) = 2	BD(`file`) = 2	C(`authenticated`) = 2

compute the attackability of a resource of s, e.g., the attackability of a method running with `root` privilege and `unauthenticated` access rights is $\frac{4}{1} = 4$, and the attackability of a `file` with `authenticated` access rights is $\frac{2}{2} = 1$.

5.2 Attack Surface Measurement

In this section, we illustrate our attack surface measurement method described in Section 4.5 by measuring the attack surface of a hypothetical IMAP server.

We consider a simple IMAP server, s. IMAP clients communicate with s using TCP sockets. The clients can login to the server, list all the emails stored on the server, fetch an email from the server, and close the communication with the server. The server, s, stores configuration information, and user email as files on the file system.

The server, s, has the set {`root`, `imapd`} of privilege levels; the set {`stream`} of channel types; the set {`file`} of data item types; and the set {`unauthenticated`, `authenticated`, `root`} of access rights.

Step 1 of our attack surface measurement method requires us to identify the set of entry points and exit points, the set of open channels, and the set of untrusted data items. In this example, we assume we are given the set {`login`, `list`, `fetch`, `close`} of entry points and exit points; the set {`port 143`, `port 993`} of open channels; and the set {`/etc/passwd`, `/etc/shadow`, `/etc/imapd.conf`, `/var/lib/imap/user` } of untrusted data items.

In Step 2, we group the above identified resources into attack classes based on their attackability. The method `login` runs with `root` privilege and `unauthenticated` access rights. The methods `list`, `fetch`, and `close` run with `imapd` privilege and `authenticated` access rights. The channels `port 143` and `port 993` have `stream` type, and `unauthenticated` access rights. The data items `/etc/shadow` and `/etc/imapd.conf` have `file` type, and `root` access rights. The data items `/etc/passwd` and `/var/lib/imap/user` have `file` type, and `authenticated` access rights. We group the resources into the set {`sac1`, `sac2`} of system attack classes, the set {`cac1`} of channel attack classes, and the set {`dac1`, `dac2`} of data attack classes. We show the attack classes in Table 1.

`sac1`	{ `login`}
`sac2`	{ `list`, `fetch`, `close`}
`cac1`	{ `port 143`, `port 993` }
`dac1`	{ `/etc/shadow`, `/etc/imapd.conf` }
`dac2`	{ `/etc/passwd`, `/var/lib/imap/user` }

Table 1: Attack Classes of s

In Step 3, we compute the total attackability of s along the three abstract dimensions of method, channel, and data item. We assign numeric values to privilege levels, channel types, data item types, and access rights levels as shown in Table 2. We compute the attackability of a resource and the

Privilege		Channel Type		Data Item Type		Access Rights	
`imapd`	1	`stream`	1	`file`	1	`unauthenticated`	1
`root`	2					`authenticated`	2
						`root`	3

Table 2: Numeric Values

attackability of an attack class as defined in Section 5.1. We show the attackability of attack classes, and the total attackability of s in Table 3.

System Attack Classes		Channel Attack Classes		Data Attack Classes	
`sac1`	2	`cac1`	2	`dac1`	0.66
`sac2`	1.5			`dac2`	1
`System Attackability`	3.5	`Channel Attackability`	2	`Data Attackability`	1.66

Table 3: Total Attackability of s

In Step 4, we obtain the attack surface of s as the triple $\langle 3.5, 2, 1.66 \rangle$.

5.3 Attack Surface Metric

In this section, we compare the attack surface of two IMAP server implementations. We assume we have two IMAP servers: s (introduced in Section 5.2) and s'. s' is similar to s except that s' has only one privilege level, i.e., `root`. Hence the methods `list`, `fetch`, and `close` of s' run with `root` privilege, and the data items `/etc/passwd` and `/var/lib/imap/user` of s' have `root` access rights. We show the attack classes of s' in Table 4. We show the attackability of attack classes and

sac1'	{ login}
sac2'	{ list, fetch, close}
cac1'	{ port 143, port 993 }
dac1'	{ /etc/shadow, /etc/imapd.conf, /etc/passwd, /var/lib/imap/user }

Table 4: Attack Classes of s'

the total attackability of s' in Table 5. The attack surface of s' is the triple $\langle 5, 2, 1.33\rangle$.

System Attack Classes		Channel Attack Classes		Data Attack Classes	
sac1	2	cac1	2	dac1	1.33
sac2	3				
System Attackability	5	Channel Attackability	2	Data Attackability	1.33

Table 5: Total Attackability of s'

Hence s has less system attackability and more data attackability compared to s'. Both s and s' have the same channel attackability. Hence s is more secure with respect to the method dimension, and less secure with respect to the data dimension compared to s'. Both s and s' are equally secure with respect to the channel dimension.

In this case, we can not determine whether one version of the IMAP server is more secure than the other by comparing their attack surface. It is not possible to decide which version of the server to use simply by looking at the attack surface measurements. The usage scenario of the IMAP server, however, helps us in making a decision. We choose version s of the IMAP server if method attackability presents higher risk compared to data attackability, else we choose version s'. For example, if the files of the IMAP server reside on a secure file system such as Microsoft's EFS [14], the attacker is more likely to attack the IMAP server using the methods rather than the files. In this case, method attackability presents higher risk compared to data attackability. Hence we choose version s of the IMAP server. On the other hand, if the email files contain sensitive information and no secure file system is available, then data attackability presents higher risk. Hence we choose version s' of the IMAP server.

6 Future Work

We discuss four possible directions for future work in this section. We plan to implement a tool that will use the entry point and exit point framework to automatically identify the set of entry points and exit points, the set of open channels, and the set of untrusted data items from the source code of a system. The tool will help us automate the attack surface measurement method described in Section 4.5. The entry points and exit points of a system act as the entry points for attacks, hence we can use the tool to identify the parts of the code where we should focus our attention to make a system more secure.

Our definition of attackability in Section 4.2 is based on intuition and experience. We plan to characterize attackability more formally in terms of our model described in section 3.1. We use a state machine to represent a system in our model. A formal definition of attackability of a system's resources will be in terms of the execution traces of the state machine representing the system.

We plan to explore the usefulness of the entry point and exit point framework in the threat modeling process [23]. Threat modeling is a systematic way of identifying and ranking the threats that are most likely to affect a system. By identifying and ranking threats, we can take appropriate countermeasures, starting with threats that present highest risk. The identification of entry points and exit points is an important step in the threat modeling process. The process, however, lacks a systematic way of performing this step. The users of the threat modeling process rely on their expertise and knowledge of a system to correctly identify the entry points and exit points. We propose to use the entry point framework to formalize this step in the threat modeling process.

Our current attack surface measurement method assumes that the source code of a system is available. Prior work on attack surface measurement ([12], [13], [19]) did not require the source code of a system. We plan to extend our current method such that we can approximate the attack surface of a system when the source code is not available.

7 Summary

We lack a good metric for measuring a software system's security (e.g., see CRA Grand Challenge # 3 [9]). In this paper, we have proposed a metric to determine whether one version of a system is more secure than another. Our security metric is based on the notion of attack surface. We view our work as a first step towards a meaningful and practical metric for security measurement. 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.

References

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

[2] 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: Systems Administration Conference (2002).

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

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

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

[6] Shawn A. Butler, *Security Attribute Evaluation Method: A Cost-Benefit Approach*, Proceedings of International Conference on Software Engineering (2002).

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

[8] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem and Dawson Engler, *An Empirical Study of Operating System Errors*, Proceedings of ACM Symposium on Operating System Design and Implementation (2001).

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

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

[11] J. Gray, *A Census of Tandem System Availability between 1985 and 1990*, IEEE Transactions on Software Engineering 39,4 (1990) p. 409-418.

[12] 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).

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

[14] Microsoft, *The Windows Server 2003 Family Encrypting File System*,
`http://www.msdn.microsoft.com/library/default.asp?url=/library/en-us/dnsecure/html/WinNETSrvr-EncryptedFileSystem.asp` (2002).

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

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

[17] I. Lee and R. Iyer, *Faults, Symptoms, and Software Fault Tolerance in the Tandem GUARDIAN Operating System*, Proceedings of International Symposium on Fault-Tolerant Computing (1993).

[18] 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.

[19] P. Manadhata and J. M. Wing, *Measuring a System's Attack Surface*, Technical Report CMU-CS-04-102, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA (2004).

[20] N. Mehta, N. Medvidovic and S. Phadke,*Towards a Taxonomy of Software Connectors*, Proceedings of International Conference on Software engineering (2000).

[21] 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.

[22] M. Sullivan and R. Chillarge, *Software Defects and their Impact on System Availability- A Study of Field Failures in Operating Systems*, Proceedings of International Symposium on Fault-Tolerant Computing (1991).

[23] F. Swiderski and W. Snyder, *Threat Modeling*, Microsoft Press (2004).

[24] 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 Annual Conference on Computer Assurance (1996).