A Formal Model for A System's Attack Surface

Pratyusa K. Manadhata **Dilsun K. Kaynar**
Jeannette M. Wing

July 2007
CMU-CS-07-144

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Practical software security metrics and measurements are essential to the development of secure software [18]. In this paper, we propose to use a software system's *attack surface measurement* as an indicator of the system's security; the larger the attack surface, the more insecure the system. We formalize the notion of a system's attack surface using an I/O automata model of the system [15] and define a quantitative measure of the attack surface in terms of three kinds of *resources* used in attacks on the system: methods, channels, and data. We demonstrate the feasibility of our approach by measuring the attack surfaces of two open source FTP daemons and two IMAP servers. Software developers can use our attack surface measurement method in the software development process and software consumers can use the method in their decision making process.

This research was sponsored by the US Army Research Office (ARO) under contract no. DAAD190210389, SAP Labs, LLC under award no. 1010751, and the Software Engineering Institute. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: Attack Surface, Attack Surface Metric, Damage Potential-Effort Ratio

1 Introduction

The importance of developing secure software is steadily increasing with the growing dependence on software in day-to-day life. Software industry has responded to the demands for improvement in software security by increasing effort to create more secure software (e.g., Microsoft's Trustworthy Computing Initiative). How can software industry determine if the effort is paying off and how can software consumers determine if industry's effort has made a difference? Security metrics and measurements are desirable to gauge progress with respect to security [5, 18, 25]; software developers can use metrics to quantify the improvement in security from one version of their software to another and software consumers can use metrics to compare alternative software that provide same functionality.

Michael Howard of Microsoft informally introduced the notion of a software system's *attack surface measurement* as an indicator of the system's security [12]. Howard, Pincus, and Wing measured the attack surfaces of seven versions of the Windows operating system [14] and Manadhata and Wing measured the attack surfaces of four versions of Linux [16]. The Windows and Linux measurement results showed that a system's attack surface measurement is an indicator of a system's perceived security and the notion of attack surface is applicable to large real world systems. Howard's measurement method, however, relies on an expert's deep understanding of a software system's features and past attacks on the system using these features (e.g., Open ports and services are features of Windows often used in attacks on Windows). Thus, we seek a *systematic* attack surface measurement method that does not require expert knowledge of a system and hence is easily applicable to a diverse range of systems. A first step toward such a method is the formalization of a system's attack surface. In this paper, our high level contributions are a formal definition of a system's attack surface and a systematic attack surface measurement method based on the formal definition.

More specifically, a system's attack surface is the set of ways in which an adversary can attack the system and potentially cause damage. We know from the past that many attacks such as buffer overflow attacks and symlink attacks require an attacker to connect to a system using the system's *channels* (e.g., sockets and pipes), invoke the system's *methods* (e.g., API), and send *data items* (e.g., input strings, URLs, and files) into the system or receive data items from the system. Hence we define a system's attack surface in terms of the system's methods, channels, and data items (henceforth collectively referred to as the system's *resources*). Not all resources, however, contribute equally to the attack surface; a resource's contribution depends on the the likelihood of the resource being used in attacks. For example, a method running with `root` privilege is more likely to be used in attacks than a method running with `non-root` privilege and hence makes a larger contribution. We measure a system's attack surface as the total contribution of the system's resources along three dimensions: methods, channels, and data. A system's attack surface measurement does not represent code quality; the measurement represents the system's "exposure" to attacks. A large attack surface measurement does not imply that a system has many vulnerabilities; instead, a larger attack surface measurement indicates that an attacker is likely to exploit the vulnerabilities present in the system with less effort and cause more damage to the system. Since a system's code is likely to contain vulnerabilities, a smaller attack surface mitigates the risk associated with the exploitation of the vulnerabilities.

We make the following technical contributions in this paper.

1. We introduce the *entry point and exit point* framework based on an I/O automata model of

a system and define the system's attack surface in terms of the framework.

2. We formally establish that with respect to the same attacker, a larger attack surface of a system leads to a larger number of *potential attacks* on the system.
3. We introduce the notions of *damage potential* and *effort* to estimate a resource's contribution to the measure of a system's attack surface and define a qualitative and a quantitative measure of the attack surface.
4. We show that our quantitative attack surface measurement method is analogous to risk modeling.

The rest of the paper is organized as follows. In Section 2, we discuss Howard's informal measurement method. In Section 3, we introduce the entry point and exit point framework culminating in a definition of the attack surface. We introduce the notions of damage potential and effort in Section 4 and define a quantitative measure of the attack surface in Section 5. In Section 6, we briefly describe the results of measuring the attack surfaces of two FTP daemons and two IMAP servers. We compare our approach to related work in Section 7 and conclude with a discussion of future work in Section 8.

2 Background and Motivation

Our work on attack surface is inspired by Howard, Pincus, and Wing's Relative Attack Surface Quotient (RASQ) measurements of Windows. Their measurement method involved two key steps: (1) identifying a set of *attack vectors*, i.e., the features of Windows often used in attacks on Windows (e.g., services running as SYSTEM) and (2) assigning weights to the attack vectors to reflect their *attackability*, i.e., the likelihood of a resource being used in attacks on Windows. A system's RASQ measurement is then the sum of the weights assigned to the attack vectors. Howard et al. used the history of attacks on Windows to identify 20 attack vectors and assign weights to them.

We later generalized Howard et al.'s method in our Linux measurements. We defined a system's *attack classes* in terms of the system's *actions* that are externally visible to its users and the system's *resources* that each action accesses or modifies. To enumerate all system actions and resources of Linux, however, is impractical; hence we again used the history of attacks on Linux to identify 14 attack classes. We had no basis to assign weights to the attack classes; hence we used the number of instances of each attack class as the measure of a system's attack surface.

The results of both the Linux and Windows measurements confirmed perceived beliefs about the relative security of the different versions and hence showed that the attack surface notion held promise. For example, the Windows measurement results showed that Windows Server 2003 has a smaller attack surface compared to Windows XP and Windows 2000; similarly, the Linux measurement results showed that Debian 3.0 has a smaller attack surface compared to Red Hat 9.0. The measurement methods, however, was ad-hoc and did not include systematic ways to identify a system's attack vectors (classes) and assign weights to them. The methods relied on our intuition, expert knowledge, and the past history of attacks on a system. Howard's method was hard to replicate in practice and we needed a systematic method that was applicable to a diverse range of systems.

These preliminary results motivated our work on formalizing a system's attack surface. We use the entry point and exit point framework to identify the relevant resources that contribute to a system's attack surface; hence our measurement method entirely avoids the need to identify a system's attack vectors (classes). We also use the notions of damage potential and effort to estimate the weights of each such resource instead of relying on the history of attacks. We demonstrate that our method is widely applicable by measuring the attack surfaces of software applications such as IMAP servers and FTP daemons.

3 I/O Automata Model

In this section, we introduce the entry point and exit point framework and use the framework to define a system's attack surface. Informally, *entry points* of a system are the ways through which data "enters" into the system from its environment and *exit points* are the ways through which data "exits" from the system to its environment. The entry points and exit points of a system act as the basis for attacks on the system.

3.1 I/O Automaton

We model a system and the entities present in its environment as I/O automata [15]. We chose I/O automata as our model for two reasons. First, our notion of entry points and exit points map naturally to the *input actions* and *output actions* of an I/O automation. Second, the *composition* property of I/O automata allows us to easily reason about the attack surface of a system in a given environment.

An I/O automaton, $A = \langle sig(A),\ states(A),\ start(A),\ steps(A)\rangle$, is a four tuple consisting of an *action signature*, $sig(A)$, that partitions a set, $acts(A)$, of *actions* into three disjoint sets, $in(A)$, $out(A)$, and $int(A)$, of *input, output* and *internal* actions, respectively, a set, $states(A)$, of *states*, a non-empty set, $start(A) \subseteq states(A)$, of *start states*, and a *transition relation*, $steps(A) \subseteq states(A) \times acts(A) \times states(A)$. An I/O automaton's environment generates input and transmits the input to the automaton using input actions. In contrast, the automaton generates output actions and internal actions autonomously and transmits output to its environment. Our model does not require an I/O automation to be *input-enabled*, i.e., unlike a standard I/O automation, input actions are not always enabled in our model. Instead, we assume that every action of an automaton is enabled in at least one state of the automaton. Notice that the composition property of the I/O automaton still holds in our model. We construct an I/O automaton modeling a complex system by composing the I/O automata modeling simpler components of the system. The composition of a set of I/O automata results in an I/O automaton.

3.2 Model

Consider 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 $T = S \setminus \{s\}$ is the set of systems excluding 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, $E_s = \langle U,\ D,\ \{s_1,\ s_2,\}\rangle$. For example, s is a web server and s_1 and s_2 are an application server and a directory server, respectively. We give an overview of our formal model in the following paragraphs.

We model every system $s \in S$ as an I/O automaton, $\langle sig(s),\ states(s),\ start(s),\ steps(s)\rangle$. We model the methods in the codebase of the system s as actions of the I/O automaton. We specify the actions using pre and post conditions: for an action, m, *m.pre* and *m.post* are the pre and post conditions of m, respectively. A state, $\mathsf{s} \in states(s)$, of s is a mapping of the state *variables* to

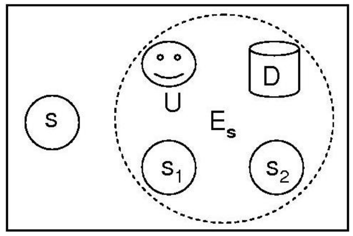

Figure 1: A system, s and its environment, E_s.

their *values*: s: $Var \to Val$. An action's pre and post conditions are first order predicates on the state variables. A state transition, $\langle \mathsf{s}, m, \mathsf{s}' \rangle \in steps(s)$, is the invocation of an action m in state s resulting in state s'. An *execution* of s is an alternating sequence of actions and states beginning with a start state and a *schedule* of an execution is a subsequence of the execution consisting only of the actions appearing in the execution.

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. Specific examples of channels are TCP/UDP sockets and named pipes. We model each channel of a system as a special state variable of the system.

We also model the user U and the data store D as I/O automata. The user U and the data store D are global with respect to the systems in S. For simplicity, we assume only one user U present in the environment. U represents the adversary who attacks the systems in S.

We model the data store D as a separate entity to allow sharing of data among the systems in S. The data store D is a set of typed *data items*. Specific examples of data items are strings, URLs, files, and cookies. For every data item $d \in D$, D has an output action, $read_d$, and an input action, $write_d$. A system s or the user U reads d from the data store through the invocation of $read_d$ and writes d to the data store through the invocation of $write_d$. To model global sharing of the data items, we add a state variable, d, to every system $s \in S$ and the user U corresponding to each data item $d \in D$. When s or U reads the data item d from the data store, the value of the data item is stored in the state variable d. Similarly, when s or U writes the data item d to the data store, the value of the state variable d is written to the data store.

3.3 Entry Points

The methods in a system's codebase that receive data from the system's environment are the system's entry points. A method of a system can receive data directly or indirectly from the environment. A method, m, of a system, s, receives data items *directly* if either i. the user U (Figure 2.a) or a system, s', (Figure 2.b) in the environment invokes m and passes data items as input to m, or ii. m reads data items from the data store (Figure 2.c), or iii. m invokes a method of a system, s', in the environment and receives data items as results returned (Figure 2.d). A method is a *direct entry point* if it receives data items directly from the environment. Few examples of the direct entry points of a web server are the methods in the API of the web server, the methods of the web server that read configuration files, and the methods of the web server that invoke the API of an application server.

In the I/O automata model, a system, s, can receive data from the environment if s has an input action, m, and an entity in the environment has a same-named output action, m. When the

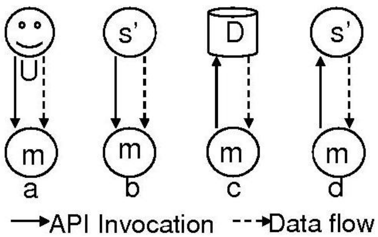

Figure 2: Direct Entry Point.

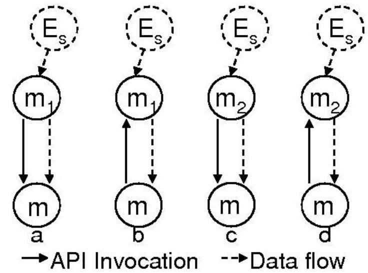

Figure 3: Indirect Entry Point.

entity performs the output action m, s performs the input action m and data is transmitted from the entity to s. We formalize the scenarios when a system, $s' \in T$, invokes m (Figure 2.b) or when m invokes a method of s' (Figure 2.d) the same way, i.e., s has an input action, m, and s' has an output action, m.

Definition 1. *A* direct entry point *of the system s is an input action, m, of s such that either i. the user U has the output action m (Figure 2.a), or ii. a system, $s' \in T$, has the output action m (Figure 2.b and Figure 2.d), or iii. the data store D has the output action m (Figure 2.c).*

A method, m, of s receives data items *indirectly* if either i. a method, m_1, of s receives a data item, d, directly, and either m_1 passes d as input to m (Figure 3.a) or m receives d as result returned from m_1 (Figure 3.b), or ii. a method, m_2, of s receives a data item, d, indirectly, and either m_2 passes d as input to m (Figure 3.c) or m receives d as result returned from m_2 (Figure 3.d). Note that we use recursion in the definition. A method is an *indirect entry point* if it receives data items indirectly from the environment. For example, a method in the API of the web server that receives login information from a user might pass the information to another method in the authentication module; the method in the authentication module is an indirect entry point.

In the I/O automata model, a system's internal actions are not visible to other systems in the environment. Hence we use a system's internal actions to formalize the system's indirect entry points. We formalize data transmission using the pre and post conditions of a system's actions. If an input action, m, of a system, s, receives a data item, d, directly from the environment, then the subsequent behavior of the system s depends on the value of d; hence d appears in the post condition of m and we write $d \in Res(m.post)$ where $Res : predicate \rightarrow 2^{Var}$ is a function such that for each post condition (or pre condition), p, $Res(p)$ is the set of resources appearing in p. Similarly, if an action, m, of s receives a data item d from another action, m_1, of s, then d appears in the post condition of m_1 and in the pre condition of m. Similar to the direct entry points, we formalize the scenarios Figure 3.a and Figure 3.b the same way and the scenarios Figure 3.c and Figure 3.d the same way. We define indirect entry points recursively.

Definition 2. *An* indirect entry point *of the system s is an internal action, m, of s such that either i. $\exists$ a direct entry point, m_1, of s such that $m_1.post \Rightarrow m.pre$ and $\exists$ a data item, d, such that $d \in Res(m_1.post) \wedge d \in Res(m.pre)$ (Figure 3.a and Figure 3.b), or ii. $\exists$ an indirect entry point, m_2, of s such that $m_2.post \Rightarrow m.pre$ and $\exists$ a data item, d, such that $d \in Res(m_2.post) \wedge d \in Res(m.pre)$ (Figure 3.c and Figure 3.d).*

The set of entry points of s is the union of the set of direct entry points and the set of indirect entry points of s.

3.4 Exit Points

The methods of a system that send data to the system's environment are the system's exit points. For example, a method that writes into a log file is an exit point. A method of a system can send data directly or indirectly into the environment. A method, m, of a system, s, sends data items *directly* if either i. the user U (Figure 4.a) or a system, s', (Figure 4.b) in the environment invokes m and receives data items as results returned from m, or ii. m writes data items to the data store (Figure 4.c), or iii. m invokes a method of a system, s', in the environment and passes data items as input (Figure 4.d).

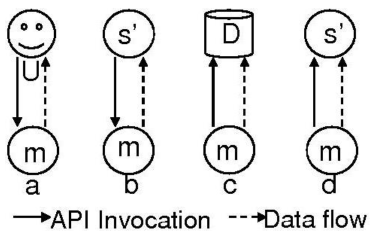

Figure 4: Direct Exit Point.

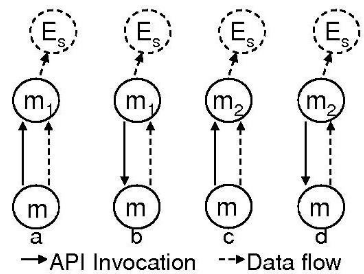

Figure 5: Indirect Exit Point.

In the I/O automata model, a system, s, can send data to the environment if s has an output action, m, and an entity in the environment has a same-named input action, m. When s performs the output action m, the entity performs the input action m and data is transmitted from s to the entity.

Definition 3. *A* direct exit point *of the system s is an output action, m, of s such that either i. the user U has the input action m (Figure 4.a), or ii. a system, $s' \in T$, has the input action m (Figure 4.b and Figure 4.d) , or iii. the data store D has the input action m (Figure 4.c).*

A method, m, of s sends data items *indirectly* to the environment if either i. m passes a data item, d, as input to a direct exit point, m_1 (Figure 5.a), or m_1 receives a data item, d, as result returned from m (Figure 5.b), and m_1 sends d directly to the environment, or ii. m passes a data item, d, as input to an indirect exit point, m_2 (Figure 5.c), or m_2 receives a data item, d, as result returned from m (Figure 5.d), and m_2 sends d indirectly to the environment. A method m of s is an *indirect exit point* if m sends data items indirectly to the environment.

Similar to indirect entry points, we formalize indirect exit points of a system using the system's internal actions. If an output action, m, sends a data item, d, to the environment, then the subsequent behavior of the environment depends on the value of d. Hence d appears in the pre condition of m and in the post condition of the same-named input action m of an entity in the environment. Note that we define indirect exit points recursively.

Definition 4. *An* indirect exit point *of the system s is an internal action, m, of s such that either i. $\exists$ a direct exit point, m_1, of s such that $m.post \Rightarrow m_1.pre$ and $\exists$ a data item, d, such that $d \in Res(m.post) \wedge d \in Res(m_1.pre)$ (Figure 5.a and Figure 5.b), or ii. $\exists$ an indirect exit point, m_2, of s such that $m.post \Rightarrow m_2.pre$ and $\exists$ a data item, d, such that $d \in Res(m.post) \wedge d \in Res(m_2.pre)$ (Figure 5.c and Figure 5.d).*

The set of exit points of s is the union of the set of direct exit points and the set of indirect exit points of s.

3.5 Channels

An attacker uses a system's channels to connect to the system and invoke a system's methods. Hence a system's channels act as another basis for attacks on the system. An entity in the environment can invoke a method, m, of a system, s, by using a channel, c, of s; hence in our I/O automata model, c appears in the pre condition of a direct entry point (or exit point), m. Every channel of s appears in the pre condition of at least one direct entry point (or exit point) of s.

3.6 Untrusted Data Items

The data store D is a collection of persistent and transient data items. The data items that are visible to both a system, s, and the user U across different executions of s are the persistent data items of s. Specific examples of persistent data items are files, cookies, database records, and registry entries. The persistent data items are shared between s and U, hence U can use the persistent data items to send (receive) data indirectly into (from) s. For example, s might read a file from the data store after U writes the file to the data store. Hence the persistent data items act as another basis for attacks on s. An *untrusted data item* of a system s is a persistent data item d such that a direct entry point of s reads d from the data store or a direct exit point of s writes d to the data store.

Definition 5. *An* untrusted data item *of a system, s, is a persistent data item, d, such that either i. $\exists$ a direct entry point, m, of s such that $d \in Res(m.post)$, or ii. $\exists$ a direct exit point, m, of s such that $d \in Res(m.pre)$.*

3.7 Attack Surface Definition

A system's attack surface is the subset of the system's resources that an attacker can use to attack the system. An attacker can use a system's entry points and exit points, channels, and untrusted data items to either send (receive) data into (from) the system to attack the system. Hence the set of entry points and exit points, the set of channels, and the set of untrusted data items are the relevant subset of resources that are part of the attack surface.

Definition 6. *Given a system, s, and its environment, E_s, s's attack surface is the triple, $\langle M, C, I\rangle$, where M is the set of entry points and exit points, C is the set of channels, and I is the set of untrusted data items of s.*

Notice that we define s's entry points and exit points, channels, and data items with respect to the given environment E_s. Hence s's attack surface, $\langle M, C, I\rangle$, is with respect to the environment E_s. We compare the attack surfaces of two similar systems (i.e., different versions of the same software or different software that provide similar functionality) along the methods, channels, and data dimensions to determine if one has a larger attack surface than another.

Definition 7. *Given an environment, $E_{AB} = \langle U, D, T\rangle$, the attack surface, $\langle M_A, C_A, I_A\rangle$, of a system, A, is larger than the attack surface, $\langle M_B, C_B, I_B\rangle$, of a system, B iff either i. $M_A \supset M_B \wedge C_A \supseteq C_B \wedge I_A \supseteq I_B$, or ii. $M_A \supseteq M_B \wedge C_A \supset C_B \wedge I_A \supseteq I_B$, or iii. $M_A \supseteq M_B \wedge C_A \supseteq C_B \wedge I_A \supset I_B$.*

Consider a system, A, and its environment, $E_A = \langle U, D, T\rangle$. We model the interaction of the system A with the entities present in its environment as parallel composition, $A||E_A$. Notice that an attacker can send data into A by invoking A's input actions and the attacker can receive data from A when A executes its output actions. Since an attacker attacks a system either by sending data into the system or by receiving data from the system, any schedule of the composition of A and E_A that contains an input action or an output action of A is a potential attack on A. We denote the set of potential attacks on s as $attacks(A)$. We show that with respect to the same attacker and operating environment, if a system, A, has a larger attack surface compared to a similar system, B, then the number of potential attacks on A is larger than B. Since A and B are similar systems, we assume both A and B have the same set of state variables and the same sets of resources except the ones appearing in the attack surfaces.

Definition 8. *Given a system, s, and its environment, $E_s = \langle U,\ D,\ T\rangle$, a potential attack on s is a schedule, β, of the composition, $P = s \,||\, U \,||\, D \,||\, (||_{t \in T}\, t)$, such that an input action (or output action), m, of s appears in β.*

Note that s's schedules may contain internal actions, but in order for a schedule to be an attack, the schedule must contain input actions or output actions.

Theorem 1. *Given an environment, $E_{AB} = \langle U,\ D,\ T\rangle$, if the attack surface, $\langle M_A, C_A, I_A\rangle$, of a system, A, is larger than the attack surface, $\langle M_B, C_B, I_B\rangle$, of a system, B, then everything else being equal $attacks(A) \supset attacks(B)$.*

Proof. (Sketch)

- Case i: $M_A \supset M_B \wedge C_A \supseteq C_B \wedge I_A \supseteq I_B$
  Without loss of generality, we assume that $M_A \backslash M_B = \{m\}$. Consider the compositions $P_A = A \,||\, U \,||\, D \,||\, \ (||_{t \in T}\, t)$ and $P_B = B \,||\, U \,||\, D \,||\, (||_{t \in T}\, t)$. Any method, $m \in M_B$, that is enabled in a state, s_B, of B is also enabled in the corresponding state s_A of A and for any transition, $\langle s_B, m, s'_B\rangle$, of P_B, there is a corresponding transition, $\langle s_A, m, s'_A\rangle$, of P_A. Hence for any schedule $\beta \in attacks(B)$, $\beta \in attacks(A)$ and $attacks(A) \supseteq attacks(B)$.

  - Case a: m is a direct entry point (or exit point) of A.
    Since m is a direct entry point (or exit point), there is an output (or input) action m of either U, D, or a system, $t \in T$. Hence there is at least one schedule, β, of P_A containing m. Moreover, β is not a schedule of P_B as $m \notin M_B$. Since β is a potential attack on A, $\beta \in attacks(A) \wedge \beta \notin attacks(B)$. Hence $attacks(A) \supset attacks(B)$.
  - Case b: m is an indirect entry point (or exit point) of A.
    Since m is an indirect entry point (or exit point) of A, there is a direct entry point (or exit point), m_A, of A such that $m_A.post \Rightarrow m.pre$ (or $m.post \Rightarrow m_A.pre$). Hence there is at least one schedule, β, of P_A such that m follows m_A (or m_A follows m) in β. Moreover, β is not an schedule of P_B as $m \notin M_B$. Since β is a potential attack on A, $\beta \in attacks(A) \wedge \beta \notin attacks(B)$. Hence $attacks(A) \supset attacks(B)$.

- Case ii: $M_A \supseteq M_B \wedge C_A \supset C_B \wedge I_A \supseteq I_B$
  Without loss of generality, we assume that $C_A \backslash C_B = \{c\}$. We know that c appears in the pre condition of a direct entry point (or exit point), $m \in M_A$. But $c \notin C_B$, hence m is never enabled in any state of B and $m \notin M_B$. Hence $M_A \supset M_B$ and from Case i, $attacks(A) \supset attacks(B)$.
- Case iii: $M_A \supseteq M_B \wedge C_A \supseteq C_B \wedge I_A \supset I_B$
  The proof is similar to case ii.

□

Theorem 1 has practical significance in the software development process. The theorem shows that if we create a newer version of a software system by adding more resources to an older version, the newer version has a larger attack surface and hence a larger number of potential attacks. Hence software developers should ideally strive towards reducing the attack surface of their software from one version to another or if adding resources to the software (e.g., adding methods to an API), then knowingly increase the attack surface.

4 Damage Potential and Effort

Not all resources contribute equally to the measure of a system's attack surface because not all resources are equally likely to be used by an attacker. A resource's contribution to a system's attack surface depends on the resource's *damage potential*, i.e., the level of harm the attacker can cause to the system in using the resource in an attack and the *effort* the attacker spends to acquire the necessary access rights in order to be able to use the resource in an attack. The higher the damage potential or the lower the effort, the higher the contribution to the attack surface. In this section, we use our I/O automata model to formalize the notions of damage potential and effort. We model the damage potential and effort of a resource, r, of a system, s, as the state variables $r.dp$ and $r.ef$, respectively.

In practice, we estimate a resource's damage potential and effort in terms of the resource's attributes. Examples of attributes are method privilege, access rights, channel protocol, and data item type. We estimate a method's damage potential in terms of the method's *privilege*. An attacker gains the same privilege as a method by using a method in an attack. For example, the attacker gains `root` privilege by exploiting a buffer overflow in a method running as `root`. The attacker can cause damage to the system after gaining `root` privilege. The attacker uses a system's channels to connect to a system and send (receive) data to (from) a system. A channel's *protocol* imposes restrictions on the data exchange allowed using the channel, e.g., a `TCP socket` allows raw bytes to be exchanged whereas an `RPC endpoint` does not. Hence we estimate a channel's damage potential in terms of the channel's protocol. The attacker uses persistent data items to send (receive) data indirectly into (from) a system. A persistent data item's *type* imposes restrictions 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`. Hence we estimate a data item's damage potential in terms of the data item's type. The attacker can use a resource in an attack if the attacker has the required *access rights*. The attacker spends effort to acquire these access rights. Hence for the three kinds of resources, i.e., method, channel, and data, we estimate the effort the attacker needs to spend to use a resource in an attack in terms of the resource's access rights.

We assume that we have a total ordering, $\succ$, among the values of each of the six attributes, i.e., method privilege and access rights, channel protocol and access rights, and data item type and access rights. In practice, we impose these total orderings using our knowledge of a system and its environment. For example, an attacker can cause more damage to a system by using a method running with `root` privilege than a method running with `non-root` privilege; hence `root` $\succ$ `non-root`. We use these total orderings to compare the contributions of resources to the attack surface. Abusing notation, we write $r_1 \succ r_2$ to express that a resource, r_1, makes a larger contribution to the attack surface than a resource, r_2.

Definition 9. *Given two resources, r_1 and r_2, of a system, A, $r_1 \succ r_2$ iff either i. $r_1.dp \succ r_2.dp \wedge r_2.ef \succ r_1.ef$, or ii. $r_1.dp = r_2.dp \wedge r_2.ef \succ r_1.ef$, or iii. $r_1.dp \succ r_2.dp \wedge r_2.ef = r_1.ef$.*

4.1 Modeling Damage Potential and Effort

In our I/O automata model, we use an action's pre and post conditions to formalize effort and damage potential, respectively. We present a parametric definition of an action, m, of a system, s, below. For simplicity, we assume that the entities in the environment connect to s using only one channel, c, to invoke m and m either reads or writes only one data item, d.

$m(MA, CA, DA, MB, CB, DB)$
  $pre : P_{pre} \wedge MA \succeq m.ef \wedge CA \succeq c.ef \wedge DA \succeq d.ef$
  $post : P_{post} \wedge MB \succeq m.dp \wedge CB \succeq c.dp \wedge DB \succeq d.dp$

The parameters MA, CA, and DA represent the method access rights, channel access rights, and data access rights acquired by an attacker so far, respectively. Similarly, the parameters MB, CB, and DB represent the benefit to the attacker in using the method m, the channel c, and the data item d in an attack, respectively. P_{pre} is the part of m's pre condition that does not involve access rights. The clause, $MA \succeq m.ef$, captures the condition that the attacker has the required access rights to invoke m; the other two clauses in the pre condition are analogous. Similarly, P_{post} is the part of m's post condition that does not involve benefit. The clause, $MB \succeq m.dp$, captures the condition that the attacker gets the expected benefit after the execution of m; the rest of the clauses are analogous.

We use the total orderings $\succ$ among the values of the attributes to define the notion of weaker (and stronger) pre conditions and post conditions. We first introduce a predicate, $\langle m_1, c_1, d_1\rangle \succ_{at} \langle m_2, c_2, d_2\rangle$, to compare the values of an attribute, $at \in \{dp, ef\}$, of the two triples, $\langle m_1, c_1, d_1\rangle$ and $\langle m_2, c_2, d_2\rangle$. We later use the predicate to compare pre and post conditions.

Definition 10. *Given two methods, m_1 and m_2, two channels, c_1 and c_2, two data items, d_1 and d_2, and an attribute, $at \in \{dp, ef\}$, $\langle m_1, c_1, d_1\rangle \succ_{at} \langle m_2, c_2, d_2\rangle$ iff either i. $m_1.at \succ m_2.at \wedge c_1.at \succeq c_2.at \wedge d_1.at \succeq d_2.at$, or ii. $m_1.at \succeq m_2.at \wedge c_1.at \succ c_2.at \wedge d_1.at \succeq d_2.at$ or iii. $m_1.at \succeq m_2.at \wedge c_1.at \succeq c_2.at \wedge d_1.at \succ d_2.at$.*

Consider two methods, m_1 and m_2. We say that m_1 has a weaker pre condition than m_2 iff $m_2.pre \Rightarrow m_1.pre$. Notice that if m_1 has a lower access rights level than m_2, i.e., $m_2.ef \succ m_1.ef$, then $\forall MA.((MA \succeq m_2.ef) \Rightarrow (MA \succeq m_1.ef))$; the rest of the clauses in the pre conditions are analogous. Hence we define the notion of weaker pre condition as follows.

Definition 11. *Given the pre condition, $m_1.pre = (P_{pre} \wedge MA \succeq m_1.ef \wedge CA \succeq c_1.ef \wedge DA \succeq d_1.ef)$, of a method, m_1, and the pre condition, $m_2.pre = (P_{pre} \wedge MA \succeq m_2.ef \wedge CA \succeq c_2.ef \wedge DA \succeq d_2.ef)$, of a method, m_2, $m_2.pre \Rightarrow m_1.pre$ if $\langle m_2, c_2, d_2\rangle \succ_{ef} \langle m_1, c_1, d_1\rangle$.*

We say that m_1 has a weaker post condition than m_2 iff $m_1.post \Rightarrow m_2.post$.

Definition 12. *Given the post condition, $m_1.post = (P_{post} \wedge MB \succeq m_1.dp \wedge CB \succeq c_1.dp \wedge DB \succeq d_1.dp)$, of a method, m_1 and the post condition, $m_2.post = (P_{post} \wedge MB \succeq m_2.dp \wedge CB \succeq c_2.dp \wedge DB \succeq d_2.dp)$, of a method, m_2, $m_1.post \Rightarrow m_2.post$ if $\langle m_1, c_1, d_1\rangle \succ_{dp} \langle m_2, c_2, d_2\rangle$.*

4.2 Attack Surface Measurement

Given two systems, A and B, if A has a larger attack surface than B (Definition 7), then everything else being equal, it is easy to see that A has a larger attack surface measurement than B. It is also possible that even though A and B both have the same attack surface, if a resource, r_A, belonging to A'a attack surface makes a larger contribution than the same-named resource, r_B, belonging to B's attack surface, then everything else being equal A has a larger attack surface measurement than B. Given the attack surface, $\langle M_A, C_A, I_A\rangle$, of a system, A, we denote the set of resources belonging to A's attack surface as $R_A = M_A \cup C_A \cup I_A$.

Definition 13. *Given an environment, $E_{AB} = \langle U, D, T\rangle$, the attack surface, $\langle M_A, C_A, I_A\rangle$, of a system, A, and the attack surface, $\langle M_B, C_B, I_B\rangle$, of a system, B, A has a larger attack surface measurement than B iff either*

1. *A has a larger attack surface than B and everything else being equal, there is a set, $R \subseteq R_A \cap R_B$, of resources such that $\forall r \in R.r_A \succ r_B$, or*
2. *$M_A = M_B \wedge C_A = C_B \wedge I_A = I_B$ and everything else being equal, there is a nonempty set, $R \subseteq R_A \cap R_B$, of resources such that $\forall r \in R.r_A \succ r_B$.*

We show that with respect to the same attacker and operating environment, if a system, A, has a larger attack surface measurement compared to a system, B, then the number of potential attacks on A is larger than B.

Theorem 2. *Given an environment, $E_{AB} = \langle U, D, T\rangle$, if the attack surface of a system A is the triple $\langle M_A, C_A, I_A\rangle$, the attack surface of of a system, B, is the triple $\langle M_B, C_B, I_B\rangle$, and A has a larger attack surface measurement than B, then $attacks(A) \supseteq attacks(B)$.*

Proof. (Sketch)

- Case 1: This is a corollary of Theorem 1.
- Case 2: $M_A = M_B \wedge C_A = C_B \wedge I_A = I_B$
  Without loss of generality, we assume that $R = \{r\}$ and $r_A \succ r_B$.

  Case i: $r_B.ef \succ r_A.ef \wedge r_A.dp \succ r_B.dp$
  From definitions 11 and 12, there is an action, $m_A \in M_A$, that has a weaker precondition and a stronger post condition than the same-named action, $m_B \in M_B$, i.e.,

$$(m_B.pre \Rightarrow m_A.pre) \wedge (m_A.post \Rightarrow m_B.post). \tag{1}$$

  Notice that any schedule of the composition P_B (as defined in the proof sketch of Theorem 1) that does not contain m_B is also a schedule of the composition P_A. Now consider a schedule, β, of P_B that contains m_B and the following sequence of actions that appear in β:..$m_1 m_B m_2$... Hence,

$$(m_1.post \Rightarrow m_B.pre) \wedge (m_B.post \Rightarrow m_2.pre). \tag{2}$$

  From equations (1) and (2), $(m_1.post \Rightarrow m_B.pre \Rightarrow m_A.pre) \wedge$ $(m_A.post \Rightarrow m_B.post \Rightarrow m_2.pre)$. Hence, $(m_1.post \Rightarrow m_A.pre) \wedge$ $(m_A.post \Rightarrow m_2.pre)$.

  That is, we can replace the occurrences of m_B in β with m_A. Hence β is also a schedule of the composition P_A and $attacks(A) \supseteq attacks(B)$.

  Case ii and Case iii: The proof is similar to Case i.

□

Theorem 2 also has practical significance in the software development process. The theorem shows that if software developers modify the values of a resource's attributes and hence modify the resource's damage potential and effort in the newer version of their software, then the attack surface measurement becomes larger and the number of potential attacks on the software increases.

5 A Quantitative Metric

In the previous section, we introduced a qualitative measure of a system's attack surface (Definition 13). The qualitative measure is an ordinal scale [8]; given two systems, we can only determine if one system has a larger attack surface measurement than another. We, however, need a quantitative measure to determine how much larger one system's attack surface measurement is than another. In this section, we introduce a quantitative measure of the attack surface; the measure is a ratio scale. We quantify a resource's contribution to the attack surface in terms of a *damage potential-effort ratio.*

5.1 Damage Potential-Effort Ratio

In the previous section, we consider a resource's damage potential and effort in isolation while estimating the resource's contribution to the attack surface. From an attacker's point of view, however, damage potential and effort are related; if the attacker gains higher privilege by using a method in an attack, then the attacker also gains the access rights of a larger set of methods. For example, the attacker can access only the methods with `authenticated` user access rights by gaining `authenticated` privilege, whereas the attacker can access methods with `authenticated` user and `root` access rights by gaining `root` privilege. The attacker is willing to spend more effort to gain a higher privilege level that enables the attacker to cause damage as well as gain more access rights. Hence we consider a resource's damage potential and effort in tandem and quantify a resource's contribution to the attack surface as a damage potential-effort ratio. The damage potential-effort ratio is similar to a cost-benefit ratio; the damage potential is the benefit to the attacker in using a resource in an attack and the effort is the cost to the attacker in using the resource.

We assume a function, der_m: method $\rightarrow \mathbb{Q}$, that maps each method to its damage potential-effort ratio belonging to the set, $\mathbb{Q}$, of rational numbers. Similarly, we assume a function, der_c: channel $\rightarrow \mathbb{Q}$, for the channels and a function, der_d: data item $\rightarrow \mathbb{Q}$, for the data items. In practice, however, we compute a resource's damage potential-effort ratio by assigning numeric values to the resource's attributes. For example, we compute a method's damage potential-effort ratio from the numeric values assigned to the method's privilege and access rights. We assign the numeric values based on our knowledge of a system and its environment; we discuss a specific method of assigning numeric values in Section 6.1.

In terms of our formal I/O automata model, the damage potential of a method, m, determines how strong the post condition of m is. m's damage potential determines the potential number of methods that m can call and hence the potential number of methods that can follow m in a schedule. The higher the damage potential, the larger the number of methods that can follow m. Similarly, m's effort determines the potential number of methods that m can follow in a schedule. The lower the effort, the larger the number of methods that m can follow. Hence the damage potential-effort ratio, $der_m(m)$, of m determines the potential number of schedules in which m can appear. Given two methods, m_1 and m_2, if $der_m(m_1) > der_m(m_2)$ then m_1 can potentially appear in more schedules (and hence more potential attacks) than m_2. Similarly, if a channel, c, (or a data item, d) appears in the pre condition of a method, m, then the damage potential-effort ratio of c (or d) determines the potential number of schedules in which m can appear. Hence we estimate a resource's contribution to the attack surface as the resource's damage potential-effort ratio.

5.2 Quantitative Attack Surface Measurement

We quantify a system's attack surface measurement along three dimensions: methods, channels, and data.

Definition 14. *Given the attack surface, $\langle M, C, I\rangle$, of a system, A, the attack surface measurement of A is the triple $\langle \sum_{m\in M} der_m(m),\ \sum_{c\in C} der_c(c),\ \sum_{d\in I} der_d(d)\rangle$.*

Our attack surface measurement method is analogous to the risk estimation method used in risk modeling [11]. A system's attack surface measurement is an indication of the system's risk from attacks on the system. In risk modeling, the risk associated with a set, E, of events is $\sum_{e\in E} p(e).C(e)$ where $p(e)$ is the probability of occurrence of an event, e, and $C(e)$ is the consequences of e. The events in risk modeling are analogous to a system's resources in our measurement method. The probability of occurrence of an event is analogous to the probability of a successful attack on the system using a resource; if the attack is not successful, then the attacker does not benefit from the attack. For example, a buffer overrun attack using a method, m, will be successful only if m has an exploitable buffer overrun vulnerability. Hence the probability, $p(m)$, associated with a method, m, is the probability that m has an exploitable vulnerability. Similarly, the probability, $p(c)$, associated with a channel, c, is the probability that the method that receives (or sends) data from (to) c has an exploitable vulnerability and the probability, $p(d)$, associated with a data item, d, is the probability that the method that reads or writes d has an exploitable vulnerability. The consequence of an event is analogous to a resource's damage potential-effort ratio. The pay-off to the attacker in using a resource in an attack is proportional to the resource's damage potential-effort ratio; hence the damage potential-effort ratio is the consequence of a resource being used in an attack. The risk along the three dimensions of the system A is the triple, $\langle \sum_{m\in M} p(m).der_m(m),$ $\sum_{c\in C} p(c).der_c(c),\ \sum_{d\in I} p(d).\ der_d(d)\rangle$, which is also the measure of A's attack surface.

In practice, however, it is difficult to predict defects in software [7] and to estimate the likelihood of vulnerabilities in software [9]. Hence we take a conservative approach in our attack surface measurement method and assume that $p(m) = 1$ for all methods, i.e., every method has an exploitable vulnerability. We assume that even if a method does not have a known vulnerability now, it might have a future vulnerability not discovered so far. We similarly assume that $p(c) = 1$ ($p(d) = 1$) for all channels (data items). With our conservative approach, the measure of a system's attack surface is the triple $\langle\ \sum_{m\in M} der_m(m),\ \sum_{c\in C} der_c(c),\ \sum_{d\in I} der_d(d)\rangle$.

Given two similar systems, A and B, we compare their attack surface measurements along each of the three dimensions to determine if one system is more secure than another along that dimension. There is, however, a seeming contradiction of our measurement method with our intuitive notion of security. For example, consider a system, A, that has a 1000 entry points each with a damage potential-effort ratio of 1 and a system, B, that has only one entry point with a damage potential-effort ratio of 999. A has a larger attack surface measurement whereas A is intuitively more secure. This contradiction is due to the presence of *extreme events*, i.e., events that have a significantly higher consequence compared to other events [11]. An entry point with a damage potential-effort ratio of 999 is analogous to an extreme event. In the presence of extreme events, the shortcomings of the risk estimation method used in the previous paragraph is well understood and the partitioned multiobjective risk method is recommended [2]. In our attack surface measurement method, however, we compare the attack surface measurements of similar systems, i.e., systems with comparable sets of resources and comparable damage potential-effort ratios of the resources; hence we do not expect extreme events such as the example shown to arise.

6 Empirical Results

In this section, we demonstrate the feasibility of our measurement method by measuring the attack surfaces of real world systems. We also briefly discuss techniques to validate our measurement method.

6.1 Attack Surface Measurement Results

We have measured the attack surfaces of two open source File Transfer Protocol (FTP) daemons that run on the Linux platform: ProFTPD 1.2.10 and Wu-FTPD 2.6.2 [17]. The ProFTP codebase contains 28K lines of C code [20] and the Wu-FTP codebase contains 26K lines of C code [10]; we only considered code specific to the FTP daemon.

As proposed by DaCosta et al. [6], we assume that a method of a system can either receive data from or send data to the system's environment by invoking specific `C` library methods (e.g., `read` and `fwrite`). We identified a set, *Input* (*Output*), of `C` library methods that a method must invoke to receive (send) data items from (to) the environment. From the call graphs of both codebases, we identified the methods that contained a call to a method in *Input* (*Output*) as the direct entry points (exit points). We could not find a source code analysis tool that enables us to identify indirect entry points or indirect exit points. We determined the privilege and access rights levels of the methods by locating the *uid-setting* system calls and the *user authentication functions* in the codebase, respectively. Statically determining the channels opened by a system and the data items accessed by the system is difficult. Hence we monitored the run time behavior of the default installations of the FTP daemons to determine the open channels and the protocol and access rights level of each such channel. Similarly, we identified the untrusted data items and the type and access rights of each such data item by runtime monitoring.

To compute a resource's damage potential-effort ratio, we imposed total orderings among the values of the six attributes and assigned numeric values in accordance to the total orderings. For example, we assumed that the attacker can cause more damage to a system with `root` privilege than `authenticated` user privilege; hence `root` $\succ$ `authenticated` in our total ordering and we assigned a higher number to `root` than `authenticated`. The numeric values were on a ratio scale and reflected the relative damage an attacker can cause to a system with different privilege levels; we assigned numeric values based on our knowledge of the FTP daemons and UNIX security. Similarly, we imposed total orderings among the access rights levels using our knowledge of UNIX security and assigned numeric values according to the total orderings. Both FTP daemons opened only `TCP` sockets and accessed data items of only `file` type; hence assigning numeric values to channel protocols and data item types was trivial. We then estimated the total contribution of the methods, channels, and data items of both FTP daemons. We show the results in Figure 6.

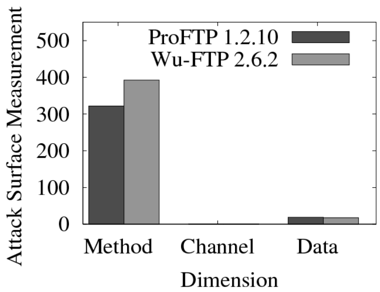

Figure 6: FTP Measurements.

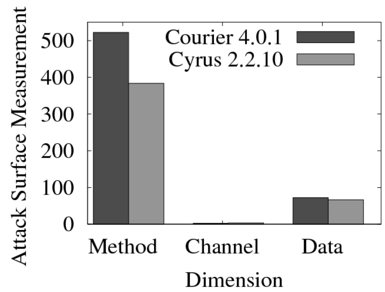

Figure 7: IMAP Measurements.

The measure of ProFTPD's attack surface is the triple $\langle 312.99,\ 1.00,\ 18.90\rangle$ and the measure of Wu-FTPD's attack surface is the triple $\langle 392.33,\ 1.00,\ 17.60\rangle$. The attack surface measurements tell us that ProFTPD is more secure along the method dimension, ProFTPD is as secure as Wu-FTPD along the channel dimension, and Wu-FTPD is more secure along the data dimension. In order to choose one FTP daemon over another, we use our knowledge of the FTP daemons and the operating environment to decide which dimension of the attack surface presents more risk, and choose the FTP daemon that is more secure along that dimension. For example, if we are concerned about privilege elevation on the host running the FTP daemon, then the method dimension presents more risk, and we choose ProFTPD over Wu-FTPD. If we are concerned about the number of open channels on the host running the FTP daemon, then the channel dimension presents more risk, and we may choose either of the daemons. If we are concerned about the safety of files stored on the FTP server, then the data dimension presents more risk, and we choose Wu-FTPD.

We also have measured the attack surfaces of two open source Internet Message Access Protocol (IMAP) servers: Courier-IMAP 4.0.1 and Cyrus 2.2.10. The Courier and Cyrus code bases contain 33K and 34K lines of C code specific to the IMAP daemon, respectively. The Courier IMAP daemon's attack surface measurement is the triple $\langle 522.00,\ 2.25,\ 72.13\rangle$ and the Cyrus IMAP daemon's attack surface measurement is the triple $\langle 383.60,\ 3.25,\ 66.50\rangle$. We show the results in Figure 7.

6.2 Validation

A key challenge in security metric research is the validation of the metric. We take a two-fold approach to validate the steps in our measurement method. We conducted an expert survey to validate the steps in our method [24].

Linux system administrators are potential users of our measurement method; hence we chose 20 experienced administrators from 10 universities, 4 corporates, and 1 government agency as the subjects of the study. The survey results show that a majority of the subjects agree with our choice of methods, channels, and data as the dimensions of the attack surface and our choice of the damage potential-effort ratio as an indicator of a resource's likelihood of being used in attacks. A majority of the participants also agreed that a method's privilege is an indicator of damage potential and a resource's access rights is an indicator of attacker effort. The survey results for protocols and data item types, however, were not statistically significant. Hence we could not conclude that channel protocols and data item types are indicators of damage potential.

We also statistically analyzed the data collected from Microsoft Security Bulletins to validate our choice of the six attributes as indicators of damage potential and effort. Each Microsoft Security Bulletin describes an exploitable vulnerability in Microsoft software and assigns a severity rating to the vulnerability. The ratings are assigned based on the impact of the exploitation on the software's users and the difficulty of exploitation [3]. In our attack surface measurement method, the impact on the users is directly proportional to damage potential and difficulty of exploitation is directly proportional to effort. Hence we expect an indicator of damage potential to be an indicator of the severity rating and to be positively correlated with the severity rating. Similarly, we expect an indicator of effort to be an indicator of the severity rating and to be negatively correlated with the severity rating. We collected data from 110 bulletins published over a period of two years from January 2004 to February 2006. The results of our analysis using Ordered Logistic Regression and two sided z-tests [27] show that the six attributes are indicators of the severity rating and positively or negatively correlated with the rating as expected. Hence we conclude that the six attributes are indicators of damage potential and effort.

While we validated the steps in our measurement method, we did not validate specific measurement results (e.g., the FTP measurement results). As part of future work, we plan to validate a system's attack surface measurement by correlating the measurement with real attacks on the system. There is, however, anecdotal evidence suggesting the effectiveness of our metric in assessing relative security of software. Our attack surface measurements show that ProFTPD is more secure than Wu-FTPD along the method dimension. The project goals mentioned on the ProFTPD website validate our measurements [21]. ProFTPD was designed and implemented from the ground up to be a secure and configurable FTP server compared to Wu-FTPD.

7 Related Work

Our attack surface measurement method differs from prior work on quantitative assessment of security in two key aspects. First, previous work assumes the knowledge of past and current vulnerabilities present in a system to assess the security of a system [1, 26, 19, 22]. Our measurement is independent of any vulnerabilities present in the system and is based on a system's inherent attributes. Our identification of all entry points and exit points encompasses all past and current vulnerabilities as well as future vulnerabilities not yet discovered or exploited. Second, while prior work makes assumptions about attacker capabilities and behavior to assess a system's security [19, 22], our measurement is based on a system's design and is independent of the attacker's capabilities and behavior.

Alves-Foss et al. use the System Vulnerability Index (SVI) as a measure of a system's vulnerability to common intrusion methods [1]. A system's SVI is obtained by evaluating factors grouped into three problem areas: system characteristics, potentially neglectful acts, and potentially malevolent acts. They, however, identify only the relevant factors of operating systems; their focus is on operating systems and not individual software applications such as FTP daemons and IMAP servers.

Voas et al. propose a minimum-time-to-intrusion (MTTI) metric based on the predicted period of time before any simulated intrusion can take place [26]. The MTTI value, however, depends on the threat classes simulated and the intrusion classes observed. In contrast, the attack surface measurement does not depend on any threat class. Moreover, the MTTI computation requires the knowledge of system vulnerabilities.

Ortalo et al. model a system's known vulnerabilities as attack state graphs and analyze the attack state graphs using Markov techniques to estimate the effort an attacker might spend to exploit the vulnerabilities; the estimated effort is a measure of the system's security [19]. Their technique, however, requires the knowledge of the vulnerabilities present in the system and the attacker's behavior.

Schneier uses attack trees to model the different ways in which a system can be attacked and to determine the cost to the attacker in the attacks [22]. Given an attacker goal, the estimated cost is a measure of the system's security. Construction of an attack tree, however, requires the knowledge of system vulnerabilities, possible attacker goals, and the attacker behavior.

8 Summary and Future Work

In this paper, we have formalized the notion of a system's attack surface and introduced a systematic method to quantitatively measure a system's attack surface. Our results are significant in practice; Mu Security's Mu-4000 Security Analyzer uses parts of the attack surface framework for security analysis [23]. Attack surface measurement is also used in a regular basis as part of Microsoft's Security Development Lifecycle [13].

In the future, we plan to extend our work in three directions. First, we are collaborating with SAP to apply our measurement method to an industrial-sized software system [4]. Second, we plan to extend our entry point and exit point framework to formalize the attack surface of a *system of systems* (e.g., the set of software applications running on a host, a network comprising of a number of hosts, and a set of web services). Third, we plan to explore the feasibility of using attack surface measurements to provide guidelines for "safe" software composition. We consider a composition of two system, A and B, to be safe iff the attack surface measurement of the composition is not greater than the sum of the attack surface measurements of A and B. We plan to identify conditions under which the composition of two given systems is safe.

References

[1] J. Alves-Foss and S. Barbosa. Assessing computer security vulnerability. *ACM SIGOPS Operating Systems Review*, 29(3):3–13, 1995.

[2] E. Asbeck and Y. Y. Haimes. The partitioned multiobjective risk method. *Large Scale Systems*, 6(1):13–38, 1984.

[3] Microsoft Corporation. Microsoft security response center security bulletin severity rating system. `http://www.microsoft.com/technet/security/bulletin/rating.mspx`.

[4] SAP Corporation. SAP - business software solutions applications and services. `http://www.sap.com/`.

[5] Computing Research Association (CRA). Four grand challenges in trustworthy computing. `http://www.cra.org/reports/trustworthy.computing.pdf`, November 2003.

[6] D. DaCosta, C. Dahn, S. Mancoridis, and V. Prevelakis. Characterizing the security vulnerability likelihood of software functions. In *Proc. of International Conference on Software Maintenance*, 2003.

[7] N. E. Fenton and M. Neil. A critique of software defect prediction models. *IEEE Transactions on Software Engineering*, 25(5), 1999.

[8] N. E. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach.* PWS Publishing Co., Boston, MA, USA, 1998.

[9] R. Gopalakrishna, E. Spafford, , and J. Vitek. Vulnerability likelihood: A probabilistic approach to software assurance. In *CERIAS Tech Report 2005-06*, 2005.

[10] The WU-FTPD Development Group. Wu-ftpd. `http://www.wu-ftpd.org/`.

[11] Y. Y. Haimes. *Risk Modeling, Assessment, and Management.* Wiley, 2004.

[12] M. 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] M. Howard and S. Lipner. *The Security Development Lifecycle.* Microsoft Press, 2006.

[14] M. Howard, J. Pincus, and J.M. Wing. Measuring relative attack surfaces. In *Proc. of Workshop on Advanced Developments in Software and Systems Security*, 2003.

[15] N. Lynch and M. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, September 1989.

[16] P. Manadhata and J. M. Wing. Measuring a system's attack surface. In *Tech. Report CMU-CS-04-102*, 2004.

[17] P. K. Manadhata, J. M. Wing, M. A. Flynn, and M. A. McQueen. Measuring the attack surfaces of two FTP daemons. In *ACM CCS Workshop on Quality of Protection*, October 2006.

[18] G. McGraw. From the ground up: The DIMACS software security workshop. *IEEE Security and Privacy*, 1(2):59–66, 2003.

[19] R. Ortalo, Y. Deswarte, and M. Kaâniche. Experimenting with quantitative evaluation tools for monitoring operational security. *IEEE Transactions on Software Engineering*, 25(5):633–650, 1999.

[20] The ProFTPD Project. The ProFTPD project home. `http://www.proftpd.org/`.

[21] The ProFTPD Project. Project goals. `http://www.proftpd.org/goals.html`.

[22] B. Schneier. Attack trees: Modeling security threats. *Dr. Dobb's Journal*, 1999.

[23] Mu Security. What is a security analyzer. `http://www.musecurity.com/solutions/overview/security.html`.

[24] W. R. Shadish, T. D. Cook, and D. T. Campbell. *Experimental and Quasi-Experimental Designs for Generalized Causal Inference.* Houghton Mifflin Company, Boston, MA, 2001.

[25] R. B. Vaughn, R. R. Henning, and A. Siraj. Information assurance measures and metrics - state of practice and proposed taxonomy. In *Proc. of Hawaii International Conference on System Sciences*, 2003.

[26] J. Voas, A. Ghosh, G. McGraw, F. Charron, and K. Miller. Defining an adaptive software security metric from a dynamic software failure tolerance measure. In *Proc. of Annual Conference on Computer Assurance*, 1996.

[27] J. M. Wooldridge. *Econometric Analysis of Cross Section and Panel Data.* The MIT Press, Cambridge, MA, USA, 2002.