An Attack Surface Metric

Pratyusa K. Manadhata

CMU-CS-08-152
November 2008

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Jeannette M. Wing, Chair
Virgil D. Gligor
Roy A. Maxion
Michael K. Reiter, University of North Carolina at Chapel Hill

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.

This research was sponsored in part by the Defense Advanced Research Project Agency (DARPA) and the Army Research Office (ARO) under contracts no. DAAD190110485 and DAAD190210389, in part by the National Science Foundation under grants no. CCR-0121547 and CNS-0433540, SAP Labs, LLC under award no. 1010751, and the Software Engineering Institute through a US government funding appropriation. The views and conclusions contained herein are those of the author 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 Metrics, Attack Surface, Attack Surface Measurement, Attack Surface Metric, Entry Point, Exit Point, Damage Potential-Effort Ratio, Metrics Validation, Software Security, Software Quality, Risk Mitigation

I dedicate this thesis to

my late uncle,
who showed me the way,

and my parents,
who have always supported me in my journey.

Abstract

Measurement of security has been a long standing challenge to the research community. Practical security measurements and metrics are critical to the improvement of software security. Hence the need for security metrics has recently become more pressing.

In this thesis, we introduce the measure of a software system's *attack surface* 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 and introduce an *attack surface metric* to measure the attack surface in a systematic manner.

Our attack surface measurement method is agnostic to a software system's implementation language and is applicable to systems of all sizes. In this thesis, we measure the attack surfaces of software implemented in `C` and `Java`. We also demonstrate that the method scales to enterprise-scale software by measuring the attack surfaces of complex SAP business applications.

Validation of security metrics is challenging and is a relatively unexplored territory. In this thesis, we conduct three exploratory empirical studies to validate our measurement method and measurements results: an expert user survey, a statistical analysis of Microsoft Security Bulletins, and an analysis of security vulnerability patches of popular open source software.

Both software developers and software consumers can use the attack surface metric. We demonstrate the use of the metric in software consumers' decision making process by comparing the attack surface measurements of two IMAP servers and two FTP daemons. Our collaboration with SAP demonstrates the use of the metric in the software development process.

Acknowledgments

I am indebted to my advisor, Jeannette M. Wing, for her guidance, encouragement, and support throughout the Ph.D. program. She taught me rigorous research, precise writing, and effective presentation skills. It has been a privilege to grow under her guidance, both as a researcher and as a person. I am forever grateful to her for the enormous amount of time she spent working with me.

I was fortunate to have Roy A. Maxion, Michael K. Reiter, and Virgil D. Gligor on my thesis committee. Their valuable feedback and insightful comments shaped my thesis to a great extent. In particular, I am grateful to Roy, who was almost like a second advisor for the last two years of the program.

I am thankful to my colleagues and coauthors: Michael Howard at Microsoft, Mark Flynn and Miles McQueen at Idaho National Laboratories, Guarav Kataria, Dilsun Kaynar, and Kymie Tan at Carnegie Mellon University, and Paul Hoffman, Yuecel Karabult, and Effrat Keren at SAP. I am also thankful to my friends at CMU and elsewhere, who have been a constant source of inspiration and encouragement.

I am thankful to the department support staff, especially Rosemary Battenfelder, Catherine Copetas, and Deborah Cavlovich, for their support throughout the Ph.D. program.

Finally, I would like to thank my family. I would be nothing without the love, sacrifices, and support of my parents, my aunt, and my late uncle. I am extremely thankful to Urmila aunty, who was instrumental in my decision to pursue a Ph.D. My wife Sargam kept me sane and believed in me when I doubted myself; this thesis might have been possible but not worth anything without her.

Contents

List of Figures

List of Tables

Chapter 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 software industry today [37, 21, 62, 97]. There is a growing demand for secure software as we are increasingly depending on software in our day-to-day life. Software industry has responded to the demands by increasing effort for creating "more secure" products and services (e.g., Microsoft's Trustworthy Computing Initiative and SAP's Software LifeCycle Security efforts). How can industry determine whether this effort is paying off and how can consumers determine whether industry's effort has made a difference? We need security metrics and measurements to gauge progress with respect to security; 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 the same functionality.

In this thesis, we formalize the notion of a system's *attack surface* and use the measure of a system's attack surface as an indicator of the system's security. Intuitively, a system's attack surface is the set of ways in which an adversary can enter the system and potentially cause damage. Hence the larger the attack surface, the more insecure the system. We also introduce an *attack surface metric* to measure a system's attack surface in a systematic manner.

Our metric does not preclude future use of the attack surface notion to define other security metrics and measurements. In this thesis, we use the attack surface metric in a *relative* manner, i.e., given two systems, we compare their attack surface measurements to indicate whether one is more secure than another with respect to the attack surface metric. Also, we use the attack surface metric to compare only *similar* systems, i.e., different versions of the same system (e.g., different versions of the Windows operating system) or different systems with similar functionality (e.g., different File Transfer Protocol (FTP) servers). We leave other contexts of use for both notions—attack surface and attack surface metric—as future work.

1.1 Motivation

Our attack surface metric is useful to both software developers and software consumers.

Software vendors have traditionally focused on improving code quality for improving software security and quality. The code quality improvement effort aims toward reducing the number of design and coding errors in software. An error causes software to behave differently from the intended behavior as defined by the software's specification; a vulnerability is an error that can be exploited by an attacker. In principle, we can use formal correctness proof techniques to identify and remove all errors in software with respect to a given specification and hence remove all its vulnerabilities. In practice, however, building large and complex software devoid of errors, and hence security vulnerabilities, remains a very difficult task. First, specifications, in particular explicit assumptions, can change over time so something that was not an error can become an error later. Second, formal specifications are rarely written in practice. Third, formal verification tools used in practice to find and fix errors, including specific security vulnerabilities such as buffer overruns, usually trade soundness for completeness or vice versa. Fourth, we do not know the vulnerabilities of the future, i.e., the errors present in software for which exploits will be developed in the future.

Software vendors have to embrace the hard fact that their software will ship with both

known and future vulnerabilities in them and many of those vulnerabilities will be discovered and exploited. They can, however, minimize the risk associated with the exploitation of these vulnerabilities. One way to minimize the risk is by reducing the attack surfaces of their software. A smaller attack surface makes the exploitation of the vulnerabilities harder and lowers the damage of exploitation and hence mitigates the security risk. As shown in Figure 1.1, the code quality effort and the attack surface reduction approach are complementary; a complete risk mitigation strategy requires a combination of both. Hence software developers can use our metric as a tool in the software development process to reduce their software's attack surfaces.

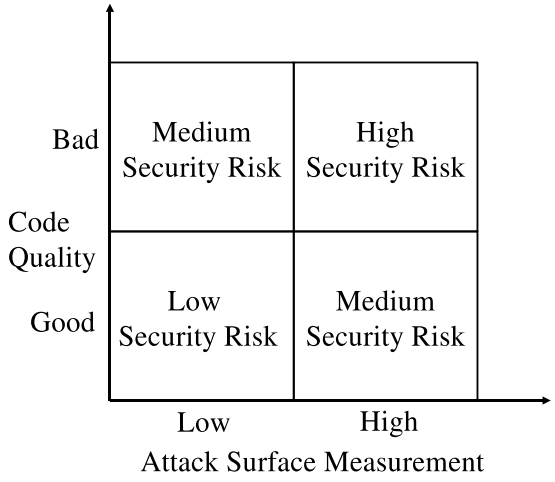

Figure 1.1: Attack Surface Reduction and Code Quality Improvement are complementary approaches for mitigating security risk and improving software security.

Software consumers often face the task of choosing one software product from a set of competing and alternative products that provide similar functionality. For example, system administrators often make a choice between different available operating systems, web servers, database servers, and FTP servers for their organization. Several factors such as ease of installation, maintenance, and use, and interoperability with existing enterprise software are relevant to software selection; security, however, is a quality that many consumers care about today and will use in choosing one software system over another. Hence software consumers can use our metric to measure the attack surfaces of alternative

software and use the measurements as a guide in their decision making process.

1.2 Attack Surface Metric

We know from the past that many attacks, e.g., exploiting a buffer overflow error, on a system take place by sending data from the system's operating environment into the system. Similarly, many other attacks, e.g., symlink attacks, on a system take place because the system sends data into its environment. In both these types of attacks, an attacker connects to a system using the system's *channels* (e.g., sockets), invokes the system's *methods* (e.g., API), and sends *data items* (e.g., input strings) into the system or receives data items from the system. An attacker can also send data indirectly into a system by using data items that are persistent (e.g., files). An attacker can send data into a system by writing to a file that the system later reads. Similarly, an attacker can receive data indirectly from the system by using shared persistent data items. Hence an attacker uses a system's methods, channels, and data items present in the system's environment to attack the system. We collectively refer to a system's methods, channels, and data items as the system's *resources* and thus define a system's attack surface in terms of the system's resources (Figure 1.2).

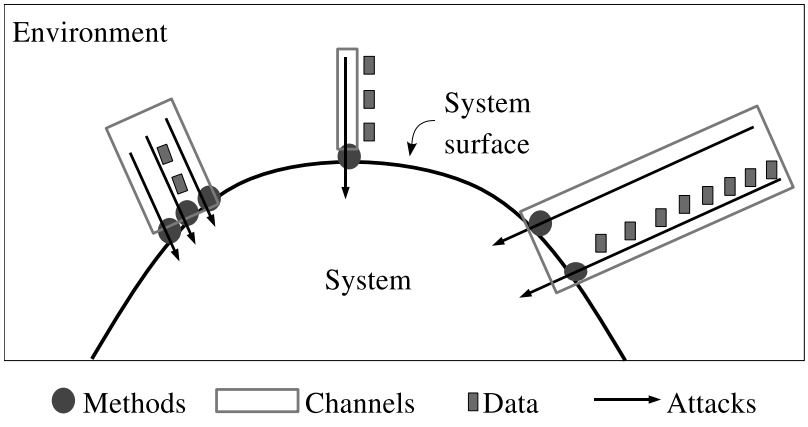

Figure 1.2: Intuitively, a system's attack surface is the subset of the system's resources (methods, channels, and data) used in attacks on the system.

Not all resources, however, are part of the attack surface and not all resources contribute equally to the attack surface measurement. In order to measure a system's attack surface, we need to identify the relevant resources that are part of the system's attack surface and to determine the contribution of each such resource to the system's attack surface measurement. A resource is part of the attack surface if an attacker can use the resource in attacks on the system; we introduce an *entry point and exit point framework* to identify these relevant resources. A resource's contribution to the attack surface measurement reflects 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. We introduce the notion of a *damage potential-effort ratio* to estimate a resource's contribution to the attack surface measurement. A system's attack surface measurement is the total contribution of the resources along the methods, channels, and data dimensions; the measurement indicates the level of damage an attacker can potentially cause to the system and the effort required for the attacker to cause such damage. Given two systems, we compare their attack surface measurements to indicate, along each of the three dimensions, whether one is more secure than the other with respect to the attack surface metric.

A system's attack surface measurement does not represent the system's code quality; hence a large attack surface measurement does not imply that the system has many vulnerabilities and having few vulnerabilities in a system does not imply a small attack surface measurement. 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, it is prudent for software developers to reduce their software's attack surfaces and for software consumers to choose software with smaller attack surfaces to mitigate security risk.

1.3 Contributions and Roadmap

We make the following key contributions in this thesis.

1. We formalize the notion of a system's attack surface in terms of our entry point and exit point framework and introduce the notion of the damage potential-effort ratio to estimate a resource's contribution to the attack surface measurement.

2. We define an abstract but systematic method to measure the attack surface. We also introduce two concrete methods to measure the attack surfaces of software implemented in `C` and `Java`.

3. We demonstrate the use of our metric in software consumers' decision making process by measuring and comparing the attack surfaces of systems routinely used in the real world. We also demonstrate the use of our metric in the software development process in our collaboration with SAP [2].

4. We demonstrate that our method scales to enterprise-scale systems by measuring the attack surfaces of complex SAP business applications.

5. We conduct three empirical studies to validate the attack surface metric. Our studies are inspired by the general validation approach used by the research community for validating software metrics.

The rest of the thesis is organized as follows. We discuss the inspiration behind this thesis research in Chapter 2. In Chapter 3, we formalize the notion of a system's attack surface using an I/O automata model of a system and its environment; we also introduce an abstract method for measuring the attack surface. In Chapter 4, we introduce a method for measuring the attack surfaces of systems implemented in `C` and apply our method to two popular open source FTP daemons and two open source IMAP servers. We discuss three empirical studies for validation of the attack surface metric in Chapter 5. In Chapter 6, we introduce a method for measuring the attack surfaces of SAP systems implemented in `Java` and apply the method to three versions of a key SAP software system. We compare our work with related work in Chapter 7. We conclude with a summary of our contributions and a discussion of possible avenues of future work in Chapter 8.

Chapter 2

Motivation

2.1 Introduction

Our thesis research on attack surface measurement is inspired by Michael Howard's Relative Attack Surface Quotient (RASQ) measurements [45]. Michael Howard of Microsoft informally introduced the notion of attack surface for the Windows operating system and Pincus and Wing further elaborated on Howard's informal notion [44]. In this chapter, we generalize Howard's measurement method and apply our method to four different versions of the Linux operating system.

We had two goals behind the Linux measurement process: (1) to understand the challenges of applying Howard's measurement method to real world software systems and (2) to demonstrate that the attack surface measurement method holds promise in spite of its drawbacks. Our subsequent work on formalizing the notion of the attack surface and defining a systematic attack surface measurement method is motivated by the preliminary results of the Linux measurement process.

2.2 Howard's Measurement Method

In this section, we describe Howard's attack surface measurement method and report the results applying the method to seven different versions of the Windows operating system.

The first step in his measurement method is the identification of the *attack vectors* of Windows, i.e., the features of Windows often used in attacks on Windows. Examples of such features are services running on Windows, open sockets, dynamic web pages, and enabled guest accounts. Not all features, however, are equally likely to be used in attacks on Windows. For example, a service running as SYSTEM is more likely to be attacked than a service running as an ordinary user. Hence the second step in Howard's method is the assignment of weights to the attack vectors to reflect their *attackability*, i.e., the likelihood of a feature being used in attacks on Windows. The weight assigned to an attack vector is the attack vector's contribution to the attack surface. The final step in Howard's method is the estimation of the total attack surface by adding the weighted counts of the attack vectors; for each instance of an attack vector, the attack vector's weight is added to the total attack surface.

Howard, Pincus, and Wing applied Howard's measurement method to seven versions of the Windows operating system [44]. They identified twenty attack vectors for Windows based on the history of attacks on Windows and then assigned weights to the attack vectors based on their expert knowledge of Windows. The measurement method was adhoc in nature and was based on intuition; the measurement results, however, confirmed perceived belief about the relative security of the seven versions of Windows. For example, Windows 2000 was perceived to have improved security compared to Windows NT [103]. The measurement results showed that Windows 2000 has a smaller attack surface than Windows NT; hence the measurements reflected the general perception. Similarly, the measurements showed that Windows Server 2003 has the smallest attack surface among the seven versions. The measurement is consistent with observed behavior in several ways, e.g., the relative susceptibility of the versions to worms such as Code Red and Nimda.

2.3 Linux Measurement Method

We applied Howard's measurement method to Linux to understand the challenges in applying the method and then to define an improved measurement method. In this section, we describe the process of applying the attack surface measurement method to four different versions of the Linux operating system.

The first step in the measurement method was the identification of the attack vectors of Linux. Howard's method did not have a formal definition of a system's attack vector. Hence there was no systematic way to identify the attack vectors of Linux. We used the history of attacks on Linux to identify fourteen attack vectors. We identified the features of Linux appearing in public vulnerability bulletins such MITRE Common Vulnerability and Exposures (CVE), Computer Emergency Response Team (CERT) Advisories, Debian Security Advisories, and Red Hat Security Advisories [71, 15, 82, 48]; these features are often used in attacks on Linux. We categorized these features into fourteen attack vectors. We show the attack vectors in Figure 2.1 and describe them in Section 2.3.1.

The second step in the measurement method was the assignment of weights to the attack vectors. Howard, Pincus, and Wing used their intuition and expertise of Windows security to assign weights in the Windows measurements. Their method, however, did not include any suggestions on how to assign weights to the attack vectors of other software systems. We could not determine a systematic way to assign weights to Linux's attack vectors. Hence we did not assign explicit numeric weights to the fourteen attack vectors identified in the first step; we assumed that each attack vector has the same weight.

The third step in the measurement method was the estimation of the attack surface measurement by adding the weighted counts of the attack vectors. Since we did not assign any weights to the attack vectors of Linux, we compared the four versions of Linux with respect to their attack vectors. We counted the number of instances of each attack vector for the four versions of Linux and compared the numbers to get a relative measure of the attack surfaces of the four versions. We describe our measurement results in Section 2.3.2

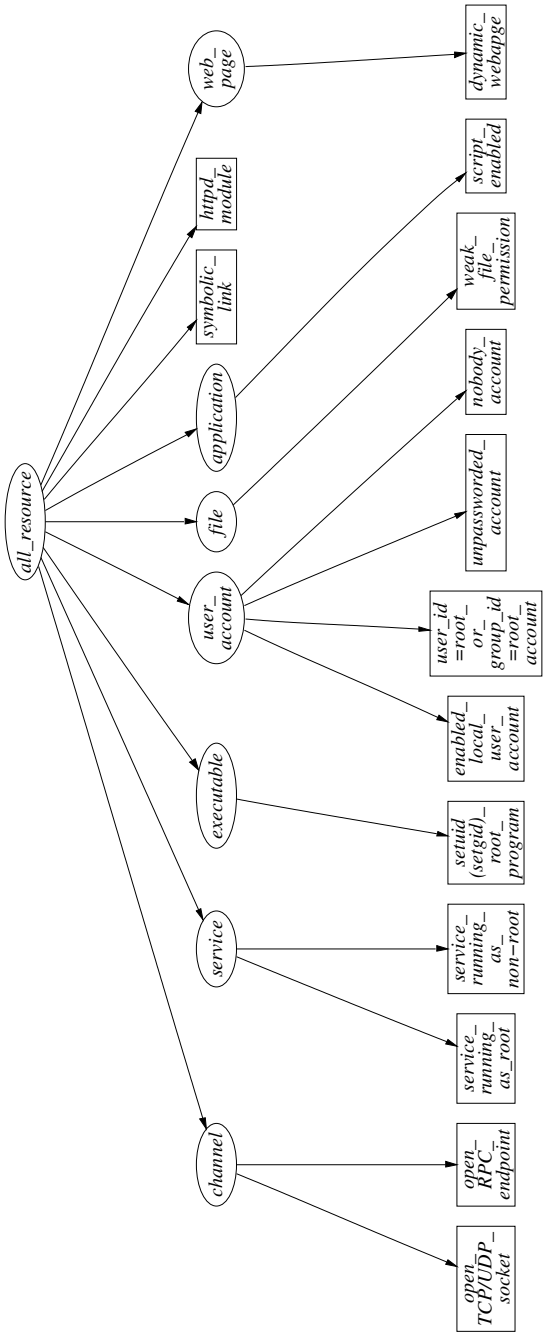

Figure 2.1: Linux Attack Vectors: the rectangular boxes represent the attack vectors.

2.3.1 Linux Attack Vectors

In this section, we describe the fourteen attack vectors of Linux. For each attack vector, we also give an example of an attack on systems running Linux using the attack vector.

open_TCP/UDP_socket: The services running on a Linux system open TCP/UDP sockets and listen for client requests on them. Multiple sockets can be opened by a service and multiple services can share the same socket. Many attacks on Linux use open TCP/UDP sockets. For example, CVE-2001-0309 describes an attack involving open sockets. The `inetd` daemon does not properly close sockets for internal services such as `daytime` and `echo`; hence an attacker can cause a denial-of-service attack by opening a series of connections to these services.

open_remote_procedure_call_(RPC)_endpoint: The RPC servers running on a Linux system register remotely accessible handlers for RPC. An attacker can invoke these RPC end points remotely and execute code on the system. For example, as described in CVE-2002-0391, a remote attacker can exploit an integer overflow vulnerability in the `SunRPC xdr_array` function to execute arbitrary code on a system.

service_running_as_root: Many daemons run as background processes on a typical Linux system to provide services to the users. The daemons run with both `root` and `non-root` privilege. Examples of daemons running with `root` privilege are `crond` and `telnetd`. Many attacks on Linux target the daemons running with `root` privilege. For example, CVE-1999-0192 describes a buffer overflow vulnerability in the `telnetd` daemon that a remote attacker can exploit to gain `root` privilege on a system.

service_running_as_non-root: Examples of services running with `non-root` privilege are `portpmap` and `rpc.statd`. CVE-2000-0666 describes a format string vulnerability in `rpc.statd` that a remote attacker can exploit to execute arbitrary code on a system.

setuid_(setgid)_root_program: Setuid root programs are executables on a Linux system that are owned by `root` and execute with the `root` privilege instead of the privilege of the user who invokes the executables. Setuid root programs are popular targets of attacks because of their elevated privilege. For example, CVE-2000-0949 describes a

heap overflow vulnerability in the `traceroute` setuid root program that local users can exploit to execute arbitrary commands on a system.

enabled_local_user_account: A typical Linux system has many local user accounts on it. Many attacks on Linux systems can be carried out only by local users. Hence a remote attacker obtains local user privileges on a system and then uses the privilege as a stepping stone to launch further attacks on the system. For example, CVE-1999-0130 describes a coding error in `Sendmail` that local users can exploit to gain `root` privilege on a system.

user_id=root_or_group_id=root_account: Many user accounts on a Linux system have their `user id` or `group id` set to `root (0)`. These accounts are popular targets of attacks on Linux systems due to their enhanced privilege. For example, CVE-2002-0875 describes a vulnerability in the way the `fam` daemon handles primary groups. Unprivileged users can exploit the vulnerability to discover a list of files accessible only by the `root` group.

unpassworded_account: An unpassworded account is an user account on a Linux system that has its `password` set to `blank`. A remote attacker can easily obtain local user privilege on a system by using an unpassworded account. CVE-1999-0502 describes the presence of an user account on a system with default, null, blank, or a missing password as a vulnerability in the system.

nobody_account: `Nobody` is a special user account on a Linux system created for running daemons on the system. Not all Linux systems, however, have a `nobody` account. CVE-2002-0424 describes a vulnerability in `efingerd` daemon running as `nobody` that local users can exploit to obtain `nobody` privilege on a system.

weak_file_permission: Many attacks on a Linux system use files on the system that grant full access rights to all users on the system. For example, CVE-2001-1322 describes a vulnerability in the `xinted` daemon that runs with a default `umask (0)`. Local users can exploit the vulnerability to read and modify files created by the applications that run under `xinted` and do not set their safe `umask`.

script_enabled: Many applications running on a Linux system are enabled to execute scripts on the system. For example, browsers are enabled to execute downloaded scripts

and email clients are enabled to execute scripts in email attachments. Many attacks on Linux target the applications that can execute scripts. For example, CVE-2001-0745 describes a vulnerability in the `Netscape` browser that a remote attacker can exploit to obtain sensitive user information via `Javascript`.

symbolic_link: A `symbolic link (symlink)` is a special file type on a Linux system that contains a reference to another file on the system. If an application running as `root` creates files in the `/tmp` directory without checking for a `symlink`, then an attacker can create a `symlink` in `/tmp` before the application starts and hence can write to sensitive files. For example, CVE-2000-0728 describes a `symlink` vulnerability in the `xpdf` PDF viewer that local users can exploit to overwrite arbitrary files.

httpd_module: An `httpd module` is a component that extends the functionality of a web server running on a Linux system. Examples of `httpd modules` of the popular `apache` web server are `mod_auth`, `mod_access`, and `mod_perl`. The `httpd modules` are the targets of many attacks on Linux. For example, CVE-2003-0789 describes a vulnerability in the handling of Common Gateway Interface (CGI) redirect paths in the `mod_cgid` module of `apache`. An attacker can exploit the vulnerability to view sensitive information.

dynamic_web_page: The contents of dynamic web pages change in response to different client requests, and under different contexts and conditions. Examples of dynamic web pages served by a web server running on a Linux system are dynamic HTML (DHTML) files, perl scripts, PHP scripts, and Java Server Pages (JSP). The dynamic web pages are popular targets of attacks on Linux systems. For example, CVE-1999-0058 describes a buffer overflow vulnerability in the `php.cgi` dynamic web page that an attacker can exploit to obtain `shell` access on a system.

2.3.2 Results

We measured the attack surfaces of the following four versions of the Linux operating system and compared their attack surface measurements.

- *Debian* is a Debian GNU/Linux 3.0r1 distribution obtained from Debian's web site [26].
- *RH Default* is a Red Hat 9.0 distribution obtained from RedHat's web site [49].
- *RH Facilities* is a customized Red Hat 9.0 Linux distribution installed by the Computing Facilities of the School of Computer Science at Carnegie Mellon University [96].
- *RH Used* is an instance of *RH Facilities* after three months of use by a graduate student.

We measured the attack surfaces of *Debian*, *RH Default*, and *RH Facilities* the very day each system was installed. We did not modify any of the systems in any manner after their installation. We measured the attack surface of *RH Used* after three months of use. We counted the number of instances of each attack vector for the four versions and compared the number of instances. The results of our measurements are shown in Table 2.1.

We did not install any web server on the system running *Debian* and *RH Default* since the system running *RH Facilities* did not have a web server installed. Hence we did not count the numbers of instances of the *httpd_module* and *dynamic_web_page* attack vectors (Rows 13 and 14 of Table 2.1). We also did not count the number of instances of the *symbolic_link* attack vector as it is impractical to determine whether the programs running as `root` check for `symlinks` before opening temporary files (Row 12 of Table 2.1).

Our measurements enable us to compare the security of the four versions of Linux in three different ways.

- Default comparison: We compared the attack surfaces of *Debian* and *RH Default* to measure the relative security of different versions of Linux.
- Customized usage-based comparison: We compared the attack surfaces of *RH Default* and *RH Facilities* to observe the change in Linux's security level due to customization.

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

Table 2.1: Attack surface measurement results (RHD = RH Default, RHF= RH Facilities, RHU = RH Used).

- Time-based comparison: We compared the attack surfaces of *RH Facilities* and *RH Used* to monitor the change in Linux's security level over time.

Debian vs. RH Default

As shown in Table 2.1, *RH Default* has higher counts in each of five attack vectors, *Debian* has a higher count in one attack vector, and both have the same counts in each of five attack vectors. Hence the attack surface exposure of Red Hat is greater than that of Debian. We believe that even though both versions provide similar functionality, design choices play an important role in making Debian's attack surface smaller than Red Hat's. Debian is perceived to be a more secure operating system and the perception is reflected in our

measurement.

RH Default vs. RH Facilities

As shown in Table 2.1, *RH Facilities* has higher counts in each of seven attack vectors, *RH Default* has higher counts in one attack vector, and both have the same counts in each of three attack vectors. The attack surface exposure of the facilities distribution is more than that of the default distribution.

The facilities distribution is customized to make it more useful than the default distribution. For example, the facilities distribution has the `AFS` file system installed; it also has `lclaadmd, kopshell,` and `terad` services installed for remote management and network backup. These features increase the counts of the *open_TCP/UDP_socket*, *service_running_as_root*, and *enabled_local_user_account* attack vectors. Our results show that the attack surface exposure has increased with customization.

RH Facilities vs. RH Used

As shown in Table 2.1, *RH Used* has higher counts in each of four attack vectors and both have the same counts in each of seven attack vectors. The used version's attack surface exposure is greater than the initially installed version. The three-month use of the system increased the counts of the *open_TCP/UDP_socket*, *service_running_as_root*, *enabled_local_user_account*, and *weak_file_permission* attack vectors. Our results show that the attack surface exposure has increased over time and usage.

2.4 Discussion

The results of both the Windows and the Linux measurements confirm perceived beliefs about the relative security of the different versions. The measurement method, however, has several shortcomings.

1. The measurement method does not have formal definitions of a system's attack surface and attack vectors; hence there is no systematic method to identify the attack vectors. Also, there is no systematic way to assign weights to the attack vectors.

2. The attack vectors of both Windows and Linux were identified based on the history of attacks on Windows and Linux, respectively. The identification process was done manually. Hence we cannot determine whether we identified all the attack vectors and whether the attack vectors were mutually exclusive.

3. The measurement method requires a security expert (e.g., Mike Howard for Windows) to assign weights to the attack vectors. Non-experts can not use the method easily.

4. The measurement method is focused on measuring the attack surfaces of operating systems and cannot be generalized to measure the attack surfaces of other software systems such as web servers, IMAP servers, and software applications.

Our thesis research on defining a *systematic* attack surface measurement method is motivated by the shortcomings in Howard's measurement method. In the thesis, we use the entry point and exit point framework to identify the relevant resources that contribute to a system's attack surface and we use the notion of the damage potential-effort ratio to estimate the weights of each such resource. Our attack surface measurement method entirely avoids the need to identify a system's attack vectors.

Our measurement method does not require a security expert; hence it can be used by software developers and software consumers without any security expertise. Furthermore, our method does not focus on operating systems and can be applied to a wide verity of software systems. We also have automated as many steps of our method as possible. For example, the identification of the relevant resources that contribute to the attack surface is completely automated. We provide detailed guidelines to the users to carry out the manual steps in our method in a systematic manner.

Chapter 3

A Formal Model for a System's Attack Surface

3.1 Introduction

In this chapter, we formalize the notion of a system's attack surface using an I/O automata model of the system and its environment [58]. We introduce the *entry point and exit point framework* based on the I/O automata model and define a system's attack surface in terms of the framework (3.2). We also introduce the notions of *damage potential* and *effort* to estimate a resource's contribution to the measure of a system's attack surface (3.3). We define a *qualitative* measure and a *quantitative* measure of the attack surface and introduce an *abstract method* to quantify the attack surface (3.4).

3.2 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. Many attacks on software systems require an attacker either to send data into a system or to receive data from a system; hence the entry points and the exit points of a system act as the basis for attacks on the system.

3.2.1 I/O Automaton

We model a system and the entities present in its environment as I/O automata [58]. 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 reachable state of the automaton.

We construct an I/O automaton modeling a complex system by *composing* the I/O automata modeling simpler components of the system. When we compose a set of automata, we identify same-named actions of different automata; we identify an output action, m, of an automaton with the input action m of each automaton having m as an input action. When an automaton having m as an output action performs m, all automata having m as an input action perform m simultaneously. The composition of a set of I/O automata results in an I/O automaton.

3.2.2 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 3.1 shows a system, s, and its environment, $E_s = \langle U, D, \{s_1, s_2, \}\rangle$. For example, s could be a web server and s_1 and s_2 could be an application server and a directory server, respectively.

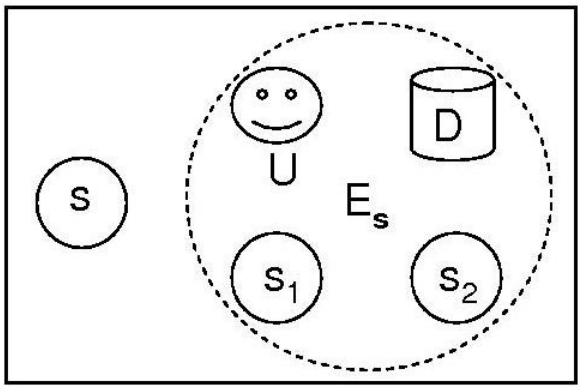

Figure 3.1: A system, s, and its environment, E_s.

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 their *values*: s: $Var \rightarrow Val$. 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, corresponding to each data item $d \in D$, we add a state variable, d, to every system, $s \in S$, and the user U. When the system s (or U) reads the data item d from the data store, the value of the data item is written to the state variable d of s (or U). Similarly, when s (or U) writes the data item d to the data store, the value of the state variable d of s (or U) is written to the data item d of the data store.

3.2.3 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 3.2.a) or a system, s', (Figure 3.2.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 3.2.c), or (iii.) m invokes a method of a system, s', in the environment and receives data items as results returned (Figure 3.2.d). A method is a *direct entry point* if it receives data items directly from the environment. 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.

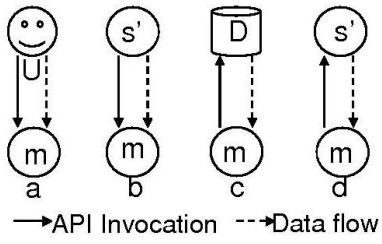

Figure 3.2: Direct Entry Point.

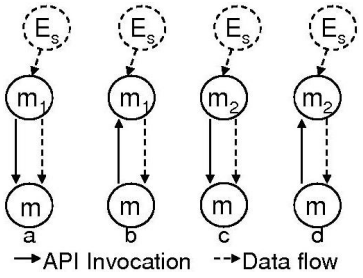

Figure 3.3: Indirect Entry Point.

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 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 3.2.b) or when m invokes a method of s' (Figure 3.2.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 3.2.a), or (ii.) a system, $s' \in T$, has the output action m (Figure 3.2.b and Figure 3.2.d), or (iii.) the data store D has the output action m (Figure 3.2.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 3.3.a) or m receives d as result returned from m_1 (Figure 3.3.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 3.3.c) or m receives d as result returned from m_2 (Figure 3.3.d). 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 API is a direct entry point and 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 an I/O automaton'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 3.3.a and Figure 3.3.b the same way and the scenarios Figure 3.3.c and Figure 3.3.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.) $\exists$ 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 3.3.a and Figure 3.3.b), or (ii.) $\exists$ indirect entry point, m_2, of s such that $m_2.post \Rightarrow m.pre$ and $\exists$ data item, d, such that $d \in Res(m_2.post) \wedge d \in Res(m.pre)$ (Figure 3.3.c and Figure 3.3.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.

3.2.4 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 3.4.a) or a system, s', (Figure 3.4.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 3.4.c), or (iii.) m invokes a method of a system, s', in the environment and passes data items as input (Figure 3.4.d).

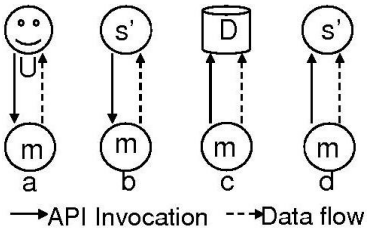

Figure 3.4: Direct Exit Point.

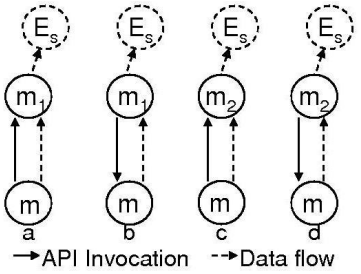

Figure 3.5: Indirect Exit Point.

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 3.4.a), or (ii.) a system, $s' \in T$, has the input action m (Figure 3.4.b and Figure 3.4.d) , or (iii.) the data store D has the input action m (Figure 3.4.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 3.5.a), or m_1 receives a data item, d, as result returned from m (Figure 3.5.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 3.5.c), or m_2 receives a data item, d, as result returned from m (Figure 3.5.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 an I/O automaton'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. Again 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 3.5.a and Figure 3.5.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 3.5.c and Figure 3.5.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.

3.2.5 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, i.e., $c \in Res(m.pre)$. In our model, every channel of s must appear in the pre condition of at least one direct entry point (or exit point) of s. Similarly, at least one channel must appear in the pre condition of every direct entry point (or direct exit point).

3.2.6 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 $d \in Res(m.pre)$.*

Notice that an attacker sends (receives) the transient data items directly into (from) s by invoking s's direct entry points (direct exit points). Since the direct entry points (direct exit points) of s act as a basis for attacks on s, we do not consider the transient data items as a different basis for attacks on s. The transient data items are untrusted data items; they are, however, already "counted" in our definition of direct entry points and direct exit points.

3.2.7 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 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, $\langle M^{E_s}, C^{E_s}, I^{E_s} \rangle$, where M^{E_s} is the set of entry points and exit points, C^{E_s} is the set of channels, and I^{E_s} 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, $\langle M^{E_s}, C^{E_s}, I^{E_s} \rangle$, 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 with respect to the same environment to determine if one has a larger attack surface than another.

Definition 7. *Given an environment, $E = \langle U, D, T \rangle$, the attack surface, $\langle M_A^E, C_A^E, I_A^E \rangle$, of a system, A, is larger than the attack surface, $\langle M_B^E, C_B^E, I_B^E \rangle$, of a system, B iff either*

(i.) $M_A^E \supset M_B^E \wedge C_A^E \supseteq C_B^E \wedge I_A^E \supseteq I_B^E$*, or (ii.)* $M_A^E \supseteq M_B^E \wedge C_A^E \supset C_B^E \wedge I_A^E \supseteq I_B^E$*, or (iii.)* $M_A^E \supseteq M_B^E \wedge C_A^E \supseteq C_B^E \wedge I_A^E \supset I_B^E$.

3.2.8 Relation between Attack Surface and Potential Attacks

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 $attacks(A)$.

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 at least one input action or output action.

We model an attacker by a set of attacks in our I/O automata model. In other models of security, e.g., for cryptography, an attacker is modeled not just by a set of attacks but also by its power and privilege [36]. Examples of an attacker's power and privilege are the attacker's skill level (e.g., script kiddies, experts, and government agencies) and the attacker's resources (e.g., computing power, storage, and tools). We, however, do not model the attacker's power and privilege in our I/O automata model. Hence our notion of attack surface is independent of the attacker's power and privilege and is dependent only on a system's design and inherent properties.

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.

Theorem 1. *Given an environment, $E = \langle U, D, T \rangle$, if the attack surface, $\langle M_A^E, C_A^E, I_A^E \rangle$, of a system, A, is larger than the attack surface, $\langle M_B^E, C_B^E, I_B^E \rangle$, of a system, B, then the rest of the resources of A and B being equal $attacks(A) \supset attacks(B)$.*

Proof. (Sketch)

- Case i: $M_A^E \supset M_B^E \wedge C_A^E \supseteq C_B^E \wedge I_A^E \supseteq I_B^E$
  Without loss of generality, we assume that $M_A^E \backslash M_B^E = \{m\}$. Consider the compositions $P_A = A \,||\, U \,||\, D \,||\, (||_{t \in T}\, t)$ and $P_B = B \,||\, U \,||\, D \,||\, (||_{t \in T}\, t)$. Any method, $m \in M_B^E$, 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 $m \notin M_B^E$. 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 $m \notin M_B^E$. 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: $M_A^E \supseteq M_B^E \wedge C_A^E \supset C_B^E \wedge I_A^E \supseteq I_B^E$
  Without loss of generality, we assume that $C_A^E \backslash C_B^E = \{c\}$. We know that c appears

in the pre condition of a direct entry point (or exit point), $m \in M_A^E$. But $c \notin C_B^E$, hence m is never enabled in any state of B and $m \notin M_B^E$. Hence $M_A^E \supset M_B^E$ and from Case i, $attacks(A) \supset attacks(B)$.

- Case iii: $M_A^E \supseteq M_B^E \wedge C_A^E \supseteq C_B^E \wedge I_A^E \supset I_B^E$
  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, then assuming all resources are counted equally (see Secion 3.3), the newer version has a larger attack surface and hence a larger number of potential attacks. 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.

3.3 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. Our estimation method is a specific instantiation of our general measurement framework. Our estimation of damage potential includes only technical impact (e.g., privilege elevation) and not business impact (e.g., monetary loss) though our framework does not preclude this generality. We do not make any assumptions about the attacker's capabilities or resources in estimating damage potential or effort.

We estimate a method's damage potential in terms of the method's *privilege*. An attacker gains the privilege of a method by using the 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.) $r_1.dp \succ r_2.dp \wedge r_2.ef = r_1.ef$.*

Definition 10. *Given two resources, r_1 and r_2, of a system, A, $r_1 \succeq r_2$ iff either (i.) $r_1 \succ r_2$ or (ii.) $r_1.dp = r_2.dp \wedge r_2.ef = r_1.ef$.*

3.3.1 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 highest 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. R_{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, R_{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 11. *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 $(m_1.R_{pre} = m_2.R_{pre}) \wedge (m_2.pre \Rightarrow m_1.pre)$. We only compare the parts of the pre conditions involving the access rights and assume that the rest of the pre conditions are the same for both m_1 and m_2. Notice that if m_1 has a lower access rights level than m_2, i.e., $m_2.ef \succ m_1.ef$, then for all access rights levels 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 12. *Given the pre condition, $m_1.pre = (R_{pre} \wedge MA \succeq m_1.ef \wedge CA \succeq c_1.ef \wedge DA \succeq d_1.ef)$, of a method, m_1, and the pre condition, $m_2.pre = (R_{pre} \wedge MA \succeq m_2.ef \wedge CA \succeq c_2.ef \wedge DA \succeq d_2.ef)$, 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 $(m_1.R_{post} = m_2.R_{post}) \wedge (m_1.post \Rightarrow m_2.post)$.

Definition 13. *Given the post condition, $m_1.post = (R_{post} \wedge MB \succeq m_1.dp \wedge CB \succeq c_1.dp \wedge DB \succeq d_1.dp)$, of a method, m_1 and the post condition, $m_2.post = (R_{post} \wedge MB \succeq m_2.dp \wedge CB \succeq c_2.dp \wedge DB \succeq d_2.dp)$, of a method, m_2, $m_1.post \Rightarrow m_2.post$ if $\langle m_1, c_1, d_1\rangle \succ_{dp} \langle m_2, c_2, d_2\rangle$.*

3.3.2 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, $A.r$, belonging to A'a attack surface makes a larger contribution than the same-named resource, $B.r$, belonging to B's attack surface, then everything else being equal A has a larger attack surface measurement than B.

Given the attack surface, $\langle M_A^E, C_A^E, I_A^E \rangle$, of a system, A, we denote the set of resources belonging to A's attack surface as $R_A = M_A^E \cup C_A^E \cup I_A^E$. Note that from Definition 7, if A has a larger attack surface than B, then $R_A \supset R_B$.

Definition 14. *Given an environment, $E = \langle U, D, T \rangle$, the attack surface, $\langle M_A^E, C_A^E, I_A^E \rangle$, of a system, A, and the attack surface, $\langle M_B^E, C_B^E, I_B^E \rangle$, of a system, B, A has a larger attack surface measurement than B ($A \gg B$) iff either*

1. *A has a larger attack surface than B (i.e., $R_A \supset R_B$) and $\forall r \in R_B.\, A.r \succeq B.r$, or*
2. *$M_A^E = M_B^E \wedge C_A^E = C_B^E \wedge I_A^E = I_B^E$ (i.e., $R_A = R_B$) and there is a nonempty set, $\mathcal{R}_{\mathcal{AB}} \subseteq R_B$, of resources such that $\forall r \in \mathcal{R}_{\mathcal{AB}}.\, A.r \succ B.r$ and $\forall r \in (R_B \setminus \mathcal{R}_{\mathcal{AB}}).\, A.r = B.r$.*

From Definitions 7 and 14, $\gg$ is transitive. For example, given three systems, A, B, and C, if A has a larger attack surface measurement than B and B has a larger attack surface measurement than C, then A has a larger attack surface measurement than C.

Theorem 2. *Given an environment, $E = \langle U, D, T \rangle$, the attack surface, R_A, of a system, A, the attack surface, R_B, of a system, B, and the attack surface, C, of a system, R_C, if $A \gg B$ and $B \gg C$, then $A \gg C$.*

Proof. (Sketch) From Definition 14, A's attack surface measurement can be larger than B's in two different ways. Similarly, B's attack surface measurement can be larger than C's in two different ways. Hence we consider four different cases in proving the theorem.

- Case 1: $R_A \supset R_B$ and $\forall r \in R_B.\, A.r \succeq B.r$.

- Case 1.1: $R_B \supset R_C$ and $\forall r \in R_C.\, B.r \succeq C.r$.
  Since $R_A \supset R_B$ and $R_B \supset R_C$, $R_A \supset R_C$. Also, since $R_B \supset R_C$ and $\forall r \in R_B.\, A.r \succeq B.r$, $\forall r \in R_C.\, A.r \succeq B.r$. From the assumptions of Case 1.1, $\forall r \in R_C.\, B.r \succeq C.r$. Hence $\forall r \in R_C.\, A.r \succeq B.r \succeq C.r$. Hence $A \gg C$.
- Case 1.2: $R_B = R_C$ and there is a nonempty set, $\mathcal{R}_{\mathcal{BC}} \subseteq R_C$, of resources such that $\forall r \in \mathcal{R}_{\mathcal{BC}}.\, B.r \succ C.r$ and $\forall r \in (R_C \setminus \mathcal{R}_{\mathcal{BC}}).\, B.r = C.r$.
  Since $R_A \supset R_B$ and $R_B = R_C$, $R_A \supset R_C$. Consider a resource, $r \in R_C$. From the assumptions of Case 1.2, if $r \in \mathcal{R}_{\mathcal{BC}}$, then $B.r \succ C.r$, and if $r \in (R_C \setminus \mathcal{R}_{\mathcal{BC}})$, then $B.r = C.r$. Hence $\forall r \in R_C.\, B.r \succeq C.r$. Also, from the assumptions of Case 1, $\forall r \in R_B.\, A.r \succeq B.r$. Since $R_B = R_C$, $\forall r \in R_C.\, A.r \succeq B.r \succeq C.r$. Hence $A \gg C$.

- Case 2: $R_A = R_B$ and there is a nonempty set, $\mathcal{R}_{\mathcal{AB}} \subseteq R_B$, of resources such that $\forall r \in \mathcal{R}_{\mathcal{AB}}.\, A.r \succ B.r$ and $\forall r \in (R_B \setminus \mathcal{R}_{\mathcal{AB}}).\, A.r = B.r$.

  - Case 2.1: $R_B \supset R_C$ and $\forall r \in R_C.\, B.r \succeq C.r$.
    The proof is similar to Case 1.2.
  - Case 2.2: $R_B = R_C$ and there is a nonempty set, $\mathcal{R}_{\mathcal{BC}} \subseteq R_C$, of resources such that $\forall r \in \mathcal{R}_{\mathcal{BC}}.\, B.r \succ C.r$ and $\forall r \in (R_C \setminus \mathcal{R}_{\mathcal{BC}}).\, B.r = C.r$.
    Since $R_A = R_B$ and $R_B = R_C$, $R_A = R_C$. Consider the set, $\mathcal{R}_{\mathcal{AC}} = \mathcal{R}_{\mathcal{AB}} \cup \mathcal{R}_{\mathcal{BC}}$, of resources. We shall prove that $\forall r \in \mathcal{R}_{\mathcal{AC}}.\, A.r \succ C.r$. Consider a resource, $r \in \mathcal{R}_{\mathcal{AC}}$. If $r \in \mathcal{R}_{\mathcal{AB}} \cap \mathcal{R}_{\mathcal{BC}}$, then $A.r \succ B.r \succ C.r$. If $r \in \mathcal{R}_{\mathcal{AB}} \setminus \mathcal{R}_{\mathcal{BC}}$, then $A.r \succ B.r = C.r$. Similarly, if $r \in \mathcal{R}_{\mathcal{BC}} \setminus \mathcal{R}_{\mathcal{AB}}$, then $A.r = B.r \succ C.r$. Hence $\forall r \in \mathcal{R}_{\mathcal{AC}}.\, A.r \succ C.r$. Also, from the assumptions of Case 2 and Case 2.2, $\forall r \in R_C \setminus \mathcal{R}_{\mathcal{AC}}.\, A.r = C.r$. Hence $A \gg C$.

□

The transitivity of $\gg$ has practical implications for attack surface reduction; while reducing A's attack surface measurement compared to C's, software developers should focus on the set $\mathcal{R}_{\mathcal{AC}}$ of resources instead of either the set $\mathcal{R}_{\mathcal{AB}}$ or the set $\mathcal{R}_{\mathcal{BC}}$.

3.3.3 Relation Between Attack Surface Measurement and Potential Attacks

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 3. *Given an environment, $E = \langle U, D, T \rangle$, if the attack surface of a system A is the triple $\langle M_A^E, C_A^E, I_A^E \rangle$, the attack surface of of a system, B, is the triple $\langle M_B^E, C_B^E, I_B^E \rangle$, 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: $M_A^E = M_B^E \wedge C_A^E = C_B^E \wedge I_A^E = I_B^E$
  Without loss of generality, we assume that $R = \{r\}$ and $A.r \succ B.r$.

  Case i: $(B.r).ef \succ (A.r).ef \wedge (A.r).dp \succ (B.r).dp$
  From definitions 12 and 13, there is an action, $m_A \in M_A^E$, that has a weaker pre-condition and a stronger post condition than the same-named action, $m_B \in M_B^E$, i.e.,

  $$(m_B.pre \Rightarrow m_A.pre) \wedge (m_A.post \Rightarrow m_B.post). \tag{3.1}$$

  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,

  $$(m_1.post \Rightarrow m_B.pre) \wedge (m_B.post \Rightarrow m_2.pre). \tag{3.2}$$

  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 3 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 all else being the same between the two versions, the attack surface measurement of the newer version becomes larger and the number of potential attacks on the software increases.

3.4 A Quantitative Metric

In the previous section, we introduced a qualitative measure of a system's attack surface (Definition 14). The qualitative measure is an ordinal scale [30]; given two systems, we can only determine if one system has a relatively larger attack surface measurement than another. We, however, can not quantify the difference in the measurements.

We need a quantitative measure of the attack surface to quantify the difference in the attack surface measurements. We can also measure the absolute attack surface using the quantitative measure. 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*.

3.4.1 Damage Potential-Effort Ratio

In the previous section, in estimating a resource's contribution to the attack surface, we consider the resource's damage potential and effort in isolation. 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 might be willing to spend more effort to gain a higher privilege level that then 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 4.2.2.

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.

3.4.2 Quantitative Attack Surface Measurement Method

We quantify a system's attack surface measurement along three dimensions: methods, channels, and data. We estimate the total contribution of the methods, the total contribution of the channels, and the total contribution of the data items to the attack surface.

Definition 15. *Given the attack surface, $\langle M^E, C^E, I^E \rangle$, of a system, A, the attack surface measurement of A is the triple $\langle \sum_{m \in M^E} der_m(m), \sum_{c \in C^E} der_c(c), \sum_{d \in I^E} der_d(d) \rangle$.*

We quantitatively measure a system's attack surface in the following three steps.

1. Given a system, s, and its environment, E_s, we identify a set, M^{E_s}, of entry points and exit points, a set, C^{E_s}, of channels, and a set, I^{E_s}, of untrusted data items of s.
2. We estimate the damage potential-effort ratio, $der_m(m)$, of each method $m \in M^{E_s}$, the damage potential-effort ratio, $der_c(c)$, of each channel $c \in C^{E_s}$, and the damage potential-effort ratio, $der_d(d)$, of each data item $d \in I^{E_s}$.
3. The measure of s's attack surface is the triple $\langle \sum_{m \in M^{E_s}} der_m(m), \sum_{c \in C^{E_s}} der_c(c), \sum_{d \in I^{E_s}} der_d(d) \rangle$.

Our attack surface measurement method is analogous to the risk estimation method used in risk modeling [41]. 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, $\langle \sum_{m \in M^{E_s}} p(m) der_m(m), \sum_{c \in C^{E_s}} p(c) der_c(c), \sum_{d \in I^{E_s}} p(d)\ der_d(d) \rangle$, which is also the measure of A's attack surface.

In practice, however, it is difficult to predict defects in software [29] and to estimate the likelihood of vulnerabilities in software [38]. 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 for all channels and $p(d)$ = 1 for all data items. With our conservative approach, the measure of a system's attack surface is the triple $\langle \sum_{m \in M^{E_s}} der_m(m), \sum_{c \in C^{E_s}} der_c(c), \sum_{d \in I^{E_s}} der_d(d) \rangle$.

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 in our measurement method with our intuitive notion of security. For example, consider a system, A, that has 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 [41]. 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 parti-

tioned multiobjective risk method is recommended [9]. 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 in practice.

Chapter 4

Empirical Attack Surface Measurement Results

4.1 Introduction

In the previous chapter, we introduced an abstract method to measure a system's attack surface. In this chapter, we instantiate the abstract method and obtain a new method to measure the attack surfaces of systems implemented in C. We demonstrate the use of our method by measuring the attack surfaces of two popular open source Internet Message Access Protocol (IMAP) servers (4.3) and two File Transfer Protocol (FTP) daemons (4.4). We also perform a *parameter sensitivity analysis* of our method to provide guidelines to users of our method (4.5).

4.2 Measurement Method for Systems Implemented in C

In this section, we describe our attack surface measurement method for systems implemented in C. Figure 4.1 shows the steps followed in our attack surface measurement method. The dotted boxes show the steps done manually and the solid boxes show the

steps done programmatically. The dotted lines represent manual inputs required for measuring the attack surface. We automated as many steps as possible in our measurement method and minimized the number of manual inputs required by the method.

Two keys steps in our attack surface measurement method are the identification of relevant resources that are part of the attack surface and the estimation of the damage potential-effort ratio of each such resource. We describe the steps in Section 4.2.1 and Section 4.2.2, respectively.

4.2.1 Identification of Relevant Resources

In Step 1 of the attack surface measurement method, we identify a system's entry points and exit points, channels, and untrusted data items. We also determine the privilege levels of the entry points and the exit points, the protocols of the channels, the types of the untrusted data items, and the access rights levels of all the resources.

Entry Points and Exit Points

A direct entry point of a system is a method that receives data items directly from the system's environment. As proposed by DaCosta et al. [25], we assume that a method of a system can receive data items from the system's environment by invoking specific `C` library methods. Hence a method is a direct entry point if the method contains a call to one of the specific `C` library methods. For example, a method is a direct entry point if it contains a call to the `read` method defined in `unistd.h`. We identify a set, $Input$, of `C` library methods that a method must invoke to receive data items from the environment. We determine a system's methods that contain a call to a method in $Input$ as the system's direct entry points.

A direct exit point of a system is a method that sends data items directly to the system's environment. We assume that a method can send data items to the system's environment by invoking specific `C` library methods. We identify a set, $Output$, of `C` library methods that a method must invoke to send data items to the environment. We determine a system's

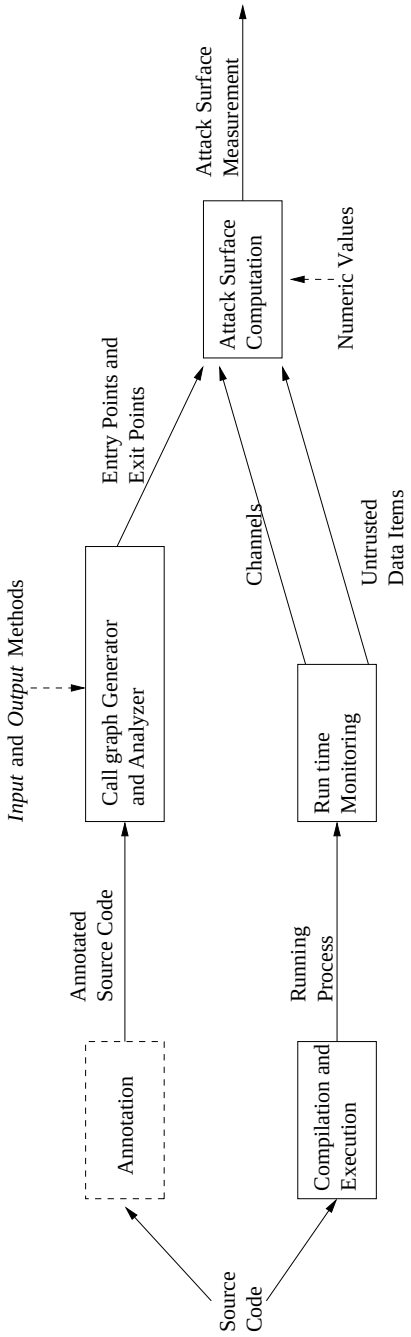

Figure 4.1: Steps of our attack surface measurement method for C.

methods that contained a call to a method in $Output$ as the system's direct exit points. Please see Appendix A for the $Input$ and $Output$ sets of methods.

An indirect entry point of a system is a method that receives data items from a direct entry point. An indirect exit point of a system is a method that sends data items to a direct exit point. We can not determine the indirect entry points (exit points) in an automated manner as there is no source code analysis tool that enables us to determine whether a direct entry point m_1 receives a data item d from the environment and a method m receives the data item d from m_1, or whether a method m passes a data item d to a direct exit point m_2 and m_2 sends the data item d to the environment. We manually determined the indirect entry points (exit points) in case of the IMAP servers; we did not determine any indirect entry points (exit points) in case of the FTP daemons.

We could not find a source code analysis tool that enables us to determine whether a direct entry point m_1 receives a data item d from the environment and a method m receives the data item d from m_1, or whether a method m passes a data item d to a direct exit point m_2 and m_2 sends the data item d to the environment; hence we do not identify the indirect entry points or the indirect exit points in an automated manner. We leave the identification of indirect entry points (indirect exit points) as future work.

We also determine the privilege and the access rights level for each entry point and exit point. On a UNIX system, a process changes its privilege through a set of *uid-setting* system calls such as `setuid` [17]. If a process changes its privilege level from p_1 to p_2 by invoking a uid-setting system call, su, then we assume that all methods invoked before su run with privilege p_1 and all methods invoked after su run with privilege p_2. For example, if a process starts with `root` privilege and then drops privilege by calling `setuid`, then all methods that are invoked before `setuid` have `root` privilege and all methods that are invoked after `setuid` have `non-root` privilege.

In order to determine the access rights levels, we identify the code locations where authentication is performed in a system's codebase. We assume that any method that is invoked before user authentication takes place has unauthenticated access rights and any method that is invoked after successful authentication has authenticated access rights.

We annotate a system's codebase to indicate the code location where privilege levels and access rights levels change. We generate the call graph from the annotated code using `cflow` [81]. From the call graph, we determine the methods that contain a call to a method in $Input$ or a method in $Output$ and the privilege and access rights of each such method. These methods are the direct entry points and direct exit points, respectively. Notice that a method may run with different privilege levels during different executions of the method. Similarly, a method may be accessible with multiple access rights levels. We identify such a method as a direct entry point (direct exit point) multiple times, once per each pair of privilege level and access rights level.

Channels

We monitor the run time behavior of a system to identify the channels opened by the systems and to determine the protocol and access rights level of each such channel.

Untrusted Data Items

We also monitor the run time behavior of a system to identify the untrusted data items accessed by the system and to determine the type and access rights level of each untrusted data item.

4.2.2 Estimation of a Resource's Damage Potential-Effort Ratio

In Step 2 of the attack surface measurement method, we quantify the damage potential-effort ratios of the resources that are part of a system's attack surface. We assign numeric values to the six attributes introduced in Section 3.3 to estimate numeric damage potential-effort ratios.

We impose a total ordering among the privilege levels such that a method running with a higher privilege level in the total ordering has a higher damage potential. We assign numeric values to the privilege levels in accordance to the total ordering, i.e., if a privilege

level, p_1, is greater than a privilege level, p_2, in the total ordering, then we assign a higher number to p_1 compared to p_2. For example, we assume a method running as `root` has a higher damage potential than a method running as `authenticated` user; hence `root` $>$ `authenticated user` in the total ordering and we assigned a higher number to `root` than `authenticated user`. The exact choice of the numeric values is subjective and depends on a system and its environment. In practice, the users of our attack surface measurement method will assign the numeric values based on their knowledge and experience of a system and its environment.

Similarly, we assign numeric values to channel protocols, data item types, and access rights levels. We estimate a resource's damage potential-effort ratio from the numeric values assigned to the resource's damage potential and effort. For example, we estimate a method's damage potential-effort ratio from the numeric values assigned to the method's privilege and access rights level.

4.3 IMAP Measurement Results

In this section, we demonstrate the use of our attack surface metric by measuring and comparing the attack surfaces of two popular open source IMAP servers. IMAP provides a mechanism for email stored on a remote server to be accessed over a network [22]. Users can connect to an IMAP server, authenticate themselves, and perform actions such as reading and deleting email. IMAP's ability to let the users access email from more than one computer has increased IMAP's popularity. Today open source IMAP servers are widely used; hence we chose to measure the attack surfaces of two open source IMAP servers: Courier-IMAP 4.0.1 and Cyrus 2.2.10.

Courier-IMAP server is the IMAP server included in the Courier mail server [47]. Courier-IMAP server is also configurable as a standalone IMAP server. The server uses the custom `Maildir` mailbox format for scalability and fast access to mailboxes. The Courier code base contains 140K lines of code that implements the IMAP daemon, an authentication library called `courier-authlib`, and a number of helper libraries and

tools.

The Cyrus IMAP server was implemented and is maintained by Project Cyrus [23]. It is designed to be a fast and scalable IMAP server to be used by small to large enterprise environments. The server uses custom mailbox formats for efficiency, scalability, and ease of administration. The Cyrus codebase contains 160K lines of code that implements the IMAP daemon, a set of authentication libraries called the `SASL library`, and a number of libraries and auxiliary tools.

We considered only the code specific to the IMAP daemon in our measurements. The Courier code base contains nearly 33K lines of code specific to the IMAP daemon and the Cyrus code base contains nearly 34K lines of code specific to the IMAP daemon.

4.3.1 Entry Points and Exit Points

We identified the entry points and exit points of the IMAP servers from their code bases. We only identified the indirect entry points reachable from the `main` method in both IMAP codebases. Hence our measurements are under-approximations of the measure of the attack surfaces.

All the methods in the Cyrus codebase run with a special UNIX user, `cyrus`, privilege. The methods are accessible with `admin`, `authenticated` user, `unauthenticated` user, and `anonymous` user access rights. The methods in the Courier codebase run with `root` and `authenticated` user privileges. The methods are accessible with `authenticated` user and `unauthenticated` user access rights. The Courier codebase does not support `admin` user and `anonymous` user access rights. We show the number of direct entry points (DEP), direct exit points (DExP), and indirect entry points (IEP) for each privilege level and access rights level pair in Table 4.1.

4.3.2 Channels

We observed the run time behavior of the default installations of both IMAP servers to identify the channels. Both IMAP daemons open a `TCP` channel on port 143 and an

Courier				
Privilege	Access Rights	DEP	DExP	IEP
root	unauthenticated	28	17	11
root	authenticated	21	10	0
authenticated	authenticated	113	28	1
Cyrus				
Privilege	Access Rights	DEP	DExP	IEP
cyrus	unauthenticated	16	17	7
cyrus	authenticated	12	21	2
cyrus	admin	13	22	2
cyrus	anonymous	12	21	2

Table 4.1: The number of entry points and exit points of the IMAP servers (DEP = Direct Entry Point, DExP = Direct Exit Point, IEP = Indirect Entry Point).

SSL channel on port 993 to listen to user requests. In addition, the Cyrus daemon opens a TCP channel on port 2000 for users to edit their sieve filters. These channels are accessible with remote unauthenticated user access rights. The Courier IMAP daemon opens a local UNIX socket channel to communicate with the authentication daemon. The Cyrus IMAP daemon opens a local UNIX socket channel to communicate with the Local Mail Transfer Protocol (LMTP) daemon. These channels are accessible with local authenticated user access rights. We show the number of channels for each channel type and access rights pair in Table 4.2.

4.3.3 Untrusted Data Items

Similar to channels, we observed the run time behavior of the default installations of both IMAP servers to identify the untrusted data items. Both IMAP daemons read or write persistent data items of file type; both daemons used configuration files, authentication files, executable files, libraries, lock files, user mail files, and mail metadata files. The files

Courier		
Type	Access Rights	Count
TCP	remote unauthenticated	1
SSL	remote unauthenticated	1
UNIX socket	local authenticated	1
Cyrus		
Type	Access Rights	Count
TCP	remote unauthenticated	2
SSL	remote unauthenticated	1
UNIX socket	local authenticated	1

Table 4.2: The number of channels opened by the IMAP servers.

Courier			Cyrus		
Type	Access Rights	Count	Type	Access Rights	Count
file	root	74	file	root	50
file	authenticated	13	file	cyrus	26
file	world	53	file	world	50

Table 4.3: The number of untrusted data items accessed by the IMAP servers.

of the Courier IMAP daemon can be accessed with `root`, `authenticated` user, and `world` access rights. The files of the Cyrus IMAP daemon can be accessed with `root`, `cyrus`, and `world` access rights. Recall that an attacker can use an untrusted data item in an attack by reading or writing the data item. Hence we identified the read and the write access rights levels of a file separately; we counted each file twice, once for the read access rights level and once for the write access rights level. We show the number of untrusted data items for each data item type and access rights pair in Table 4.3.

4.3.4 Estimation of the Damage Potential-Effort Ratio

We assigned the following total ordering among the set of privilege levels: `root` > `cyrus` > `authenticated` > `unauthenticated`. A method running with `cyrus` privilege in the Cyrus IMAP daemon has access to every user's email files; hence we assumed a method running as `cyrus` has higher damage potential than a method running as `authenticated` user. We assigned the following total ordering among the set of access rights levels of the methods: `admin` > `authenticated` > `anonymous` = `unauthenticated`. `admin` users are special users in Cyrus; hence we assumed the attacker spends higher effort to acquire `admin` access rights compared to `authenticated` access rights.

We assigned the following total ordering among the channel types: `TCP` = `SSL` = `UNIX socket`, i.e., we assumed that each channel has the same damage potential. We assigned the following total ordering among the access rights levels of the channels: `local authenticated` > `remote unauthenticated`.

Both IMAP daemons have untrusted data items of `file` type only, hence assigning a total ordering was trivial. We assigned the following total ordering among the access rights levels of the data items: `root` > `cyrus` > `authenticated` > `world`. The `cyrus` user is a special user; hence we assumed the attacker spends more effort to acquire `cyrus` access rights compared to `authenticated` access rights.

We assigned the numeric values to the attributes based on our knowledge of the IMAP servers and UNIX security. We show the numeric values in Table 4.4.

4.3.5 Attack Surface Measurements

In Step 3 of the attack surface measurement method, we estimated the total contribution of the methods, the total contribution of the channels, and the total contribution of the data items to the attack surfaces of both IMAP daemons. From Table 4.1 and Table 4.4, the total contribution of the methods of Courier is ($56 \times (\frac{5}{1})$ + $30 \times (\frac{5}{3})$ + $142 \times (\frac{3}{3})$) = 522.00. From Table 4.2 and Table 4.4, the total contribution of the channels of Courier is ($1 \times (\frac{1}{1})$ +

Method Privilege	Numeric Value	Access Rights	Numeric Value
`root`	5	`admin`	4
`cyrus`	4	`authenticated`	3
`authenticated`	3	`anonymous`	1
`unauthenticated`	1	`unauthenticated`	1
Channel Type	Numeric Value	Access Rights	Numeric Value
`TCP`	1	`local authenticated`	4
`SSL`	1	`remote unauthenticated`	1
`UNIX socket`	1		
Data Item Type	Numeric Value	Access Rights	Numeric Value
`file`	1	`root`	5
		`cyrus`	4
		`authenticated`	3
		`world`	1

Table 4.4: Numeric values assigned to the values of the attributes.

$1 \times (\frac{1}{1}) + 1 \times (\frac{1}{4})) = 2.25$. From Table 4.3 and Table 4.4, the total contribution of the data items of Courier is $(74 \times (\frac{1}{5}) + 13 \times (\frac{1}{3}) + 53 \times (\frac{1}{1})) = 72.13$. Hence the measure of the Courier IMAP daemon's attack surface is the triple $\langle 522.00, 2.25, 72.13 \rangle$. Similarly, the measure of the Cyrus IMAP daemon's attack surface is the triple $\langle 383.60, 3.25, 66.50 \rangle$. We show the measurement results in Figure 4.2.

The attack surface metric tells us that the Cyrus IMAP daemon is more secure along the method dimension and data dimension whereas the Courier IMAP daemon is more secure along the channel dimension. The measurement of the attack surface along three dimensions offers a design choice to the users of our metric. For example, keeping the attack surface measurement separated along three different dimensions, rather than coalescing the numbers into one, lets system administrators choose a dimension appropriate for their need.

In order to choose one IMAP daemon over another, we first determine the dimension of the attack surface that presents more risk using our knowledge of the IMAP daemons

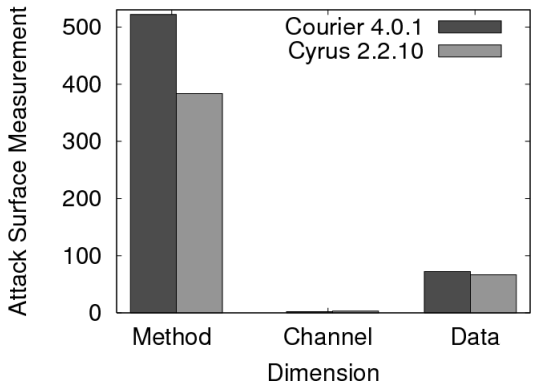

Figure 4.2: Attack surface measurements of the IMAP servers.

and the operating environment; we then choose the IMAP daemon that is more secure along that dimension. For example, if we were concerned about privilege elevation on the host running the IMAP daemon, then the method dimension presents more risk and the attack surface metric suggests that we would choose the Cyrus daemon over the Courier daemon. Similarly, if we were concerned about the number of open channels on the host running the IMAP daemon, then the channel dimension presents more risk and we would choose the Courier daemon. If we were concerned about the safety of email files, then the data dimension presents more risk and we would choose the Cyrus daemon.

4.4 FTP Measurement Results

We also measured and compared the attack surfaces of two open source FTP daemons: ProFTPD 1.2.10 and Wu-FTPD 2.6.2. Our choice of the FTP daemons was guided by two factors: popularity and availability of source code. ProFTPD was implemented and is maintained by the ProFTPD project group [84]. Wu-FTPD was implemented and is maintained at the University of Washington [40]. The ProFTP codebase contains 28K lines of C code and the Wu-FTP codebase contains 26K lines of C code; we only considered code specific to the FTP daemon.

4.4.1 Entry Points and Exit Points

The methods in the ProFTP codebase run with `root` and a special UNIX user, `proftpd`, privilege. The methods are accessible with `root`, `authenticated` user, `unauthenticated` user, and `anonymous` user access rights. The methods in the Wu-FTP codebase run with `root` and `authenticated` user privilege. The methods are accessible with `authenticated` user, `anonymous` user, and `guest` user access rights. We show the number of direct entry points (DEP) and direct exit points (DExP) for each privilege level and access rights level pair in Table 4.5. We did not identify any indirect entry points (exit points) due to the lack of an adequate source code analysis tool. Hence our measurements are under-approximations of the measure of the attack surfaces.

ProFTPD			
Privilege	Access Rights	DEP	DExP
`root`	`root`	8	8
`root`	`authenticated`	12	13
`root`	`unauthenticated`	13	14
`proftpd`	`authenticated`	6	4
`proftpd`	`unauthenticated`	13	6
`proftpd`	`anonymous`	6	4
Wu-FTPD			
Privilege	Access Rights	DEP	DExP
`root`	`authenticated`	9	2
`root`	`unauthenticated`	30	9
`authenticated`	`authenticated`	11	3
`authenticated`	`anonymous`	11	3
`authenticated`	`guest`	27	14

Table 4.5: The number of entry points and exit points of the FTP daemons (DEP = Direct Entry Point, DExP = Direct Exit Point).

4.4.2 Channels

We monitored the run time behavior of the default installations of both FTP daemons to identify the channels. Both FTP daemons open a `TCP` channel so that FTP clients can communicate with the daemons. These channels are accessible with `remote unauthenticated` user access rights. We show the number of channels for each protocol and access rights pair in Table 4.6.

ProFTPD		
Protocol	Access Rights	Count
`TCP`	`remote unauthenticated`	1
Wu-FTPD		
Protocol	Access Rights	Count
`TCP`	`remote unauthenticated`	1

Table 4.6: The number of channels opened by the FTP daemons.

4.4.3 Untrusted Data Items

We also monitored the run time behavior of the default installations of both FTP daemons to identify the untrusted data items. Both FTP daemons read or write persistent data items of `file` type; both daemons used configuration files, authentication files, executable files, libraries, and log files. The files of ProFTPD can be accessed with `root`, `proftpd` user, and `world` access rights. The files of Wu-FTPD can be accessed with `root`, `authenticated` user, and `world` access rights. We show the number of untrusted data items for each data item type and access rights pair in Table 4.7.

4.4.4 Estimation of the Damage Potential-Effort Ratio

We assigned the following total ordering among the set of privilege levels: `root` > `proftpd` > `authenticated`. A method running with `proftpd` privilege in ProFTPD

ProFTPD			Wu-FTPD		
Type	Access Rights	Count	Type	Access Rights	Count
file	root	12	file	root	23
file	proftpd	18	file	auth	12
file	world	12	file	world	9

Table 4.7: The number of untrusted data items accessed by the FTP daemons.

has access to all the files on the FTP server; hence we assumed a method running as `proftpd` user has higher damage potential than a method running as `authenticated` user. We assigned the following total ordering among the set of access rights levels of the methods: `root` > `authenticated` > `anonymous` = `unauthenticated` = `guest`.

Both FTP daemons have channels with only `TCP` as the protocol and `remote unauthenticated` access rights; hence assigning a total ordering was trivial. Also, both FTP daemons have untrusted data items of `file` type only and hence assigning a total ordering was trivial.

We assigned the following total ordering among the access rights levels of the data items: `root` > `proftpd` > `authenticated` > `world`. The `proftpd` user is a special user, hence we assumed the attacker spends more effort to acquire `proftpd` access rights compared to `authenticated` access rights.

We assigned the numeric values to the attributes based on our knowledge of the FTP daemons and UNIX security. We show the numeric values in Table 4.8.

4.4.5 Attack Surface Measurements

In Step 3 of the attack surface measurement method, we estimated the total contribution of the methods, the total contribution of the channels, and the total contribution of the data items to the attack surfaces of both FTP daemons. From Table 4.5 and Table 4.8, the total contribution of the methods of ProFTPD is $(16 \times (\frac{5}{5}) + 25 \times (\frac{5}{3}) + 10 \times (\frac{4}{3}) + 19 \times (\frac{4}{1})$

Method Privilege	Value	Access Rights	Value
`root`	5	`root`	5
`proftpd`	4	`authenticated`	3
`authenticated`	3	`anonymous`	1
		`unauthenticated`	1
		`guest`	1
Channel Protocol	Value	Access Rights	Value
`TCP`	1	`remote unauth`	1
Data Item Type	Value	Access Rights	Value
`file`	1	`root`	5
		`proftpd`	4
		`authenticated`	3
		`world`	1

Table 4.8: Numeric values assigned to the values of the attributes.

$+ 10 \times (\frac{4}{1})) = 321.9$. From Table 4.6 and Table 4.8, the total contribution of the channels of ProFTPD is $1\times (\frac{1}{1}) = 1$. From Table 4.7 and Table 4.8, the total contribution of the data items of ProFTPD is $(12 \times (\frac{1}{5}) + 18 \times (\frac{1}{4}) + 12 \times (\frac{1}{1})) = 18.9$. Hence the measure of ProFTPD's attack surface is the triple $\langle 312.99,\ 1.00,\ 18.90 \rangle$. Similarly, the measure of Wu-FTPD's attack surface is the triple $\langle 392.33,\ 1.00,\ 17.60 \rangle$. We show the attack surface measurements in Figure 4.3.

The attack surface metric tells 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. Similar to the IMAP servers, in order to choose one FTP daemon over another, we first determine the dimension of the attack surface that presents more risk using our knowledge of the FTP daemons and the operating environment and then choose the FTP daemon that is more secure along that dimension.

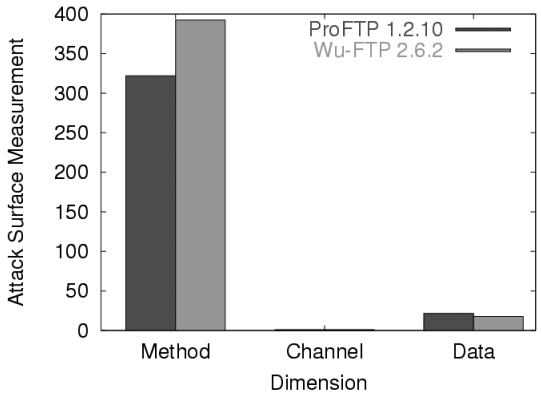

Figure 4.3: Attack surface measurements of the FTP daemons.

4.5 Parameter Sensitivity Analysis

A key step in our attack surface measurement method is the estimation of the damage potential-effort ratio. In our current method, we rely on the the users of our metric to estimate a resource's damage potential-effort ratio; we assume that the users use their knowledge of a system and its environment to impose a total ordering among the values of an attribute and then to assign numeric values according to the total ordering. Since numeric value assignment is dependent on domain knowledge, we used parameter sensitivity analysis to provide guidelines to the users for choosing appropriate numeric values [86].

In our analysis, we assumed that the users already have imposed total orderings among the values of the attributes. We leave the provision of guidelines for imposing a total ordering as future work; we discuss this in details in Chapter 8.

4.5.1 Method

The attack surface measurement along the method dimension depends on the following three parameters: the number of entry points and exit points, the numeric values assigned to the privilege levels, and the numeric values assigned to the access rights levels. Given two systems, either both systems have comparable numbers of entry points and exit points (e.g., ProFTPD = 107 and Wu-FTPD = 109) or the number of entry points and exit points

of one system differs significantly from the other system (e.g., Cyrus = 147 and Courier = 239). For both these cases, we analyzed the effects of the privilege numeric values and the access rights numeric values on the attack surface measurement.

We studied the effects on the measurement as we increase the difference in the numeric values assigned to the attributes. To keep our analysis simple, we assumed that the difference, `diff`, in the numeric values assigned to successive privilege and access rights levels is uniform. For example, the difference in the numeric values assigned to `authenticated` and `proftpd` is the same as the difference in the numeric values assigned to `proftpd` and `root`. We assigned a fixed numeric value, 3, to the lowest privilege level `authenticated`. We then assigned the numeric value (3 + `diff`) to `proftpd` and the numeric value (3 + 2 * `diff`) to `root`. We similarly assigned a fixed numeric value, 1, to the lowest access rights level `unauthenticated` and then assign numeric values to the rest of the access rights levels. We observed the effects of changing the value of `diff` from a low value of 1 to a high value of 20.

FTP Measurements Analysis

We show the effects of changing the value of `diff` on the attack surface measurements of the FTP daemons in Figure 4.4[1]. For example, when the privilege difference is 2 and the access rights difference is 17, ProFTPD's attack surface measurement is 349.7 and Wu-FTPD's attack surface measurement is 444.6. Hence Wu-FTPD has a larger attack surface measurement than ProFTPD. Similarly, when the privilege difference is 17 and the access rights difference is 6, ProFTPD's attack surface measurement is 1785.3 and Wu-FTPD's attack surface measurement is 1672.1. Hence Wu-FTPD has a smaller attack surface measurement than ProFTPD.

We show the attack surface measurements as a projection onto a two dimensional plane in Figure 4.5. In the projection, we only see the FTP daemon that has a larger attack surface measurement. For example, when the privilege difference is 2 and the access

[1]We expect the measurements to form a three dimensional plane; Figure 4.4, however, shows the measurements to form a curved surface. This is an artifact of Gnuplot's rendering.

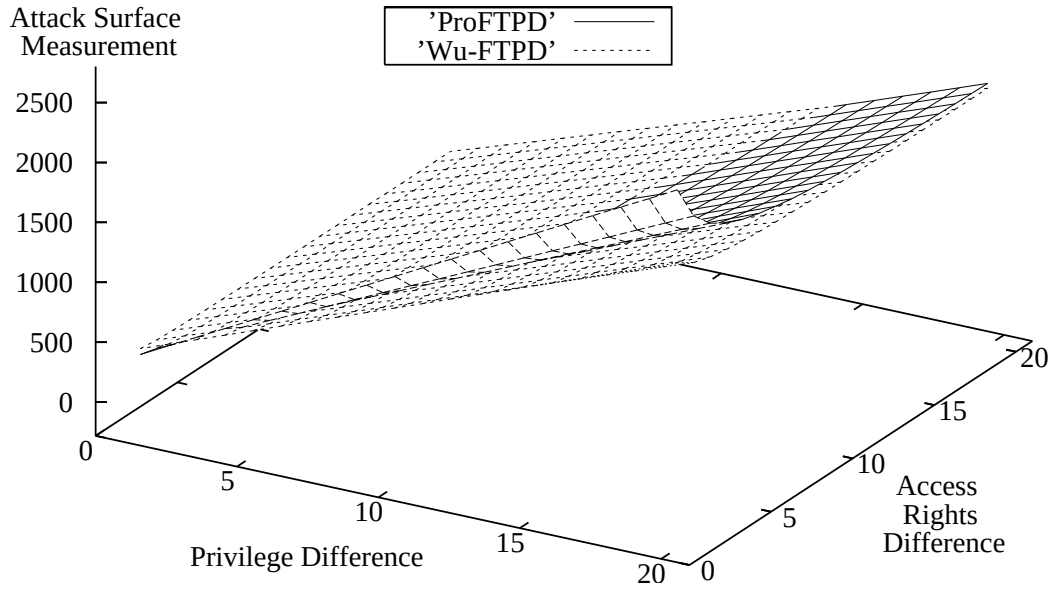

Figure 4.4: Attack surface measurements of the FTP daemons along the method dimension.

rights difference is 17, Wu-FTPD has a larger attack surface than ProFTPD and hence we see Wu-FTPD in the projection. Similarly, when the privilege difference is 17 and the access rights difference is 6, we see ProFTPD in the projection.

From Figure 4.5, when the privilege difference is low (1-2), Wu-FTPD has a larger attack surface measurement for all possible values of the access rights difference. When the privilege difference is low, the access rights values do not matter; the number of entry points and exit points is the dominating parameter. Since Wu-FTPD has a larger number of entry points and exit points, it has a larger attack surface measurement irrespective of the access rights difference.

When the privilege difference is high (15-20), ProFTPD has a larger attack surface measurement for all possible values of the access rights difference. The `proftpd` privilege level contributes more in case of ProFTPD compared to the `authenticated` privilege level of Wu-FTPD. The access rights values do not matter; the privilege values are the dominating parameter.

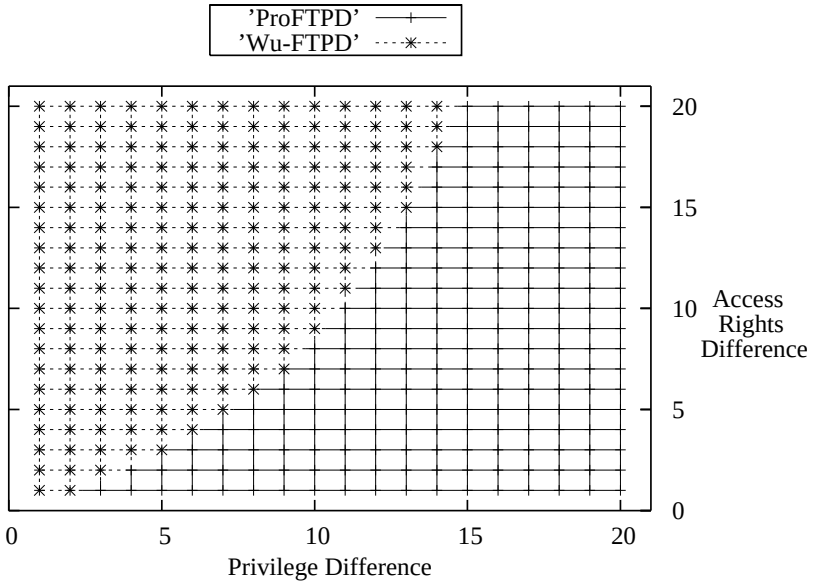

Figure 4.5: Projection of the measurements of the FTP daemons.

When the privilege difference is medium (3-14), the access rights values do matter. ProFTPD has a larger number of methods accessible with `authenticated` access rights than the number of methods accessible with `unauthenticated` access rights. Wu-FTPD has a smaller number of methods accessible with `authenticated` access rights than the number of methods accessible with `unauthenticated` access rights. Hence with increasing access rights difference, the methods of ProFTPD make smaller contribution to the attack surface compared to the methods of Wu-FTPD. Hence ProFTPD has a larger measurement for lower access rights difference and Wu-FTPD has a larger measurement for higher access rights difference.

IMAP Measurements Analysis

We show the attack surface measurements of the IMAP servers as a projection onto a two dimensional plane in Figure 4.6. In the projection, we only see the IMAP server that has a larger attack surface measurement.

From Figure 4.6, Courier has a larger attack surface measurement for almost all pos-

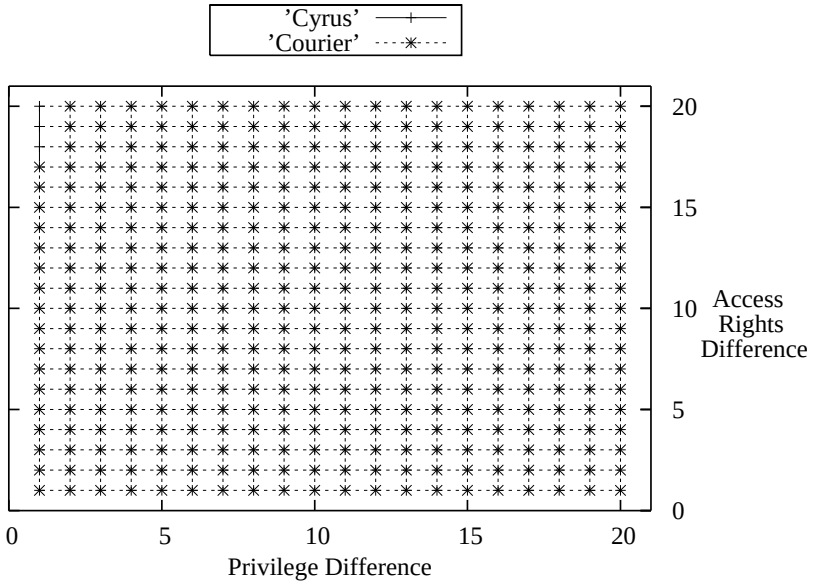

Figure 4.6: Projection of the measurements of the IMAP servers.

sible privilege difference and access rights difference. The privilege values and access rights values do not matter; the number of entry points and exit points is the dominating parameter. Courier has a significantly larger number of entry points and exit points than Cyrus; hence Courier has a larger attack surface measurement.

Observations

Our choice of the numeric values should be such that both the privilege values and the access rights values affect the outcome of the attack surface measurements comparison. The FTP measurements analysis shows that if both systems have comparable numbers of entry points and exit points, then the access rights values do not affect the measurements if the privilege difference is low or high. Hence we should choose a medium difference for the privilege values. The IMAP measurement analysis shows that if one system has a significantly larger number of entry points and exit points than the other, then no choice of the privilege difference or the access rights difference affects the measurement. Also, if we choose a medium or a high difference for the privilege values, then we should not choose

a low difference for the access rights values; otherwise the privilege values will dominate the access rights values in the damage potential-effort ratio. Hence these two observations combined suggest that the users of our metric should choose a medium difference for the privilege values and either a medium or a high difference for access rights values.

4.5.2 Channel

We performed a similar analysis for the measurements along the channel dimension by changing the difference in the numeric values assigned to the protocols and the access rights levels. ProFTPD and Wu-FTPD open the same set of channels; hence ProFTPD and Wu-FTPD have the same measurements for all possible differences in the protocol values and the access rights values. The set of channels opened by Courier is a subset of the channels opened by Cyrus; hence Cyrus has a larger attack measurement than Courier for all possible differences in the protocol values and the access rights values.

Both the FTP measurements and the IMAP measurements show that similar systems open comparable sets of channels, i.e., either they open the same set of channels or the set of channels opened by one system is a subset of the other. Thus we do not need to impose a total ordering and assign numeric values to the attributes; we can determine whether one system has a larger attack surface along the method dimension from the number of channels opened by the systems. If the channels opened by the system, however, are not comparable, then we should follow the suggestions for the method dimension (discussed in Section 4.5.1) to assign numeric values to the protocols and the access rights levels.

4.5.3 Data

We performed a similar analysis for the measurements along the data dimension by changing the difference in the numeric values assigned to the data types and the access rights levels. ProFTPD has a larger attack surface measurement than Wu-FTPD for all possible differences in the numeric values assigned to the data types and the access rights levels. The number of files accessed by ProFTPD (42) is comparable to the number of files ac-

cessed by Wu-FTPD (44). Wu-FTPD, however, has a smaller attack surface measurement because it has a large number of files accessible with the `root` access rights. An attacker spends the maximum effort to obtain the `root` access rights level; hence we assign the highest numeric value to the `root` access rights level. Thus the files accessible with the `root` access rights level have the lowest damage potential-effort ratio and make the least contribution to the attack surface measurement.

Courier has a larger attack surface measurement than Cyrus for all possible differences in the numeric values assigned to the data types and the access rights levels. Cyrus and Courier access comparable numbers of files. The larger attack surface measurement of Courier, however, is due to a larger number of files accessible with the `unauthenticated` access rights. An attacker spends the least effort to obtain the `unauthenticated` access rights level; hence we assign the lowest numeric value to the `unauthenticated` access rights level. Thus the files accessible with the `unauthenticated` access rights level have the highest damage potential-effort ratio and make the highest contribution to the attack surface measurement.

Both the FTP and the IMAP measurements show that the systems access data items of only `file` type. Hence assigning numeric values to the data types is trivial. If a system, however, accesses data items of other type, then we should follow the suggestions for the method dimension (discussed in Section 4.5.1) to assign numeric values to the data types. We should also follow the same suggestions for assigning numeric values to the access rights levels.

Chapter 5

Empirical Studies for Validation

5.1 Introduction

A key challenge in security metrics research is the validation of a metric. Validating a software attribute's measure is hard in general [51]; also, there is no consensus in the community on a validation framework [89, 66, 12, 104, 107, 51, 74, 52]. Security is a software attribute that is hard to measure; hence validating a security measure is even harder. There is not much prior work on validating security measurements and metrics. In this chapter, we conduct three empirical studies to validate our attack surface metric. Our studies are exploratory in nature and open up avenues for future work, which we discuss in Chapter 8.

Our empirical studies are inspired by the software engineering research community's approaches of validating software metrics [30, 12]. In practice, validation approaches are based on distinguishing *measures* from *prediction systems*; measures are used to numerically characterize software attributes whereas prediction systems are used to predict the values of software attributes. For example, lines of code (LOC) is a measure of software "length"; the measure becomes a prediction system if we use LOC to predict software "complexity." A software measure is validated by establishing that the measure is a proper numerical characterization of an attribute; Kitchenham, Pfleeger, and Fenton introduce

concepts needed for validating a software measure and recommend the use of both theoretical and empirical approaches for validation [51]. Similarly, prediction systems are validated by establishing their accuracy via empirical means.

Our attack surface metric plays a dual role: the metric is a measure of a software attribute, i.e., the attack surface and also a prediction system to indicate the security risk of software. Hence we took a two-step approach for validation. First, we validated the measure by validating our attack surface measurement method. Second, we validated the prediction system by validating attack surface measurements. In general, any software security metric is likely to have two components: a software measure and a prediction system. Hence we recommend the two-step validation approach for software security metrics.

5.1.1 Validating a Software Measure

We conducted two empirical studies to validate our measurement method: a statistical analysis of data collected from Microsoft Security Bulletins (MSB) (Section 5.2) and an expert user survey (Section 5.3). Our approach is motivated by the notion of *convergent evidence* in Psychology [1] [14, 42]; since each study has its own strengths and weaknesses, the convergence in the studies' findings enhances our belief that the findings are valid and not methodological artifacts [95]. Also, the statistical analysis is with respect to Microsoft Windows whereas the expert survey is with respect to Linux. Hence our validation approach is agnostic to operating system and system software.

Our validation approach conforms to Kitchenham et al.'s validation concepts. Kitchenham et al. suggest that we confirm the following to establish the validity of a measure: (1) *attribute validity*, i.e., the attributes we are interested in are exhibited by software, (2) *unit validity*, i.e., the measurement unit is appropriate for an attribute, (3) *instrument validity*, i.e., the model underlying a measurement instrument is valid, and (4) *protocol validity*,

[1] Convergent evidence is used when there is no single line of strong evidence supporting a claim; if there are several lines of weak or indirect evidence that support the claim and trend in the same direction, then all combined they support the claim more than any single line of evidence.

i.e., an appropriate measurement protocol is used.

In case of the attack surface metric, establishing attribute validity is akin to showing that the three dimensions of the attack surface and the six attributes used for estimating damage potential and effort are appropriate. We established theoretical attribute validity in our I/O automata model and empirical validity using both our empirical studies. We do not use explicit units in attack surface measurements; the attack surface measurement's unit, however, is the same as the damage potential-effort ratio's unit. Hence establishing unit validity is akin to showing that a resource's damage potential-effort ratio is the resource's contribution to the attack surface measurement. We used the expert survey to establish unit validity. Our attack surface measurement method is based on our formal I/O automata model; the formal model establishes instrument validity. Hence instrument validity is inherent in our measurement method. Kitchenham et al. suggest that a valid measurement protocol must be unambiguous and self-consistent and prevent problems such as double counting; any protocol that satisfies the criteria is validated through peer acceptance rather than any theoretical or empirical approach. Our measurement method satisfies Kitchenham et al.'s criteria of a valid measurement protocol; hence protocol validity is inherent in our measurement method.

Table 5.1.1 summarizes our validation approaches and shows the mapping of our approaches to Kitchenham et al.'s validation framework.

Step	Validity	Theoretical	Empirical
Software Measure	Attribute Validity	IO Automata Model	Expert Survey, MSB Analysis
	Unit Validity		Expert Survey
	Instrument Validity	IO Automata Model	
	Protocol Validity		
Prediction System	Prediction Validity	IO Automata Model	Patch Analysis, Anecdotal Evidence

Table 5.1: Summary of our validation approaches.

5.1.2 Validating a Prediction System

We took two different approaches to validate the prediction system of the attack surface metric. First, we formally showed that a larger attack surface leads to a larger number of potential attacks on software in our I/O automata model (Chapter 3). Second, we established a correlation between attack surface measurements and patches of security vulnerabilities in software. A patch of a software system improves the system's security by removing an exploitable vulnerability from the system; hence we expect the patch to reduce the system's attack surface measurement. We demonstrated that a majority of patches in open source software reduce the attack surface measurement (Section 5.4). We also present anecdotal evidence from software industry illustrating the effectiveness of attack surface reduction in mitigating security risk of software (Section 5.5). Table 5.1.1 summarizes our approaches for validating the attack surface metric's prediction system.

5.1.3 Liu and Traore's Independent Validation Framework

Liu and Traore introduce a theoretical validation framework for security measures and metrics [56, 57]. Their validation framework is based on software security design principles accepted by the community and is independent of the two-step validation approach introduced earlier. Liu and Traore introduce a set of measurement properties of a valid security metric and demonstrate that the attack surface metric possesses the properties; hence the attack surface metric is a valid metric in their framework. Liu and Traore's theoretical validation complements our two-step validation of the attack surface metric. We discuss Liu and Traore's theoretical validation of the attack surface metric in Section 5.6.

5.2 Statistical Analysis of Microsoft Security Bulletins

Our attack surface measurement method is based on the following three key hypotheses; hence we validated the three hypotheses in order to validate the method.

1. Methods, channels, and data are the dimensions of a system's attack surface.

2. The six resource attributes (a method's privilege and access rights, a channel's protocol and access rights, and a data item's type and access rights) are indicators of damage potential and effort.

3. A resource's damage potential-effort ratio is an indicator of how likely the resource is going to be used in attacks.

In our first empirical study, we used a statistical analysis of data collected from Microsoft Security Bulletins to validate the hypotheses. An MSB describes an exploitable vulnerability present in a Microsoft software product [19]. The bulletin mentions the software's code (methods in our framework) that contains the vulnerability and the resources (communication channels, data items, or a combination of both) that an attacker has to use to exploit the vulnerability. Hence to validate hypothesis 1, we used the data collected from MSBs to show that a software system's methods, channels, and data items are used in attacks on the system.

Microsoft assigns a *severity rating* to each bulletin based on their assessment of the impact of the vulnerability's exploitation on the software's users and the difficulty of exploitation [20]. There are four levels of severity ratings: Critical, Important, Moderate, and Low. The impact of exploitation is equivalent to damage potential in our framework and the difficulty of exploitation is equivalent to attacker effort. Each bulletin also mentions the six resource attributes that we use to estimate damage potential and attacker effort in our framework. Hence to validate hypothesis 2, we used the data collected from the MSBs to establish statistically significant correlations between the severity rating and the six attributes.

The bulletins did not have any data that are relevant to a resource's likelihood of being used in attacks. Hence we could not use the MSBs to validate hypothesis 3. We used an expert survey described in Section 5.3 to validate hypothesis 3.

5.2.1 Data Collection

We collected data from 110 bulletins published over a period of two years from January 2004 to February 2006. Our data collection was a manual process based on our interpretation of a bulletin's description and hence is subject to human error. From the description contained in each bulletin, we identified the resources (methods, channels, and data items) that the attacker has to use to exploit the vulnerability described in the bulletin. For each such resource, we also identified the values of the resource's attributes that indicate the resource's damage potential and effort; we identified the methods' privilege levels and access rights levels, the channels' protocols and access rights levels, and the data items' types and access rights levels. We also identified each bulletin's severity rating. Many bulletins contained multiple vulnerabilities and many vulnerabilities were assigned different ratings for different versions of the Windows operating system. Hence the 110 bulletins resulted in 202 observations.

5.2.2 Hypothesis 1

Out of the 202 observations, 202 observations mention methods, 170 observations mention channels, and 108 observations mention data items as the resources used in exploiting the vulnerabilities reported in Microsoft security bulletins (Table 5.2). These findings suggest that methods, channels, and data items are the resources used in attacks on software systems and hence are the dimensions of a system's attack surface.

Resource	Count
Methods	202
Channels	170
Data	108

Table 5.2: Number of observations that mention methods, channels, and data.

The data does not rule out the existence of other dimensions of the attack surface. We, however, did not find any other resource types mentioned in the bulletins. We could assign one of the three types to all resources mentioned in the bulletins.

5.2.3 Hypothesis 2

A bulletin's severity rating depends on Microsoft's assessment of the impact of exploiting the vulnerability described in the bulletin and the difficulty of exploitation. The higher the impact, the higher the rating; the lower the difficulty, the higher the rating. The impact and the difficulty of exploitation are equivalent to the damage potential and the attacker effort in our measurement method, respectively. Hence we expect the severity rating to depend on the six attributes that are indicators of damage potential and attacker effort; we also expect the severity rating to increase with an increase in the value of an attribute that is an indicator of damage potential, e.g., method privilege, and to decrease with an increase in the value of an attribute that is an indicator of attacker effort, e.g., method access rights. In other words, we expect an indicator of damage potential to be a significant predictor of the severity rating and to be positively correlated with the severity rating. Similarly, we expect an indicator of attacker effort to be a significant predictor of the severity rating and to be negatively correlated with the severity rating.

We used ordered logistic regression analysis to test for the significance of the attributes in explaining the severity rating [106]. Logistic regression uses maximum likelihood estimates (MLE) to compute the regression coefficients; MLE seeks to maximize the likelihood of the observed values of the dependent variable, i.e., the severity rating, being predicted from the observed values of the independents variables, i.e., the resource attributes. A positive coefficient indicates a positive correlation between an attribute and the severity rating, and a negative coefficient indicates a negative correlation. We used ordered logistic regression because the dependent variable, i.e., the severity rating, is an *ordinal* variable, i.e., the values assigned to the severity rating can be ordered as Critical $>$ Important $>$ Medium $>$ Low.

We used a two sided z-test's p-value to determine if an attribute is a significant predictor of the severity rating. The z-test is used to test the null hypothesis that an attribute's regression coefficient is zero and hence the attribute is not a significant predictor of the severity rating. We used a two sided z-test because the regression coefficients can be either positive or negative. A p-value of less than 0.05 indicates statistical significance and

thus that an attribute is a significant predictor of the severity rating.

We below summarize our findings that suggest that the six attributes are indicators of damage potential and effort.

1. A method's privilege is a significant predictor of the severity rating and is positively correlated with the rating (Table 5.3).
2. A channel's protocol (Table 5.4) and a data item's type (Table 5.5) are significant predictors of the severity rating. We could not determine the correlation between these two attributes and the severity rating.
3. A method's access rights (Table 5.3, row 2), a channel's access rights (Table 5.4, row 4), and a data item's access rights (Table 5.5, row 6) are significant predictors of the severity rating and are negatively correlated with the severity rating.

We assigned numeric values to the six attributes and the severity rating to generate a data set for performing ordered logistic regression. We imposed total orderings among the method access rights levels, channel access rights levels, data item access rights levels, and the severity ratings and we assigned numeric values according to the total ordering; we assigned numeric values on an ordinal scale. We, however, could not impose a total ordering among channel protocols. Hence we assigned numeric values on a *nominal* scale, i.e., the values are useful only for classification and not for ordering [30]. Assignment of nominal values implies that we considered each protocol to be an attribute on its own instead of considering channel protocol as one attribute corresponding to all protocols. Since nominal values are not ordered, we cannot increase or decrease them; hence we cannot determine a positive or a negative correlation with the severity rating.

In case of the data items, 108 observations mentioned data items of only the `file` type. Since all these observations have the same numeric value for the data type attribute, our initial logistic regression analysis did not include the data type attribute in the analysis. Hence we identified each data item's file format (e.g., doc and html) and assigned numeric values to the file formats on a nominal scale. We considered each file format to be an attribute on its own instead of considering data item type as one attribute.

Methods			
Attribute	Coefficient	Standard Error	p-value
Privilege	0.948	0.236	$p < 0.001$
Access Rights	-0.584	0.110	$p < 0.001$

Table 5.3: Significance of the method privilege and access rights.

Channels			
Attribute	Coefficient	Standard Error	p-value
SMTP	2.535	0.504	$p < 0.001$
TCP	0.957	0.466	$p = 0.040$
Pipe	0.948	0.574	$p = 0.099$
Access Rights	-0.312	0.109	$p = 0.004$

Table 5.4: Significance of the channel protocol and access rights.

Data Items			
Attribute	Coefficient	Standard Error	p-value
HTML	-0.651	0.263	$p = 0.013$
DHTML	-0.589	0.437	$p = 0.177$
ActiveX	1.522	0.480	$p = 0.002$
WMF	46.314	2.58e+09	$p = 1.000$
Doc	-1.123	0.462	$p = 0.015$
Access Rights	-0.310	0.078	$p < 0.001$

Table 5.5: Significance of the data item type and access rights.

From Table 5.3, the privilege attribute's positive coefficient shows that privilege is positively correlated with the severity rating. The p-value shows that privilege is a significant predictor of the severity rating. Similarly, access rights is negatively correlated with the severity rating and is a significant predictor of the severity rating.

We identified the top three frequently mentioned protocols in the data set and tested for the significance of the three protocols with respect to other protocols in explaining the severity rating. We show the results in Table 5.4. The p-values show that both `SMTP` and `TCP` are significant and `pipe` is insignificant in explaining the severity rating. Since two of the three protocols are significant, the finding suggests that channel protocol is a significant predictor of the severity rating. Note that the signs of the regression coefficients do not imply any correlation with the severity rating as we assigned nominal values to channel protocols. Table 5.4 also shows that similar to method access rights, channel access rights is negatively correlated with the severity rating and is a significant predictor of the severity rating.

We identified the top five frequently mentioned file formats in the data set and tested for the significance of the five formats with respect to the other formats in explaining the severity rating. We show the results in Table 5.5. The p-values show that `HTML`, `ActiveX`, and `Doc` are significant and `DHTML` and `WMF` are insignificant in explaining the severity rating. The findings suggest that file format is a significant predictor of the severity rating and hence data item type is a significant predictor of the severity rating. Similar to channel protocols, the signs of the regression coefficients do not imply any correlation with the severity rating. Table 5.5 also shows that similar to method access rights and channel access rights, data access rights is negatively correlated with the severity rating and is a significant predictor of the severity rating.

5.3 Expert User Survey

Statistical survey is a widely used technique in social and decision sciences [92, 85, 31]. A survey is an efficient way to collect a wide range of information from a large number of

participants; we can measure the participants' perception, attitude, beliefs, and behaviors using surveys.

Use of surveys and user studies for empirical validation is not new in software engineering research [73, 50, 16, 67]. We conducted an expert user survey as our second empirical study for two reasons. First, we wanted to find out the perception of potential users of our metric. The knowledge would help us identify improvements to the metric to make the metric more useful and widely accepted in practice. Second, as discussed in the previous section, we did not find an existing data set that is relevant to a resource's likelihood of being used in attacks. Hence we collected relevant data from the survey to validate hypothesis 3. We also validated the other two hypotheses using the survey data.

Software developers and software consumers are two potential user groups of our metric. System administrators are examples of software consumers; they can use attack surface measurements to compare competing and alternative software. Hence we conducted an email survey of expert Linux system administrators to know their perception of the metric. We also collaborated with software developers and and got their feedback on improving the metric; we discuss the collaboration in Chapter 6.

5.3.1 Subjects

We identified twenty system administrators as the subjects of our survey. We chose the subjects from diverse backgrounds to avoid any bias: fifteen of them work in ten universities, four of them work in four corporations, and one works in a government agency. Nineteen of the subjects are geographically distributed over the US and one is based in Europe. We also chose experienced system administrators who were knowledgeable about software security in order to obtain reliable responses. Six of the subjects have 2-5 years of full time experience of managing Linux systems; eleven, 5-10 years of experience, and the remaining three, more than 10 years of experience. The subjects have installed and maintained software such as web servers, IMAP servers, and database servers on Linux. The subjects either have implemented or possess the knowledge to implement software attacks such as buffer overflow exploitation, format string exploitation, and cross-site scripting

attacks.

5.3.2 Questionnaire

The survey questionnaire consisted of six explanatory questions. We designed the questions to measure the subjects' attitude about the steps in our attack surface measurement method. The first five questions asked the subjects to indicate their degree of agreement or disagreement with the steps in our measurement method. We also asked the subjects to explain the reasons behind their choices and to suggest alternative ways to carry out the steps in our method.

We used the last question to collect information about the experience and expertise of our subjects. We used the information to avoid self-selection bias, i.e., the subjects incorrectly consider themselves expert system administrators without relevant experience or expertise. We analyzed the responses from only those system administrators who had at least two years of experience in system administration, had installed and maintained many software applications, and were knowledgable about attacks on Linux software.

The subjects indicated their attitude toward the steps of our measurement method on a five-point Likert scale [54]. The Likert scale is the most commonly used psychometric response scale in survey research [85]. A five-point Likert scale has five categories for indicating the degree of agreement: strongly agree, agree, neither agree or disagree, disagree, and strongly disagree. The bipolar scaling nature of the Likert scale allows us to measure both positive and negative responses.

We conducted six rounds of pretesting the questionnaire to identify and remove leading questions, ambiguous terms, and overall confusing questions from the questionnaire [92]. After each round of pretesting, we interviewed the participant and refined our questions based on the participants' feedback. Please see Appendix B for the questionnaire.

5.3.3 Data Collection and Analysis

We cannot assume that the subjects of our survey consider all pairs of adjacent categories on the Likert scale to be equidistant. Hence the data collected from the subjects' responses are ordinal data. Ordinal data is analyzed using descriptive techniques such as collecting summary statistics in the form of median, mode, percentile, and frequency and collating the data into summary graphical tools such as frequency tables, bar charts, and pie charts.

Likert scale responses may suffer from central tendency bias, i.e., subjects may avoid using extreme response categories such as "strongly agree". Hence for each question, we combined the "strongly agree" and the "agree" responses to an "agree (strongly or otherwise)" category and the "strongly disagree" and the "disagree" responses to a "disagree (strongly or otherwise)" category. We computed the proportion of the subjects who agree (strongly or otherwise), disagree (strongly or otherwise), and are neutral with the steps in our measurement method.

We performed one sample t-tests to determine the statistical significance of the survey responses. We used the one sample t-test to test the null hypothesis that the mean of a survey question's Likert scale responses is "neutral". The one sample t-test computes the mean of the Likert scale responses obtained from the subjects and compares the mean with the mean given in the null hypothesis. We chose a p-value of 0.05 as the threshold; hence responses with p-values less than 0.05 are statistically significant.

5.3.4 Results and Discussion

We below summarize the findings of the survey.

1. Methods, channels, and data are the dimensions of a system's attack surface (Table 5.6).

2. A resource's damage potential-effort ratio is an indicator of the likelihood of the resource being used in attacks (Table 5.7).

3. A method's privilege is an indicator of damage potential (Table 5.8, row 1) and a method's access rights (Table 5.8, row 4), a channel's access rights (Table 5.8, row 5) , and a data item's access rights (Table 5.8, row 6) are indicators of attacker effort.

4. The findings are not conclusive with respect to channel protocol (Table 5.8, row 2) and data item type (Table 5.8, row 3) as being indicators of damage potential.

Our first set of questions probed the subjects about their perception of our choice of methods, channels, and data as the dimensions of the attack surface. We asked the subjects whether a system's methods, channels, and data can be used in attacks on the system. We show the percentage of the subjects who agree, disagree, and neither agree or disagree with our choice in Table 5.6. The findings show that a majority of the subjects agree with our choice of the dimensions; the p-values show that the findings are statistically significant. The findings do not rule out the existence of other dimensions. None of the subjects, however, suggested a different dimension of the attack surface.

Dimension	Agree	Disagree	Neutral	p-value
Methods	95%	0%	5%	$p < 0.0001$
Channels	95%	0%	5%	$p < 0.0001$
Data	85%	0%	15%	$p < 0.0001$

Table 5.6: A majority of the subjects agree with our choice of the dimensions.

Our second set of questions asked subjects whether a resource's damage potential-effort ratio is an indicator of the resource's likelihood of being used in attacks. We show the responses in Table 5.7. A majority of the subjects agree with our choice of the damage potential-effort ratio and the finding is statistically significant. Also, none of the subjects suggested any other indicator or measure of the likelihood. We conclude that a resource's damage potential-effort ratio is an indicator of the resource's likelihood of being used in attacks.

Our third set of questions probed the subjects about their perception of our choice of the six resource attributes as indicators of damage potential and attacker effort. We show the percentage of the subjects who agree, disagree, and neither agree or disagree with

	Agree	Disagree	Neutral	p-value
Damage Potential-Effort Ratio	70%	20%	10%	p = 0.0141

Table 5.7: A majority of the subjects agree with our notion of the damage potential-effort ratio.

	Attribute	Agree	Disagree	Neutral	p-value
Damage Potential	Method Privilege	90%	0%	10%	$p < 0.0001$
	Channel Protocol	45%	25%	30%	p = 0.2967
	Data Item Type	45%	5%	50%	p = 0.8252
Attacker Effort	Method Access rights	70%	5%	25%	p = 0.0001
	Channel Access Rights	75%	5%	20%	$p < 0.0001$
	Data Item Access Rights	85%	10%	5%	$p < 0.0001$

Table 5.8: Perception of the subjects about the choice of our attributes.

our choice of the attributes in Table 5.8. A majority of the subjects agree that a method's privilege is an indicator of the method's damage potential and that a method's access rights, a channel's access rights, and a data item's access rights are indicators of attacker effort. The p-values show that the findings are statistically significant. We conclude that these four attributes are indicators of damage potential and effort.

Though a majority of the subjects agree that a channel's protocol is an indicator of the channel's damage potential, the p-value shows that the result is not statistically significant. Similarly, though a majority of the subjects are neutral with respect to a data item's type being an indicator of damage potential, the finding is not statistically significant. Hence the findings of the survey are not conclusive with respect to our choice of a channel's protocol and a data item's type as indicators of damage potential.

The subjects who disagreed with our choice were of the opinion that a channel's or a data item's damage potential is dependent on a system's methods that process the data received from the channel or the data item; hence they concluded that a `TCP socket` and an `RPC end point` are equally attractive to an attacker irrespective of their protocol. Similarly, a `file` and a `cookie` are equally attractive irrespective of their type. These

findings suggest that we should assign the same damage potential, i.e., 1, to all channels and data items. In that case, we do not have to perform the difficult step of assigning total orderings among the channel protocols and the data item types.

5.4 Correlation between Patches and Attack Surface Measurement

In this section, we demonstrate that patches of security vulnerabilities in open source software reduce software's attack surface measurements.

Not all patches, however, are relevant to a system's attack surface measurement. Intuitively, a patch is relevant if we expect the patch to affect the attack surface's three dimensions or the six resource attributes. For example, we expect a patch that resolves authentication issues to affect the access rights levels of resources; hence the patch is relevant. We, however, do not expect the patches for buffer overruns, integer overflows, and dangling pointer references to affect the attack surface measurement; hence the patches are not relevant.

Moreover, not all "relevant" patches reduce a system's attack surface measurement. We can patch a vulnerability in different ways. For example, we can patch a format string vulnerability by either parsing the input or by appending a %s to the vulnerable line of code. The first patch reduces the attack surface, but the second does not.

5.4.1 Patches Relevant to Attack Surface Measurement

In this section, we present an informal definition of relevant patches. We also introduce a set of heuristics to decide whether a vulnerability's patch is relevant to a system's attack surface measurement.

Informal Definition of a Relevant Patch

A patch is relevant to a system's attack surface measurement if we expect the patch to remove a vulnerability by modifying the number of resources that are part of the system's attack surface or by modifying the damage potential-effort ratios of such resources. In our analysis, we focus on patches written in the C and C++ programming languages. A patch implemented in C or C++ is relevant if we expect the patch to make one or more of the following changes to a system.

1. Modification of the number of entry points, exit points, channels, and untrusted data items: The patch either (a) adds (removes) methods, channels, and untrusted data items to (from) the system, or (b) restricts the flow of data from the system's environment into the system or vice versa.

2. Modification of the damage potential-effort ratio of an entry point or an exit point: The patch modifies the privilege or access rights of some parts of a system's source code.

3. Modification of the damage potential-effort ratio of a channel: The patch modifies a channel's protocol or access rights.

4. Modification of the damage potential-effort ratio of an untrusted data item: The patch modifies an untrusted data item's type or access rights.

Heuristics to Identify Relevant Patches

We use a vulnerability's National Vulnerability Database (NVD) bulletin to decide whether the vulnerability's patch is relevant [78]. The NVD is the U.S. government repository of software vulnerability data; the repository contains bulletins for security vulnerabilities discovered in both open source and commercial software. Each bulletin contains a technical description of the vulnerability, a list of affected software, metrics on the impact of the vulnerability's exploitation, and links to external advisories, patches, and tools for the

vulnerability. Each bulletin also contains a *type* for the vulnerability described in the bulletin; the type is assigned using the Common Weakness Enumeration (CWE) developed at MITRE [72]. Please see Appendix C for a complete list of vulnerability types included in the CWE.

The CWE definitions of vulnerability types provide information on how to find vulnerabilities in software and how to deal with discovered vulnerabilities [76]. Hence we use a vulnerability's type to decide whether we expect the vulnerability's patch to make the changes mentioned in the previous section and hence whether the patch is relevant to the attack surface measurement. We identify the following types to be relevant to the attack surface measurement: Authentication Issues, Permissions, Privileges, and Access Control, Cross-Site Scripting (XSS), Format String Vulnerability, SQL Injection, Operation System (OS) Command Injection, Information Disclosure.

Unfortunately, type information is absent in a large percentage of the NVD bulletins. In the absence of type information, we infer the type from the vulnerability's description included in its NVD bulletin. If a vulnerability belongs to one of the last five types, we can unambiguously determine the vulnerability's type from its description. It is, however, difficult to infer whether a vulnerability has one of the first two types from its description. In such cases, we use the *Authentication* and *Impact Type* information in an NVD bulletin to infer type information; we look for the occurrence of the words *privilege, access rights,* and *authentication* in the bulletin. The occurrences of these words indicate that the vulnerability described in the bulletin has one of the first two types. Our inference process is manual and hence is subject to human error.

Not all relevant patches reduce the attack surface measurement. If a vulnerability has one of the first two types, we expect the vulnerability's patch definitely to reduce the attack surface. The vulnerabilities belonging to the last five types can be patched in different ways; hence the patches may not always reduce the attack surface.

5.4.2 Results and Discussion

In our experiment, we analyzed the source code of a patch to measure the change in the attack surface due to the patch. We followed the measurement method introduced in Chapter 4. We considered the effect of a patch in isolation from the rest of the system.

We analyzed all the patches released by the Mozilla Foundation for their Firefox 2.0 web browser [33]. We also analyzed the patches of all vulnerabilities found in the NVD for all versions of the ProFTP server [84]. Our results indicate that in case of Firefox 2.0, 67% of the relevant patches reduced Firefox's attack surface measurement. Similarly, in case of the ProFTP server, 70% of the relevant patches reduced the ProFTP server's attack surface measurement. The p-values of t-tests indicate that our results are statistically significant (Confidence Interval = 95%, $p < 0.05$).

Both in case of Firefox and ProFTP, we had to infer many vulnerabilities' missing type information. The inference process is subject to human error. Hence we repeated our experiment by analyzing only those patches that have a type assigned in the NVD. Our results show that 76.9% of the relevant patches reduced the attack surface measurement. The p-values of t-tests indicate that our results are statistically significant (Confidence Interval = 95%, $p < 0.05$). Hence we conclude that a majority of the relevant patches reduce a system's attack surface measurement.

Firefox

Our primary source of data for Firefox 2.0 was the Mozilla Foundation Security Advisories published by the Mozilla Foundation [35]. Each advisory describes one or more vulnerabilities in Firefox and contains links to the NVD bulletins for the vulnerabilities. We used type information in the NVD bulletins to decide whether the patches of the vulnerabilities were relevant to the attack surface measurement. The advisory also contains links to the Firefox Bugzilla database entries for the vulnerabilities [32]. A vulnerability's Bugzilla database entry contains the source code of the vulnerability's patch; we analyzed the code to measure the change in the attack surface due to the patch. Figure 5.1 shows our data collection process.

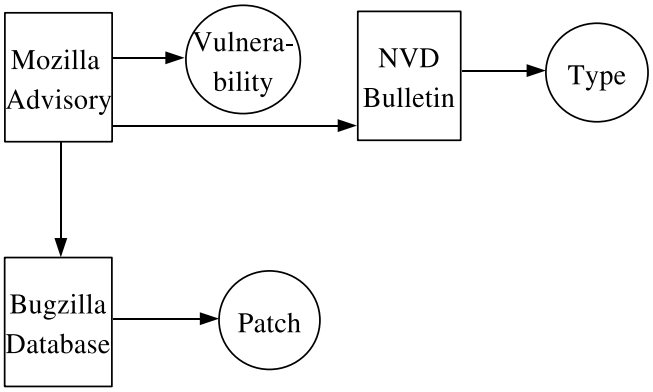

Figure 5.1: Data collection process for Firefox.

We collected data from the Mozilla Foundation Security Advisories published for Firefox versions 2.0.0.1 to 2.0.0.8. There are forty-five advisories published over a period of eleven months from December 2006 to October 2007. These advisories contain links to fifty-three NVD bulletins describing fifty-three unique vulnerabilities. Four out of the fifty-three are JavaScript vulnerabilities; also, the patch source code of one is not publicly available. Hence we did not consider these five vulnerabilities in our analysis.

A majority of the forty-eight NVD bulletins for the remaining forty-eight vulnerabilities did not have any type information. We obtained a dataset from MITRE that complemented type information of NVD bulletins [18]. This dataset, however, was incomplete and we could not obtain type information for all the Firefox NVD bulletins; hence we inferred the types from the description of the bulletins. We followed the process described in Section 5.4.1 and used the explicit and inferred type information to identify the patches relevant to the attack surface measurement.

We identified twelve out of the forty-eight patches to be relevant to Firefox's attack surface measurement. Eight out of these twelve patches reduced Firefox's attack surface measurement; the remaining four did not change the attack surface measurement. Three out of the four patches that did not reduce the attack surface are patches of three XSS vulnerabilities. We do not expect the patches of XSS vulnerabilities to always reduce the attack surface.

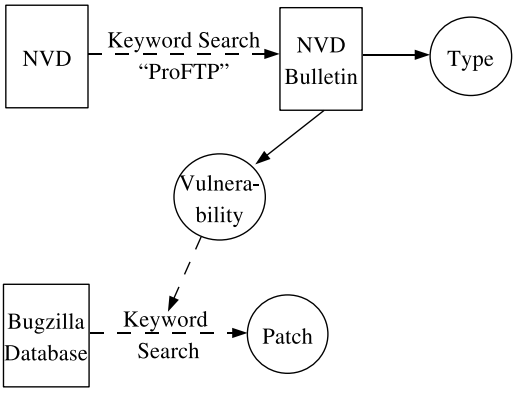

Figure 5.2: Data collection process for ProFTP.

ProFTP

The ProFTPD project group maintains the ProFTP server [84]. Unlike the Mozilla Foundation, the group does not publish any public security advisories for the ProFTP server. Hence we collected the vulnerability list for the ProFTP server from the NVD. We obtained twenty five NVD bulletins by searching for the phrase "ProFTP" in the NVD. Three out of these twenty-five bulletins were not ProFTP vulnerabilities.

The remaining twenty-two NVD bulletins describe twenty-two vulnerabilities in the ProFTP server. We searched the ProFTP Bugzilla database for the patches of these vulnerabilities [83]. We could locate the patches for twenty-one of these vulnerabilities; we could not locate the patch for one. Figure 5.2 shows our data collection process.

A majority of the twenty-one NVD bulletins for the twenty-one vulnerabilities did not have any type information. The dataset that we obtained from MITRE also did not have type information for all the twenty-one bulletins. Hence we inferred the types from the description of the bulletins as described in Section 5.4.1 and identified the patches relevant to ProFTP's attack surface measurement.

We identified ten out of the twenty-one patches to be relevant to ProFTP's attack surface measurement. Seven out these ten patches reduced ProFTP's attack surface measurement. The three patches that did not reduce the attack surface are patches of three

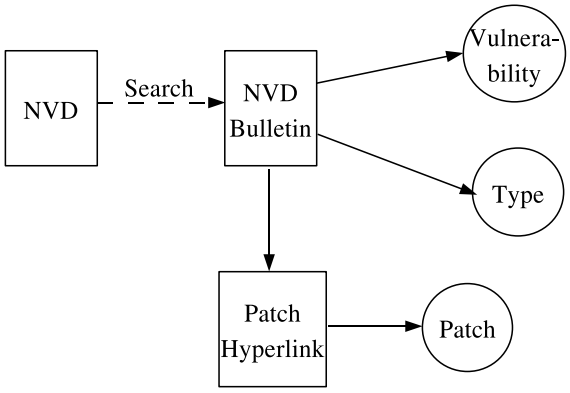

Figure 5.3: Data collection process for NVD bulletins that have a type assigned.

format string vulnerabilities. We do not expect the patches of format string vulnerabilities to always reduce the attack surface.

Analysis of Patches that have a Type Assigned

In this section, we report the results of analyzing only those patches that have a type assigned in NVD. An NVD bulletin contains hyperlinks to the patches of the vulnerability described in the bulletin; these hyperlinks are labeled as *patch information* in the bulletin. We require a patch's source code in our analysis; hence we identified the NVD bulletins that have a type assigned and that contain one or more hyperlinks labeled as patch information. Figure 5.3 shows our data collection process.

The NVD contains 25,000 bulletins; only 363 bulletins, however, have a type assigned to them and contain one or more hyperlinks to their patches. We identified 113 bulletins out of the 363 bulletins to be relevant to the attack surface measurement based on their types. Out of the 113 bulletins, seventy-three bulletins describe vulnerabilities in software implemented in `C, C++,` and `Java`; the remaining forty bulletins describe vulnerabilities in software implemented in other languages such as `PHP, Perl,` and `Ruby`. Hence we did not include these forty bulletins in our analysis.

We could, however, obtain the source code of the patches of only thirteen out of

seventy-three bulletins by following the hyperlinks labeled as patch information. In the case of the remaining sixty bulletins, the hyperlinks labeled as patch information point to downloadable patches in binary format; the source code of the patches are not publicly available (e.g., in the case of patches of commercial software).

We measured the change in the attack surface due to the thirteen patches. Ten out of these thirteen patches reduced the attack surface. One of the three patches that did not reduce the attack surface is a patch for a format string vulnerability. We do not expect the patches of format string vulnerabilities to always reduce the attack surface. The other two patches used cryptographic techniques to remove the vulnerability in the code and hence did not reduce the attack surface.

Discussion

In this section, we make three qualifying remarks on our analysis.

First, a few steps in our analysis are subject to human error. For example, MITRE's analysts could have introduced errors either by assigning wrong types to vulnerabilities or by not assigning types that should have been. We could have inferred the wrong types for the NVD bulletins missing type information. Hence we could have wrongly classified some patches as relevant and could have excluded some relevant patches from the analysis.

Second, the Firefox and ProFTP results obtained after type inference is similar to the result obtained from analyzing only those patches that have a type assigned. Hence our type inference process produces similar results as MITRE's type assignment process.

Third, the relevant patches that reduce a system's attack surface remove vulnerabilities that pose significant security risk. Each NVD bulletin contains a severity rating for the vulnerability described in the bulletin. Three different ratings, *High, Medium,* and *Low*, are assigned based on the vulnerability's Common Vulnerability Scoring System (CVSS) score [77]; the higher the rating, the higher the security risk. In the case of Firefox, the eight patches that reduced the attack surface removed vulnerabilities with High and Medium ratings. Similarly, the seven patches that reduced ProFTP's attack surface also removed vulnerabilities with High and Medium ratings. Finally, in the case of patches that

have a type assigned, nine out of the ten patches that reduced the attack surface removed vulnerabilities with High and Medium ratings.

5.5 Anecdotal Evidence

In this section, we present anecdotal evidence from the software industry to demonstrate that reduction in a software system's attack surface mitigates the system's security risk. We describe three examples from Microsoft and one from Firefox to illustrate the benefit of attack surface reduction.

The Sasser worm is a computer worm that affected computers running vulnerable versions of the Microsoft Windows operating system [105]. The worm exploited a buffer overflow vulnerability present in the Local Security Authority Subsystem Service (LSASS) of the Microsoft Windows operating system [69]. The entry point to exploit the vulnerability is an RPC interface that is accessible by everyone in Windows 2000 and Windows XP; the interface, however, is accessible only by local administrators in Windows Server 2003 due to the attack surface reduction process [46]. The worm could easily spread to a vulnerable host running Windows 2000 or Windows XP, but it could not spread to a host running Windows Server 2003 because of the higher access rights level of the entry point. Hence the worm did not affect Windows Server 2003 due to the smaller attack surface of Windows Server 2003.

Similarly, the Zotob worm exploited a buffer overflow vulnerability present in the Plug and Play (PnP) service of the Microsoft Windows operating system [70]. The entry point to exploit the vulnerability is an RPC interface that is accessible by everyone in Windows 2000. The interface is accessible only by remote authenticated users in Windows XP SP1 and local authenticated users and remote administrators in Windows XP SP2 and Windows Server 2003 due to the attack surface reduction process [46]. The worm could easily spread to a vulnerable host running Windows 2000, but no other versions of Windows because of the higher access rights level of the entry point. Hence the Zotob worm affected only Windows 2000; no other version of Windows was affected due to a smaller attack surface.

One of the vulnerabilities exploited by the Nachi worm is a buffer overflow vulnerability present in the core component `ntdll.dll` of the Microsoft Windows operating system [68]. The vulnerability can be exploited by sending an ill-formed Web-based Distributed Authoring and Versioning (WebDAV) request to an Internet Information Server (IIS). The entry point to exploit the vulnerability is the IIS service in Windows 2000 and the `w3wp.exe` process (World Wide Web Worker Process) in Windows Server 2003. The IIS service runs with administrative privilege whereas the `w3wp.exe` process runs with the restrictive network-service privilege as part of the attack surface reduction process [46]. Windows Server 2003 does not have the buffer overflow vulnerability; but even if it had the vulnerability, the potential scope for damage would have been limited due to its smaller attack surface.

Both Firefox 2.0 and Firefox 1.5 contained a buffer overflow vulnerability in the Mozilla Network Security Services (NSS) code for processing the SSL 2 protocol [34]. As part of the attack surface reduction process, SSL 2 protocol was turned off in the default configuration of Firefox 2.0 [61]. The attack surface reduction process removed the entry point required to exploit the vulnerability in Firefox 2.0. Hence Firefox 2.0 is immune to attacks that exploit the buffer overflow vulnerability, whereas Firefox 1.5 is not.

5.6 Theoretical Validation By Liu and Traore

In this section, we briefly describe Liu and Traore's theoretical validation of the attack surface metric [56, 57]. Liu and Traore introduce a set of measurement properties for software security metrics and verify that the attack surface metric possesses the set of properties. Hence the attack surface metric is a valid measure of security in their framework.

Liu and Traore identify four *internal* software attributes that are related to software security. A software system's internal attributes are those that can be measured purely in terms of the software [30]. They derive the four attributes from the collective security design principles accepted by the community, e.g., Saltzer and Schroeder's design principles for protection mechanisms [87]. They introduce a *service-oriented model* of software and

represent a software system as a collection of services working in concert. They identify the following four internal attributes that are related to security in their model:

1. service complexity: represents the level of complexity in software;
2. service coupling: represents the level of dependency between the services of software;
3. excess privilege: represents the excess privilege granted to the services; and
4. mechanisms strength: represents the combined strength of security mechanisms protecting the services.

They also identify the relationship between software security and the four attributes using their intuitive understanding of the security design principles. They expect software security to decrease as service complexity increases, service coupling increases, excess privilege increases, and mechanisms strength decreases.

For each of the four internal attributes, they identify a set of measurement properties that any valid measure of the attribute must possess. The set of properties are sound but not complete. They identify the properties from the security design principles and the relationship between software security and the attributes. For example, two of the measurement properties for service complexity state that a service's complexity is non-negative and a software system's complexity is no less than the complexity of any of the system's services. They identify similar properties for service coupling, excess privilege, and mechanism strength.

They also show that our attack surface metric can be easily expressed in their service-oriented model and derive three measures of service complexity, service coupling, and excess privilege from the attack surface metric. They demonstrate that the three measures possess the necessary measurement properties; hence the measures derived from the attack surface metric are valid measures. They also demonstrate the validity of a measure of mechanisms strength derived from Dacier et. al's Mean-Time-To-Failure metric [24].

Chapter 6

Attack Surface Measurement of SAP Business Applications

6.1 Introduction

In Chapter 4, we applied our attack surface measurement method to small software systems such as FTP servers and IMAP servers. These systems are small in their code size and simple in their architectural design. In this chapter, we study the applicability of our measurement method to large and complex enterprise-scale software systems.

We collaborated with SAP to measure the attack surfaces of SAP's enterprise software applications. SAP is the world's largest enterprise software company with more than 46,100 customers worldwide [3]. SAP provides a comprehensive range of enterprise software and business applications covering all aspects of the customers' businesses [4]. The business applications have a varied customer base of small, medium, and large enterprises and support a wide range of business functionalities; hence the applications are large in size and complex in their design.

Our motivation behind this collaboration was two fold. First, we demonstrated that our measurement method scales to enterprise-scale software. Second, we had the opportunity to interact closely with SAP's software developers and architects and get their feedback on

our measurement process. Based on their feedback, we identified a set of improvements to our measurement process to make the process more useful in practice. We discuss these future steps in Section 6.6.

SAP also had two goals behind this collaboration. The short-term goal was to measure an enterprise software system's attack surface and to identify possible ways to reduce the attack surface. The long-term goal was to evaluate the possibility of integrating our measurement process with SAP's software development process to improve business application security. SAP has been focusing on code quality improvement to make their software more secure [7]; we, however, believe that a complete security risk mitigation approach requires a combination of the code quality effort and the attack surface reduction approach. Hence it is crucial to integrate the attack surface measurement process with the software development process.

The rest of this chapter is organized as follows. In Section 6.2, we describe our choice of the enterprise software system used in the collaboration. We introduce a method to measure the attack surfaces of SAP systems in Section 6.3. We describe the implementation of a tool to measure the attack surface in Section 6.4. We discuss the measurement results in Section 6.5. In Section 6.6, we discuss various lessons learned from the collaboration and conclude with a discussion of future work.

6.2 Choice of an Enterprise Software System

Choosing an appropriate software system was a vital step in our collaboration with SAP. The choice of a system was guided by three requirements. First, the chosen system should be a heavily used system so that reduction in its attack surface benefits a large number of SAP customers by reducing their security risk. Second, the chosen system should be representative of SAP's software systems; measuring the system's attack surface will guide us in measuring the attack surfaces of other SAP systems. Third, the product development group in charge of the system should be committed to the collaboration.

We had discussions with six different SAP product development groups before making

our choice. We identified a component of the SAP NetWeaver platform as the enterprise software system to be used in our collaboration [5]. SAP NetWeaver is the common technology platform for SAP business applications; the platform provides the run time environment for all SAP applications. The platform enables development, life cycle management, composition, and integration of SAP business applications [6]. Our chosen component is a core building block of the NetWeaver platform and provides a critical service inside the platform. Henceforth, we refer to the chosen component as *the service*.

We chose to measure the attack surface of the service due to the following three reasons. First, the service is used by many SAP customers as it is a core part of the NetWeaver platform. Second, the architectural design of the service is also very similar to other SAP applications. Third, the product development team in charge of the service committed the necessary resources to collaborate with us.

6.3 Measurement Method for SAP Software Systems

In this section, we introduce a method to measure the attack surfaces of SAP software systems. A majority of SAP systems are implemented in `Java`, `JavaScript`, and `ABAP` (a proprietary SAP language). In this collaboration, we measure the attack surfaces of systems implemented in `Java`. We leave the measurement methods for systems implemented in `JavaScript` and `ABAP` for future work.

In Chapter 3, we introduced an abstract attack surface measurement method based on our formal I/O automata model. Here, we instantiate the abstract method and obtain a new method to measure the attack surface of SAP systems implemented in `Java`. Our new method has the same steps as the `C` measurement method introduced in Section 4.2. The implementations of the steps, however, are different for the method dimension; the channel dimension and the data dimension remain unchanged.

Recall that the two key steps in measuring the attack surface along the method dimension are (1) the identification of the entry points and the exit points (Section 6.3.1), and (2) the estimation of the damage potential-effort ratio of the entry points and the exit

points (Section 6.3.2). We describe these two steps of our new method in the following paragraphs.

6.3.1 Identification of Entry Points and Exit Points

A direct entry point of a system is a method that receives data items directly from the system's environment. A method, m, of a system, s, implemented in `Java` can receive data items in three different ways: (a) m is a method in s's public *interface* and receives data items as input, (b) m invokes a method in the interface of a system, s', in the environment and receives data items as result, and (c) m invokes a `Java I/O` library method. For example, a method, m, is a direct entry point if m invokes the `read` method of the `java.io.DataInputStream` class.

A direct exit point of a system is a method that sends data items directly to the system's environment. A method, m, of a system, s, implemented in `Java` can send data items in three different ways: (a) m is a method in s's public interface and sends data items as result, (b) m invokes a method in the interface of a system, s', in the environment and sends data items as input, and (c) m invokes a `Java I/O` library method. For example, a method, m, is a direct exit point if m invokes the `write` method of the `java.io.DataOutputStream` class.

Given a system, s, we generate s's call graph starting from the methods in s's public interface. From the call graph, we identify all methods of s that invoke either a method in the interface of a system, s', in s's environment or a `Java I/O` library method. These methods are s's direct entry points and direct exit points.

We implemented a tool, described in Section 6.4, to measure the attack surfaces of SAP systems in an automated manner. The tool identifies only direct entry points and direct exit points of a system. As discussed in Section 4.2.2, we leave the identification of the indirect entry points and the indirect exit points as future work. Hence our measurement is an under-approximation of the measure of the attack surface.

6.3.2 Estimation of the Damage Potential-Effort Ratio

In the previous chapter, we estimated a method's damage potential using the method's privilege and the attacker effort using the method's access rights. In case of the SAP systems, however, a method's privilege is not a useful estimate of the method's damage potential. The entire code base of the NetWeaver platform runs with the same privilege, i.e., the privilege of the application server hosting the platform. If we were to estimate the damage potential using the privilege, we could not make any meaningful suggestion to reduce the attack surface. Hence we do not use a method's privilege in estimating the method's damage potential. Instead, we estimate a method's damage potential using the method's *sources of input data (destinations of output data)*. A method can receive (send) data items from (to) three sources: an input parameter, the data store, and other systems present in the environment. For example, a method receives data items from an attacker as an input parameter in case of SQL injection attacks and Cross Site Scripting (XSS) attacks whereas the method receives data items from the data store in case of File Existence Check attacks. Methods in SAP systems can have three different sources of input: `parameter`, `data store`, and `other systems`.

Similar to systems implemented in `C`, we use a method's access rights level to estimate the attacker effort. A typical SAP system has two different types of interfaces: (1) public interfaces that can be accessed by all entities belonging to any NetWeaver *role* and (2) internal interfaces that can be accessed by only other components of the NetWeaver platform. Hence the methods in SAP systems can be accessed with two different access rights levels: `public` access rights level for methods in public interfaces and `internal` access rights level for methods in internal interfaces.

In the case of an SAP system, we annotate the system's code base to indicate the access rights levels of the system's interfaces. We generate the call graph of the annotated code base and determine the sources of input and the access rights levels of the methods from the call graph. Notice that a method may have multiple sources of input. Similarly, a method may be accessible with multiple access rights levels. We identify such a method as a direct entry point (direct exit point) multiple times.

Similar to systems implemented in `C`, we impose total orderings among the sources of input and the access rights levels and assign numeric values to the sources of input and access rights levels in accordance to the total orderings. The exact choice of the numeric values is subjective and depends on a system and its environment. We discuss a specific way of assigning the numeric values in case of the service in Section 6.4.3. We estimate a method's damage potential-effort ratio from the numeric values assigned to the method's source of input and access rights level.

6.4 Implementation of a Measurement Tool

In this section, we describe a tool we implemented to measure the attack surfaces of SAP systems implemented in `Java`. There are two key objectives that guided the implementation of the tool: (1) the tool should be an integral part of the software development process, and (2) the software developers and architects can use the tool easily and frequently without spending too much time and effort. The software developers and architects at SAP are already under time pressure; hence it is crucial for the adoption of the tool that we do not burden them with more work and we do not require them to go out of their normal routine to use the tool.

SAP's software developers use a customized version of the *Eclipse* Integrated Development Environment (IDE) for their software development activities [27]. We implemented our tool as a *plugin* for the Eclipse IDE so that the developers can use the tool inside their software development environment. We show a screen shot of our tool in Figure 6.1 and describe the tool in details in the following paragraphs.

6.4.1 Call Graph Generation

A key component of our tool is the generation of a system's call graph from the system's source code. We use two different techniques to generate the call graph to provide a precision-scalability tradeoff to the software developers: the TACLE Eclipse plugin developed at the Ohio State University, which gives a very precise call graph, but does not scale

Figure 6.1: Screenshot of the Attack Surface Measurement tool implemented as an Eclipse plugin.

well to large programs [39]; and an Eclipse API, which gives a less precise call graph, but scales [28].

The TACLE plugin provides us an off-the-shelf implementation of the *Rapid Type Analysis (RTA)* algorithm to perform type analysis and construct call graphs [10]. The plugin performs the operations on a *Java project* inside a *Java Development Tools (JDT)* environment. The plugin implements the RTA algorithm using *Abstract Syntax Tree (AST)* analysis [93]; for each `Java` class included in the source code, the algorithm constructs and stores the AST of the class in main memory. Hence the algorithm does not scale well to large software systems that contain a large number of `Java` classes. Moreover, to construct a complete call graph, the plugin requires the source code of all the `Java` libraries and `Java Archives (JAR)` used for the compilation of the code base. In practice, we do not expect the software developers to possess the source code of the libraries required by their software. For example, the service requires 23 SAP JARs for its compilation. It is unrealistic to expect the developers of the service to include the source code of these 23 JARs in their `Java` development project.

The Eclipse IDE has an internal API for the construction of call graphs from `Java` source code. The API is included in the `org.eclipse.jdt.internal.corext.callhierarchy` package of the IDE [28]. The API does not require the source code of the libraries and JARs used by a system's code base; hence scales well to large code bases. The constructed call graph, however, is less precise than the call graph generated by the TACLE plugin. For example, the API does not resolve an interface method to a class method that implements the interface method. We extended the API by finding all class methods that implement an interface method and then including the implementing methods in the call graph. The resulting call graph, however, is an over-approximation of the actual call graph as only one of the class methods will be invoked in place of the interface method. Hence our measurement is an over-approximation of the measure of the attack surface.

6.4.2 Entry Points and Exit Points Identification

Our tool identifies a system's entry points and exit points from the system's call graph. A method of a system is an entry point (exit point) if the method invokes a method in the public interface of another system present in the environment or a `Java I/O` library method. Hence our tool needs the list of interface methods of other systems and the list of `Java I/O` library methods. We provide the list of methods to the tool through two configuration files: `osmethods.txt` and `dsmethods.txt`.

The service invokes the methods of many SAP systems included in the NetWeaver platform. Invocation of an interface method, however, is only *relevant* to the attack surface measurement if the invocation results in data transfer from the service to other systems in its environment or vice versa. We used the javadoc description of the interfaces and interface methods included in the 23 SAP JAR libraries needed by the service and identified 25 interfaces and 126 interface methods to be relevant to the attack surface measurement. We discussed the methods with the developers of the service to avoid any error and included the methods in the `osmethods.txt` configuration file of our tool.

The service does not read or write any untrusted data items from the data store. Hence there were no `Java I/O` library methods to identify as relevant. But in general, we should provide the list of relevant methods through the `dsmethods.txt` configuration file.

6.4.3 Numeric Value Assignment

Another key component of our tool is the estimation of the numeric damage potential-effort ratios of a system's entry points and exit points. We estimate the damage potential and the attacker effort in terms of the sources of input and the access rights level, respectively. The tool determines the sources of input and the access rights levels from the system's call graph; the tool, however, requires the numeric values assigned to the different sources of input and the access rights levels to estimate numeric damage potential-effort ratios. We provide these numeric values through a configuration file, `weights.txt`.

The methods of the service have three different sources of input: `parameter`, `data store`, and `other system`. As discussed in Section 6.3.2, different types of attacks on the service require the methods to have different sources of input. We assign numeric values to the sources of input by correlating the sources with possible attacks on the service.

SAP conducted a threat modeling process for the service. The process identified possible attacks on the service and assigned severity ratings to the attacks. We correlated the sources of inputs with the possible attacks identified by the threat modeling process. For each source of input, we computed the average severity rating of the attacks that require the source of the input. We show the sources of input in the first column and the average severity ratings in the second column of Table 6.1.

We assigned numeric values to the sources of input in proportion to the average severity ratings. The parameter sensitivity analysis of Section 4.5 suggests that the difference between the numeric values assigned to successive damage potential levels should be in the range of 3-14. Hence we chose the midpoint, 8.5, of the range as the difference. For example, we assigned 1 to the source `other system`, and $1 + (3 - 1) \times 8.5 = 18$ to the source `data store`. We show the numeric values in third column of Table 6.1.

Source of Input	Average Severity Rating	Numeric Value
`other system`	1	1
`data store`	3	18
`parameter`	5	35

Table 6.1: Numeric values assigned to the sources of input.

The methods of the service can be accessed by two different access rights level: `public` and `internal`. We imposed the following total ordering among the access rights level: `internal` > `public`. The parameter sensitivity analysis of Section 4.5 suggests that the difference between the numeric values assigned to successive access rights level should be high (15-20). Hence we chose a difference of 17. We show the numeric values assigned to the access rights level in Table 6.2.

Access Rights Level	Numeric Value
`public`	1
`internal`	18

Table 6.2: Numeric values assigned to the access rights levels.

We use the numeric values shown in Table 6.1 and Table 6.2 to compute the numeric damage potential-effort ratios. For example, consider an entry point, m, of a system, s. m is a method in s's public interface and has two input parameters; m also invokes three interface methods of a system, s', in the environment. Then m's damage potential is $2 \times 35 + 3 \times 1 = 73$. If m is accessible with the `public` access rights level, then m's damage potential-effort ratio is $73/1 = 73$. Similarly, if m is accessible with the `internal` access rights level, then m's damage potential-effort ratio is $73/18 = 4.05$.

6.4.4 Usage of the Tool

The software developers can use our tool to generate a system's call graph rooted at any method of the system and to compute the attack surface measurement from the call graph. The developers can choose the method in the *Outline* view window of the Eclipse IDE. Figure 6.1 shows the two options for measuring the attack surface in the right context menu of the Outline window; the developers can choose the option appropriate for their scalability and precision requirements.

SAP's software systems have multiple interfaces and each interface typically contains multiple methods. To compute the attack surface of an SAP system, we create a new `Java` class with only a `main` method. The `main` method invokes all the methods included in the interfaces of the system. We generate a call graph rooted at the `main` method and compute the attack surface of the system.

The tool generates its output in the form of a text file, containing (1) the system's attack surface measurement, (2) a list of the system's entry points and exit points, and (3) for each entry point (exit point), a list of input sources, the access rights level, and its contribution

to the attack surface measurement. The software developers can use the detailed output as a guide in reducing the attack surfaces of their software. For example, they can focus on the top x% of the entry points and the exit points to reduce the attack surface. They can also focus on the top contributing interfaces and components instead of considering the entire code base of the system.

Our tool allows the developers to perform *incremental* measurements. They can measure the increase in the attack surface due to the addition of a new interface method by generating a call graph rooted at the method; they do not have to measure the attack surface of the entire system. They can also measure the potential reduction in the attack surface due to the removal of an interface method.

The tool also allows the developers to consider many *what-if* scenarios during software development. For example, the developers can easily determine the effect of adding a new feature to the system on the system's attack surface. Similarly, while reducing the attack surface, they can consider the removal of different features and the effect of the removal on the attack surface measurement. They can use the incremental measurements to make an informed decision.

6.5 Results and Discussion

We measured the attack surfaces of three different of the service included in three different versions of the NetWeaver platform. We identify the three versions of the service as `S1`, `S2`, and `S3`. The `S1` version is the first released version of the service, followed by `S2` and `S3` versions, respectively.

We only considered the method dimension of the attack surface in our measurement. The three versions of the service do not use any persistent data items and open only one channel, i.e., a `TCP` socket. Hence we did not measure the attack surface along the channel dimension and the data dimension.

The `S3` version of the service implements 8 public interfaces and 2 internal interfaces. The `S2` and `S1` versions implement 9 and 8 public interfaces, respectively, and no internal

interfaces. We show the number of entry points and exit points of the three versions for each access rights level in Table 6.3.

Version	Count	
	Public	Internal
S3	71	4
S2	67	0
S1	63	0

Table 6.3: The number of entry points and exit points of the three versions of the service for each access rights level.

We estimated the damage potential-effort ratio of each entry point (exit point) as described in Section 6.4.3; the ratio is the contribution of the entry point to the attack surface. We summed up the contributions of the entry points and the exit points to obtain the attack surface measurement along the method dimension. We show the attack surface measurements in Table 6.4. The measurements indicate that the S3 version has the highest security risk along the method dimension followed by S2 and S1.

Version	Attack Surface Measurement
S3	5298.44
S2	4687.00
S1	4649.00

Table 6.4: Attack surface measurements of the three versions of the service.

The S1 version is the first version of the service released to the customers. The S2 version is backward compatible with S1 for the convenience of the customers. Moreover, S2 added new features to S1 resulting in an increase in the number of public interfaces. Hence the set of methods of S2 is a superset of the set of methods of S1 and as shown in Table 6.4, the attack surface measurement of S2 is greater than S1.

The S3 version is the latest version of the service released to the customers. The S3

version differs from the S2 version in two significant ways: (1) S3 converted a public interface of S2 to an internal interface to mitigate security risk, and (2) S3 added new features to the service resulting in an increase in the number of public interfaces and internal interfaces. If no new features were added, the attack surface measurement of S3 would have been smaller than S2 due to the conversion of a public interface to an internal interface. The increase in the number of total interfaces due to the addition of new features, however, increases the attack surface measurement of S3. Hence as shown in Table 6.4, the attack surface measurement of S3 is greater than S2.

Notice that S3's attack surface measurement would have been greater than its current measurement if all its interfaces were public; the presence of internal interfaces results in a minimal attack surface measurement. Hence the addition of the internal interfaces was a good design decision that reduced the attack surface measurement and hence mitigated the security risk of the service.

6.6 Lessons Learned and Future Work

The collaboration with SAP was a valuable experience for us. We received positive feedback from the product development team on the utility of the attack surface measurement process. The team found the tool's detailed output to be useful in reducing the attack surface. They also found it useful to have two options for measuring the attack surface and the ability to perform incremental measurements.

We also learned important lessons on how to improve our measurement method and our measurement tool to make the measurement process more useful in practice. For example, our tool's output guides the developers to focus on the relevant parts of a system to reduce the attack surface; the output, however, does not help in deciding when to stop the reduction process. A possible direction of future work to address this issue is to develop a method to estimate the minimum and maximum possible attack surface measurement of a system given the system's functionality. We discuss a possible method of estimating the minimum and maximum measurements in Chapter 8.

Software developers and architects can use the minimum and maximum estimation method for prioritizing software testing effort. For example, if a system's attack surface measurement is closer to the maximum, then they should invest more in testing efforts; if the measurement is closer to the minimum, they can reduce their testing effort. We discuss the use of the attack surface measurement in prioritizing software testing effort in Chapter 8.

We also identified further improvements to our tool based on the feedback received and the lessons learned from this collaboration. There are four possible extensions of our tool to make it more useful for the software developers: (1) extension of the tool to identify a system's indirect entry points and indirect exit points, (2) presentation of the measurement results in a graphical window inside the Eclipse IDE, (3) implementation of a Graphical User Interface (GUI) to update the configuration information, and (4) extension of the tool to measure the attack surfaces of systems implemented in `JavaScript` and `ABAP`.

Chapter 7

Related Work

In this chapter, we compare our work with prior work on attack surface measurement in Section 7.1 and with previous work on quantitative assessment of software security in Section 7.2. We also discuss prior work on network security measurements and metrics in Section 7.3.

7.1 Attack Surface Measurement

Michael Howard of Microsoft informally introduced the notion of attack surface for the Windows operating system [45]. Pincus and Wing further elaborated Howard's Relative Attack Surface Quotient (RASQ) measurement method [44]. Their attack surface measurement method, however, was based on intuition and was ad-hoc in nature. In this thesis, we formalize the notion of a system's attack surface and propose a systematic method for measuring a system's attack surface. Please see Chapter 2 for a detailed discussion of previous work on attack surface measurement.

7.2 Other Software Security Measurements and Metrics

Our attack surface metric differs from prior work in three key aspects. First, our attack surface measurement is based on a system's inherent properties and is independent of any vulnerabilities present in the system. Previous work assumes the knowledge of the known vulnerabilities present in the system [8, 98, 79, 90, 64, 53]. In contrast, our identification of all entry points and exit points encompasses all known vulnerabilities as well as potential vulnerabilities not yet discovered or exploited. Moreover, a system's attack surface measurement indicates the security risk of the exploitation of the system's vulnerabilities; hence our metric is complementary to previous work and can be used in conjunction with previous work.

Second, prior research on measurement of security has taken an *attacker-centric approach* [79, 90, 64, 53]. In contrast, we take a *system-centric approach*. The attacker-centric approach makes assumptions about attacker capabilities and resources whereas the system-centric approach assesses a system's security without reference to or assumptions about attacker capabilities [75]. Our attack surface measurement is based on a system's design and is independent of the attacker's capabilities and behavior; hence our metric can be used as a tool in the software design and development process.

Third, many of the prior works on quantification of security are conceptual in nature and haven't been applied to real software systems [8, 55, 53, 59, 88]. In contrast, we demonstrate the applicability of our metric to real systems by measuring the attack surfaces of two FTP servers, two IMAP servers, and three versions of an SAP software system.

Alves-Foss et al. use the System Vulnerability Index (SVI) as a measure of a system's vulnerability to common intrusion methods [8]. A system's SVI is obtained by evaluating factors grouped into three problem areas: system characteristics, potentially neglectful acts, and potentially malevolent acts. Alves-Foss et al., however, identify only the relevant factors of operating systems; their focus is on operating systems and not individual or generic software applications. Moreover, they assume that they can quantify all the factors that determine a system's SVI. In contrast, we assume that we can quantify a resource's damage potential and effort.

Littlewood et al. explore the use of probabilistic methods used in traditional reliability analysis in assessing the operational security of a system [55]. In their conceptual framework, they propose to use the effort made by an attacker to breach a system as an appropriate measure of the system's security. They conjecture that effort in security plays the same role as time in reliability. They, however, do not propose a concrete method to estimate the attacker effort.

Voas et al. propose a relative security metric based on a fault injection technique [98]. They simulate different threat classes of a system by mutating program variables during the execution of the system and then observe the impact of the threat classes on the behavior of the executing system in terms of successful intrusions. They propose a Minimum-Time-To-Intrusion (MTTI) metric based on the predicted period of time before any simulated intrusion can take place. The higher the MTTI value, the more secure the system. The MTTI value, however, depends on the threat classes simulated and the intrusion classes observed. In contrast, the attack surface metric does not depend on any threat class. Moreover, the MTTI computation assumes the knowledge of system vulnerabilities.

Ortalo et al. model a system's known vulnerabilities as a privilege graph [24] and combine assumptions about the attacker's behavior with the privilege graphs to obtain attack state graphs [79]. They 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. Their technique, however, assumes the knowledge of the vulnerabilities present in the system and the attacker's behavior. Moreover, their approach focuses on assessing the operational security of operating systems and not individual software applications.

Schneier uses attack trees to model the different ways in which a system can be attacked [90]. Given an attacker goal, Schneier constructs an attack tree to identify the different ways in which the goal can be satisfied and to determine the cost to the attacker in satisfying the goal. The estimated cost is a measure of the system's security. Construction of an attack tree, however, assumes the knowledge of the following three factors: system vulnerabilities, possible attacker goals, and the attacker behavior.

McQueen et al. use an estimate of a system's expected time-to-compromise as an

indicator of the system's security risk [64, 65]. The time-to-compromise (TTC) is the time needed by an attacker to gain a privilege level in a system; TTC depends on the nature of the vulnerabilities present in the system and the attacker's skill level. They model the process of compromising a system as a random process and estimate the expected time to compromise the system. They, however, assume the knowledge of the vulnerabilities present in a system and the capabilities of differing attacker skill levels.

Leversage and Byers use a state space model to estimate the mean time-to-compromise (MTTC) a system [53]. Their model requires the estimation of the attack paths and the state times to calculate the MTTC. They use a modified version of McQueen et al.'s algorithm to estimate the state times. Hence the state space model assumes the knowledge of the vulnerabilities present in a system and the differing attacker skill levels. The model also focuses on attacks launched remotely from the Internet.

Madan et al. use stochastic modeling techniques to quantify the security attributes of intrusion tolerant systems [59, 60]. They model attacker behavior and a system's response to intrusions as semi-Markov processes (SMP) and then quantify measures such as a system's steady-state availability and the mean time to security failure. They, however, assume the knowledge of the SMP model's parameters. Moreover, their work focuses on intrusion tolerant systems such as the SITAR system [99] and is useful studying different intrusion tolerance strategies.

Schechter introduces an economic approach to measure the *security strength* of software in units of dollars [88]. A system's security strength is the cost of breaching the system's security. He proposes to use the market price of a new vulnerability in a system as the system's security strength. Schechter's approach uses market means to quantify a system's security strength. In contrast, our attack surface metric uses a system's inherent attributes to quantify the system's security.

7.3 Network Security Measurements and Metrics

In this thesis, we use the notion of a system's attack surface as a software security metric. A recent line of work introduces four conceptual network security metrics to measure a network's security [80, 101, 102, 100]. Unlike the attack surface metric, the four network security metrics haven't been applied to any real systems. A network's security is dependent on its configuration in all the four metrics. Similarly, a system's attack surface measurement is dependent on the system's configuration. The metrics use a network's attack graph to estimate the network's security. The construction of an attack graph assumes the knowledge of the vulnerabilities present in the network's hosts. Also, the use of attack graphs is suitable for analyzing the security of networks and not of software. Hence the proposed metrics are not suitable for measuring software security. We, however, believe that our attack surface metric can be used to measure network security because of the composition property of the I/O automata; we can model a network as a composition of the I/O automata modeling the hosts in the network and hence measure the network's attack surface in a given configuration. Moreover, our metric does not assume the knowledge of the vulnerabilities present in the hosts; hence a network's attack surface measurement is complementary to the four proposed metrics.

Pamula et al. introduce the *weakest-adversary metric* to quantify the security of a network in a given configuration in terms of the strength of the weakest adversary that can penetrate the network [80]. Given a network configuration, a set of known exploits, an attacker goal, and an attacker class, they construct an attack graph and identify the minimal set of attributes needed by the adversary to reach the goal. They define a network configuration, C_1, to be less secure than a configuration, C_2, if C_1 is vulnerable to a set, S_1, of adversary attributes, C_2 is vulnerable to a set, S_2, of adversary attributes, and $S_1 \subset S_2$.

Wang et al. introduce an *attack resistance* metric to measure and compare the security of different network configurations [101]. Given a network configuration and a set of known exploits, they assume the knowledge of the individual *resistance* of each exploit. An exploit's resistance reflects the time and effort required to carry out the exploit. They construct an attack graph of the given network configuration and compute the resistance

of the configuration from the attack graph using the individual resistance of the exploits. They also show that Pamula et al.'s metric is a special case of their attack resistance metric under ceratin conditions. Wang et al. later generalize the attack resistance metric to a framework for measuring other aspects of network security such as the potential damage caused by attacks and the cost of reconfiguring a network using various possible network hardening operations [102].

Wang et al. also introduce an attack graph based probabilistic metric to measure the security of network configurations [100]. In their attack graph, they model a system's states as security *conditions* and the *exploit* of a vulnerability as a transition between system states. They assume the knowledge of the probabilities of the exploits being executed and the conditions being satisfied. Given an attacker goal and a network configuration, they compute the probability that an attacker will reach the goal from the probabilities of the exploits and conditions; they propose to use the probability as an indicator of the network configuration's security.

Chapter 8

Summary and Future Work

In this chapter, we summarize the contributions of this thesis and discuss possible avenues of future work.

8.1 Summary of Contributions

Our research on attack surface measurement is inspired by Michael Howard's informal RASQ measurements. We applied Howard's method to four different versions of Linux to identify the shortcomings in his method. We addressed the shortcomings in this thesis and proposed a systematic attack surface measurement method.

The key contribution of this thesis is the formal definition of a system's attack surface. We introduced an I/O automata model of a system and its environment to formalize the notion of the attack surface. We introduced the entry point and exit point framework to identify the resources that are part of the attack surface. We also introduced the notions of damage potential and effort to estimate a resource's contribution to the attack surface. We defined an abstract method to measure a system's attack surface and showed that the method is analogous to risk modeling. In our formal model, we showed that a larger attack surface leads to a larger number of potential attacks on a system.

We then introduced a concrete method to measure the attack surfaces of systems implemented in `C`. We applied our method to two popular open source IMAP servers and two FTP daemons. We demonstrated the use of our attack surface metric in the decision making process by comparing the attack surfaces of the IMAP servers and those of the FTP daemons. We also performed a parameter sensitivity analysis to provide guidelines to the users of our metric.

A key challenge in our research was the validation of the attack surface metric. We conducted three empirical studies for validation; these studies are based on the general approach to validating software metrics. We validated the steps in our measurement method by conducting an expert user survey and by statistically analyzing the data collected from Microsoft security bulletins. We validated attack surface measurements by demonstrating a correlation between attack surface measurements and patches of security vulnerabilities in software.

We demonstrated the applicability of our measurement method to enterprise-scale software by measuring the attack surfaces of SAP business applications. We introduced a method to measures the attack surfaces of business applications implemented in `Java` and implemented a tool to automatically measure the attack surface from a business application's source code. We demonstrated the utility of our metric in the software development process by collaborating with a Product Development Group of SAP.

8.2 Future Work

In this section, we discuss several possible directions of future work.

8.2.1 Usage of the Entry Point and Exit Point Framework

A possible direction of future work is to explore the use of the entry point and exit point framework in the threat modeling process [94]. 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, system designers can take appropriate countermeasures to mitigate the system's security 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 can use the entry point and exit point framework to formalize this step in the threat modeling process.

8.2.2 Extensions of the Formal Model

We do not make any assumptions about an attacker's resources, capabilities, and behavior in our I/O automata model. In terms of an attacker profile used in cryptography, we do not characterize an attacker's power and privilege. A useful extension of our work would be to include an attacker's power and privilege in our formal I/O automata model.

Our I/O automata model is not expressive enough to include attacks such as side channel attacks, covert channel attacks, and attacks where one user of a software system can affect other users (e.g., fork bombs). We could extend the current formal model by extending our formalization of damage potential and attacker effort to include such attacks.

8.2.3 Extensions of the Measurement Method

Our attack surface measurement method requires a system's source code in order to identify the system's entry points and exit. It may not, however, be always feasible to obtain the source code of a system (e.g., commercial software). Moreover, the size of the codebase may be prohibitively large (e.g., the codebase of a Linux distribution). A useful extension of the measurement method would be to define a systematic way to approximate a system's attack surface measurement in the absence of the source code. In the absence of source code, we can define a system's entry points and exit points in terms of the system's *components*. A component in the context of the attack surface measurement can be thought of as a unit of executable code that can be added to or removed from the system. Few examples of components are libraries, executable files, installed daemons, and web server modules.

We can extend the current method by proposing a systematic way to identify a system's components and to estimate the damage potential-effort ratio of each such component.

A key challenge in our attack surface measurement method is the estimation of the damage potential-effort ratio. Recall that in order to estimate the damage potential-effort ratio of a resource, we have to impose total orderings among the values of the attributes of the resource. For example, we impose total orderings among the privilege levels and the access rights levels of the methods, the protocols and the access rights levels of the channels, and the types and the access rights levels of the data items. Natural total orderings exist for four out of these six attributes; it is easy to impose total orderings among the privilege levels of the methods, the access rights levels of the methods, the access rights levels of the channels, and the access rights levels of the data items. For example, the total ordering among the access rights levels of the methods is `root` > `authenticated` > `unauthenticated`. No such natural orderings, however, exist for the protocols of the channels and the data types of the data items. We need domain knowledge to decide whether the damage potential of `TCP` is greater than that of `UDP` or whether the damage potential of `cookies` is greater than that of `database records`. A useful extension of the measurement method would be to provide a set of guidelines to the users so that they can impose appropriate total orderings among the values of these two attributes using their domain knowledge. For example, we can guide the user to identify attributes of protocols such as the data exchange format and the number of steps in connection establishment that will help the user impose a total ordering among the protocols.

Currently there are no source code analysis tools that enable us to identify a system's indirect entry points (exit points). Hence in both our `C` and `Java` measurements, we could not identify the indirect entry points (exit points) in an automated manner. A possible direction of future work would be to explore code analysis techniques to identify a system's indirect entry points and indirect exit points.

8.2.4 Validation Approaches

In this thesis, we conducted three empirical studies for validating the attack surface metric. These studies were exploratory in nature. Another possible validation approach would be to correlate a system's attack surface measurement with the number of observed attacks on the system. Honeypots are dedicated hosts on the Internet that can simulate vulnerable software systems to an attacker. The honeypots act as devices that monitor attack activities; hence we can analyze honeypot data to estimate the number of real attacks on a software system. If a system, A, has a larger attack surface compared to a system, B, then we would expect to see a larger number of attacks on A compared to B.

Another validation approach would be to integrate the attack surface measurement and reduction process with the software development process and then observe the benefits of attack surface reduction after the release of software. If the newer version of a system undergoes attack surface reduction, then we would expect to see fewer exploitable vulnerabilities in the newer version than the older versions over similar time periods.

Another validation approach would be to use Cadar et al.'s approach of automatically generating inputs that crash a software system [13]. Cadar et al. use symbolic execution to generate the relevant inputs and to identify the code locations where crashes occur. A system's method that contains a crash location receives data from the system's environment; hence in terms of our I/O model, the method is an entry point of the system. Intuitively, if we identify a set, E, of a system's entry points and also use Cadar et al.'s approach to identify a set, C, of methods that contain crash locations, then we would expect to see that C is a subset of E.

8.2.5 Improvements of the Measurement Tool

We implemented a tool as an Eclipse plug-in to measure the attack surfaces of SAP business applications. Based on the feedback received from the SAP product development group, we have identified three possible extensions of the tool to make it more useful for software developers. First, the tool currently outputs its result in the form of a text file. We

could improve the tool by presenting the results in a graphical window inside the Eclipse IDE so that the developers can access the measurement results within the IDE. Second, we could improve the usability of the tool by implementing a Graphical User Interface (GUI) to update the configuration information required by the tool. Third, the tool currently measures the attack surfaces of systems implemented in `Java`. The tool would be more useful in practice if we were to extend the tool to measure the attack surfaces of software implemented in other languages such as `JavaScript` and `ABAP` [1].

8.2.6 Attack Surface Range Analysis

The result of our attack surface measurement method guides the software developers to focus on a system's relevant parts to reduce the system's attack surface. For example, the developers can analyze the top contributing entry points and exit points instead of the entire code base to reduce the attack surface. The result, however, does not help in deciding when to stop the reduction process. In order to address this issue, a possible extension of our work is to develop a method to estimate the minimum and the maximum possible attack surface measurements of a system given the system's functionality. We briefly describe such a method in the following paragraph.

In order to estimate the minimum and the maximum attack surface measurements, we need to estimate an entry point's (exit point) minimum and maximum contributions to the attack surface, i.e., we need to estimate the minimum and the maximum damage potential-effort ratios. We can estimate the minimum and the maximum damage potential-effort ratios from the range of numeric values assigned to damage potential and effort. We also need to estimate the appropriate number of entry points and exit points required to implement the system's functionality. In the absence of such an estimate, we can simplify our analysis by assuming that the appropriate number is the same as the observed number of entry points and exit points. Hence we can estimate the minimum and the maximum attack surface measurements by multiplying the number of entry points and exit points with the minimum and the maximum damage potential-effort ratios, respectively.

8.2.7 Usage of Attack Surface Measurements

A system's attack surface measurement can be used in other contexts besides the obvious, i.e., as an indication of the system's security risk. We describe four possibilities here; a useful direction of future work would be to explore other usage of the attack surface measurement.

First, software developers and architects can use the minimum and the maximum attack surface measurement estimates to prioritize software testing effort. For example, if a system's attack surface measurement is closer to the maximum, then they should invest more in testing efforts; if the measurement is closer to the minimum, they can reduce their testing effort.

Second, software developers can use attack surface measurements as a guide while implementing patches of security vulnerabilities in their software systems. A good patch should not only remove a vulnerability from a system, but also should not increase the system's attack surface. Software developers can use our measurement method to ensure that their patches do not increase the attack surface.

Third, we can use attack surface measurements to provide guidelines for "safe" software composition. We consider a composition of two systems, 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. A possible direction of future work would be to identify the conditions under which the composition of two given systems is safe.

Fourth, software consumers often have to make a choice between several possible configurations of software. For example, SAP business applications can be configured in many different ways; SAP customers choose the configuration best for them. Configuring large enterprise-scale software is a complex process; hence choosing an appropriate configuration is a non-trivial and error-prone task. We could use a system's attack surface measurement as a guide in choosing an appropriate configuration. Since a system's attack surface measurement is dependent on the system's configuration, we would choose a configuration that results in a smaller attack surface exposure.

8.3 Final Word

The research community has acknowledged that security metrics and measurements is a very challenging area in security research [37, 21, 11, 63]. There is, however, a pressing need for practical security metrics and measurements today. This thesis research is a first step in the grander challenge of security metrics. We have formalized the pragmatic approach of attack surface measurements; our metric is useful to both software industry and software consumers.

Howard's measurement method is already used in a regular basis as part of Microsoft's Security Development Lifecycle [43]. Mu Security's Mu-4000 Security Analyzer uses our attack surface measurement framework for security analysis [91]. SAP is also planning to implement a tool based on the prototype discussed in Chapter 6 to be used as part of their software quality improvement process.

We, however, believe that no single security metric or measurement will be able to fulfill our requirements. We certainly need multiple metrics and measurements to quantify different aspects of security. We hope that our work will rekindle interest in security metric research. We also believe that our understanding over time would lead us to more meaningful and useful quantitative security metrics.

Appendix A

Input and *Output* Methods

In this section, we below list the *Input* and *Output* sets of library methods required by our C measurement method.

$Input$ = {canonicalize_file_name, catgets, confstr, ctermid, ctermid, cuserid, dgettext, dngettext, fgetc, fgetc_unlocked, fgets, fgets_unlocked, fpathconf, fread, fread_unlocked, fscanf, getc, getchar, getchar_unlocked, getc_unlocked, get_current_dir_name, getcwd, getdelim, getdelim, __getdelim, getdents, getenv, gethostbyaddr, gethostbyname, gethostbyname2, gethostent, gethostid, getline, getline, getlogin, getlogin_r, getmsg, getopt, _getopt_internal, getopt_long, getopt_long_only, getpass, getpmsg, gets, gettext, getw, getwd, ngettext, pathconf, pread, pread64, ptsname, ptsname_r, read, readdir, readlink, readv, realpath, recv, recv_from, recvmesg, scanf, __secure_getenv, signal, sysconf, ttyname, ttyname_r, vfscanf, vscanf}

$Output$ = {dprintf, fprintf, fputc, fputchar_unlocked, fputc_unlocked, fputs, fputs_unlocked, fwrite, fwrite_unlocked, perror, printf, psignal, putc, putchar, putc_unlocked, putenv, putmsg, putpmsg, puts, putw, pwrite, pwrite64, send, sendmsg, sendto, setenv, sethostid, setlogin, ungetc, vdprintf, vfprintf, vsyslog, write, writev}

Appendix B

Survey Questionnaire

In this section, we reproduce the survey questionnaire sent to the survey participants.

Attack Surface Measurement Survey

The attack surface measurement project at Carnegie Mellon University aims to find a systematic method to measure a software product's _attack surface_. Intuitively, a software product's attack surface is the set of ways in which the product can be attacked. We propose to use a software product's attack surface measurement as an indicator of the product's security. Your expertise as a Linux system administrator will help us in defining a systematic attack surface measurement method.

This survey contains 6 questions and takes less than 30 minutes to fill out. If you have any questions, please email Pratyusa Manadhata (pratyus@cs.cmu.edu).

For each question, unless otherwise stated, please indicate your choice by putting an X between the parentheses to the left of your choice.

1. Software products running on Linux are attacked using the product's _resources_. Examples of software products are web servers, database servers, and IMAP servers. The resources of a product can be broadly categorized into three types: methods in the source code of the product,

communication channels opened by the product, and data accessed by the product. The following are few examples of the three types of resources.

-method: the APIs/interfaces of daemons, libraries, and reusable components
-channel: sockets and pipes
-data: configuration files, scripts, and database records

An attacker might attack a product using a combination of the product's resources. For example, in a buffer overrun attack, an attacker connects to the product using a communication channel and invokes the method of the product that has the buffer overrun vulnerability.

Consider a web application server running on Linux that serves both static and dynamic content. Please rate your degree of agreement or disagreement with the following statements.

(a) The web server's methods (possibly in combination with other resources) that take input from the server's clients can be used in attacks on the web server.

( ) Strongly agree
( ) Agree
( ) Neither agree or disagree
( ) Disagree
( ) Strongly disagree

(b) The communication channels (possibly in combination with other resources) opened by the web server can be used in attacks on the web server.

( ) Strongly agree
( ) Agree
( ) Neither agree or disagree
( ) Disagree
( ) Strongly disagree

(c) The data (possibly in combination with other resources) accessed by the web server can be used in attacks on the web server.

( ) Strongly agree
( ) Agree
( ) Neither agree or disagree
( ) Disagree
( ) Strongly disagree

(d) Please explain the reasons behind each of your answers in 1-2 sentences in the space below. What other resources of the web server are used to attack the web server?

2. We define a resource's _damage potential-effort ratio_ as the ratio of the resource's damage potential to attacker effort where

damage potential is the damage (technical impact (e.g., root elevation) and not business impact (e.g., monetary loss)) the attacker can cause to the product in using the resource in an attack, and

effort is the effort the attacker spends to use the resource in an attack.

The higher the damage potential or the lower the effort, the higher the damage potential-effort ratio.

(a) Please rate your degree of agreement or disagreement with the following statement.

The damage potential-effort ratio of a resource is an indicator of how likely the resource is going to be used in an attack.

( ) Strongly agree
( ) Agree
( ) Neither agree or disagree
( ) Disagree
( ) Strongly disagree

(b) Please explain the reasons behind your answer in 1-2 sentences in the space below. List any other indicators of how likely the resource is going to be used in an attack.

In the following three questions (Questions 3-5), the definitions of damage potential and effort are as given in Question 2. We estimate a resource's damage potential-effort ratio in terms of the resource's properties and not the attacker's capabilities. We do not assume a specific attacker profile and consider the entire range of profiles from script kiddies to organized crime.

3. Consider the methods in the source code of the web server.

The _privilege_ of the method is the Linux privilege (either root or non-root user) with which the method runs. If the attacker exploits a vulnerability present in the method, the attacker gains the privilege of the method.

Please rate your degree of agreement or disagreement with the following statement.

(a) The privilege of the method is an indicator of the method's damage potential.

( ) Strongly agree
( ) Agree
( ) Neither agree or disagree
( ) Disagree
( ) Strongly disagree

(b) Please explain the reasons behind your answer in 1-2 sentences in the space below. What other properties of the method might be indicators of damage potential?

The _access rights_ of the method determines who can access the method (e.g., unauthenticated users, authenticated users, admin user, and root). For example, anyone can access the methods of the httpd daemon whereas only admin users can access the methods in the interface of the administrative module of a web server. The attacker spends effort to acquire necessary access rights in order to be able to use a method in an attack.

(c) Please rate your degree of agreement or disagreement with the following statement.

The access rights of the method is an indicator of the attacker effort.

( ) Strongly agree
( ) Agree
( ) Neither agree or disagree
( ) Disagree
( ) Strongly disagree

(d) Please explain the reasons behind your answer in 1-2 sentences in the space below. What other properties of the method might be indicators of attacker effort?

4. Consider the communication channels opened by the web server.

The _protocol_ (e.g., TCP, UDP, and RPC) of a channel defines the rules of communication and the format of the data exchanged over the channel. For example, TCP allows raw streams of data whereas RPC allows only well formatted data to be exchanged. Hence the attacker can send arbitrary data using a TCP socket whereas the attacker can send only well formatted data using an RPC end point.

(a) Please rate your degree of agreement or disagreement with the following statement.

The protocol of the channel is an indicator of the channel's damage potential.

( ) Strongly agree
( ) Agree
( ) Neither agree or disagree
( ) Disagree
( ) Strongly disagree

(b) Please explain the reasons behind your answer in 1-2 sentences in the space below. What other properties of the channel might be indicators of damage potential?

The _access rights_ of a channel determines who can access the channel (e.g., accessible by users on the same computer, accessible by users on the same subnet, and accessible by users on the Internet). For example, TCP sockets are accessible by users on the Internet whereas UNIX pipes are accessible by users on the same computer.

(c) Please rate your degree of agreement or disagreement with the following statement.

The access rights of the channel is an indicator of the attacker effort.

( ) Strongly agree
( ) Agree
( ) Neither agree or disagree
( ) Disagree
( ) Strongly disagree

(d) Please explain the reasons behind your answer in 1-2 sentences in the space below. What other properties of the channel might be indicators of attacker effort?

5. Consider the data accessed by the web server.

The _type_ of the data (e.g., cookies, database records, and files) determines the format of the data. For example, files can contain arbitrary data whereas cookies have well defined format. The attacker can indirectly send arbitrary data to the web server using a file whereas the attacker can send only well formatted data using a cookie. For example, if the attacker can write to a file that is later read by the web server, the attacker can indirectly send arbitrary data to the web server.

(a) Please rate your degree of agreement or disagreement with the following statement.

The type of the data is an indicator of the data's damage potential.

( ) Strongly agree
( ) Agree
( ) Neither agree or disagree
( ) Disagree
( ) Strongly disagree

(b) Please explain the reasons behind your answer in 1-2 sentences in the space below. What other properties of the data might be indicators of damage potential?

The _access rights_ of the data determines who can read and write the data (e.g., UNIX rwxrwxrwx access rights determine the access rights for owner, group, and world).

(c) Please rate your degree of agreement or disagreement with the following statement.

The access rights of the data is an indicator of the attacker effort.

( ) Strongly agree
( ) Agree
( ) Neither agree or disagree
( ) Disagree

( ) Strongly disagree

(d) Please explain the reasons behind your answer in 1-2 sentences in the space below. What other properties of the data might be indicators of attacker effort?

6. Background Information

(a) What is your affiliation?

( ) University

( ) Corporate

( ) Government

( ) Other: _____

(b) How many years of full-time Linux administration experience do you have?

( ) Less than 2

( ) 2-5

( ) 6-10

( ) More than 10

(c) Please indicate which Linux software you have installed or maintained.

( ) Web server

( ) IMAP server

( ) Database server

( ) Others: _____

(d) Does your job involve making software purchase decisions? (e.g., deciding which web server product to buy).

( ) Yes
( ) No

(e) If so, please rank the factors that guide your decision.
(1 = most important, 5 = least important)

( ) Ease of administration
( ) Ease of installation
( ) Feature set
( ) Open source
( ) Security

(f) How familiar are you with the following types of software attacks?
(
1=I haven't heard about the attack,
2=I have heard about the attack but not really understand what it is
3=I have heard about the attack and understand the steps involved in it
4=I know how to implement the attack
)

Buffer Overrun ( ) 1, ( ) 2, ( ) 3, ( ) 4
Format String ( ) 1, ( ) 2, ( ) 3, ( ) 4
Cross Site Scripting ( ) 1, ( ) 2, ( ) 3, ( ) 4
DoS/DDoS ( ) 1, ( ) 2, ( ) 3, ( ) 4

Thank you for filling out our survey!

Appendix C

Common Weakness Enumeration (CWE) Definitions

In this section, we list the 23 CWE definitions used by NVD bulletins. We obtained the definitions from NIST's website [76].

Name	CWE-ID	Description
Authentication Issues	CWE-287	Failure to properly authenticate users.
Credentials Management	CWE-255	Failure to properly create, store, transmit, or protect passwords and other credentials.
Permissions, Privileges, and Access Control	CWE-264	Failure to enforce permissions or other access restrictions for resources, or a privilege management problem.

Name	CWE-ID	Description
Buffer Errors	CWE-119	Buffer overflows and other buffer boundary errors in which a program attempts to put more data in a buffer than the buffer can hold, or when a program attempts to put data in a memory area outside of the boundaries of the buffer.
Cross-Site Request Forgery (CSRF)	CWE-352	Failure to verify that the sender of a web request actually intended to do so. CSRF attacks can be launched by sending a formatted request to a victim, then tricking the victim into loading the request (often automatically), which makes it appear that the request came from the victim. CSRF is often associated with XSS, but it is a distinct issue.
Cross-Site Scripting (XSS)	CWE-79	Failure of a site to validate, filter, or encode user input before returning it to another users web client.
Cryptographic Issues	CWE-310	An insecure algorithm or the inappropriate use of one; an incorrect implementation of an algorithm that reduces security; the lack of encryption (plaintext); also, weak key or certificate management, key disclosure, random number generator problems.
Path Traversal	CWE-22	When user-supplied input can contain .. or similar characters that are passed through to file access APIs, causing access to files outside of an intended subdirectory.
Code Injection	CWE-94	Causing a system to read an attacker-controlled file and execute arbitrary code within that file. Includes PHP remote file inclusion, uploading of files with executable extensions, insertion of code into executable files, and others.
Format String Vulnerability	CWE-134	The use of attacker-controlled input as the format string parameter in certain functions.
Configuration	CWE-16	A general configuration problem that is not associated with passwords or permissions.
Information Leak / Disclosure	CWE-200	Exposure of system information, sensitive or private information, fingerprinting, etc.

Name	CWE-ID	Description
Input Validation	CWE-20	Failure to ensure that input contains well-formed, valid data that conforms to the applications specifications. Note: this overlaps other categories like XSS, Numeric Errors, and SQL Injection.
Numeric Errors	CWE-189	Integer overflow, signedness, truncation, underflow, and other errors that can occur when handling numbers.
OS Command Injections	CWE-78	Allowing user-controlled input to be injected into command lines that are created to invoke other programs, using system() or similar functions.
Race Conditions	CWE-362	The state of a resource can change between the time the resource is checked to when it is accessed.
Resource Management Errors	CWE-399	The software allows attackers to consume excess resources, such as memory exhaustion from memory leaks, CPU consumption from infinite loops, disk space consumption, etc.
SQL Injection	CWE-89	When user input can be embedded into SQL statements without proper filtering or quoting, leading to modification of query logic or execution of SQL commands.
Link Following	CWE-59	Failure to protect against the use of symbolic or hard links that can point to files that are not intended to be accessed by the application.
Other	No Mapping	NVD is only using a subset of CWE for mapping instead of the entire CWE, and the weakness type is not covered by that subset.
Not in CWE	No Mapping	The weakness type is not covered in the version of CWE that was used for mapping.
Insufficient Information	No Mapping	There is insufficient information about the issue to classify it; details are unkown or unspecified.
Design Error	No Mapping	A vulnerability is characterized as a Design error if there exists no errors in the implementation or configuration of a system, but the initial design causes a vulnerability to exist.

Bibliography

[1] SAP AG. Abap development. https://www.sdn.sap.com/irj/sdn/abap. 8.2.5

[2] SAP AG. SAP - business software solutions applications and services. http://www.sap.com. 3

[3] SAP AG. SAP - business software solutions applications and services. http://www.sap.com/index.epx. 6.1

[4] SAP AG. SAP - enterprise solutions technology leader. http://www.sap.com/about/index.epx. 6.1

[5] SAP AG. SAP NetWeaver. http://www.sap.com/platform/netweaver/index.epx. 6.2

[6] SAP AG. SAP NetWeaver products. https://www.sdn.sap.com/irj/sdn/nw-products. 6.2

[7] SAP AG. Software lifecycle security. https://www.sdn.sap.com/irj/sdn/security?rid=/webcontent/uuid/e0422d71-b23e-2a10-35bd-862787ccae26. 6.1

[8] J. Alves-Foss and S. Barbosa. Assessing computer security vulnerability. *ACM SIGOPS Operating Systems Review*, 29(3):3–13, 1995. 7.2

[9] E. Asbeck and Y. Y. Haimes. The partitioned multiobjective risk method. *Large Scale Systems*, 6(1):13–38, 1984. 3.4.2

[10] David F. Bacon and Peter F. Sweeney. Fast static analysis of c++ virtual function calls. In *OOPSLA '96: Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 324–341, New York, NY, USA, 1996. ACM. 6.4.1

[11] S. M. Bellovin. On the brittleness of software and the infeasibility of security metrics. *IEEE Security and Privacy*, 04(4):96, 2006. 8.3

[12] Lionel Briand, Khaled El Emam, and Sandro Morasca. Theoretical and empirical validation of software product measures. Technical Report ISERN-95-03, Fraunhofer Institute for Experimental Software Engineering, 1995. 5.1

[13] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. Exe: automatically generating inputs of death. In *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security*, pages 322–335, New York, NY, USA, 2006. ACM. 8.2.4

[14] D.T. Campbell and D. W. Fiske. Convergent and discriminant validation by the multitrait-multimethod matrix. *Psychological Bulletin*, 2(56):81–105, 1959. 5.1.1

[15] CERT. CERT advisories. `http://www.cert.org/`. 2.3

[16] Prodromos D. Chatzoglou and Linda A. Macaulay. A rule-based approach to developing software development prediction models. *Automated Software Engg.*, 5(2):211–243, 1998. 5.3

[17] Hao Chen, David Wagner, and Drew Dean. Setuid demystified. In *Proceedings of the 11th USENIX Security Symposium*, pages 171–190, Berkeley, CA, USA, 2002. USENIX Association. 4.2.1

[18] Steven M. Christey. Personal communication, 2007. 5.4.2

[19] Microsoft Corporation. Microsoft security bulletin search. `http://www.microsoft.com/technet/security/current.aspx`. 5.2

[20] Microsoft Corporation. Microsoft security response center security bulletin severity rating system. `http://www.microsoft.com/technet/security/bulletin/rating.mspx`. 5.2

[21] Computing Research Association (CRA). Four grand challenges in trustworthy computing. `http://www.cra.org/reports/trustworthy.computing.pdf`, November 2003. 1, 8.3

[22] M. Crispin. Internet message access protocol version 4rev1. RFC 3501, March 2003. 4.3

[23] Project Cyrus. Cyrus IMAP server. `http://asg.web.cmu.edu/cyrus/imapd/`. 4.3

[24] M. Dacier and Y. Deswarte. Privilege graph: An extension to the typed access matrix model. In *Proc. of European Symposium on Research in Computer Security*, 1994. 5.6, 7.2

[25] Dan DaCosta, Christopher Dahn, Spiros Mancoridis, and Vassilis Prevelakis. Characterizing the security vulnerability likelihood of software functions. In *Proc. of International Conference on Software Maintenance*, 2003. 4.2.1

[26] Project Debian. Debian – the universal opearting system. `http://www.debian.org/`. 2.3.2

[27] Eclipse. Eclipse - an open development platform. `http://www.eclipse.org/`. 6.4

[28] Eclipse. Eclipse package org.eclipse.jdt.internal.corext.callhierarchy. `http://mobius.inria.fr/eclipse-doc/org/eclipse/jdt/internal/corext/callhierarchy/package-summary.html`. 6.4.1

[29] N. E. Fenton and M. Neil. A critique of software defect prediction models. *IEEE Transactions on Software Engineering*, 25(5), 1999. 3.4.2

[30] Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Co., Boston, MA, USA, 1998. 3.4, 5.1, 5.2.3, 5.6

[31] A. Fink and J. Kosecoff. *How to Conduct Surveys: A Step-By-Step Guide*. Sage Publications, Beverly Hills, CA, USA, 1985. 5.3

[32] Mozilla Foundation. Bugzilla@mozilla: Find a specific bug. `https://bugzilla.mozilla.org/query.cgi`. 5.4.2

[33] Mozilla Foundation. Firefox web browser — faster, more secure, and customizable. `http://www.mozilla.com/en-US/firefox/`. 5.4.2

[34] Mozilla Foundation. MFSA 2007-06: Mozilla network security services (NSS) SSLv2 buffer overflow. `http://www.mozilla.org/security/announce/2007/mfsa2007-06.html`. 5.5

[35] Mozilla Foundation. Mozilla foundation security advisories. `http://www.mozilla.org/projects/security/known-vulnerabilities.htm`. 5.4.2

[36] Virgil D. Gligor. Personal communication, 2008. 3.2.8

[37] Seymour E. Goodman and Herbert S. Lin, editors. *Toward a Safer and More Secure Cyberspace*. The National Academics Press, 2007. 1, 8.3

[38] R. Gopalakrishna, E. Spafford, , and J. Vitek. Vulnerability likelihood: A probabilistic approach to software assurance. Technical Report 2005-06, CERIAS, Purdue Univeristy, 2005. 3.4.2

[39] PRESTO Research Group. TACLE project. `http://presto.cse.ohio-state.edu/tacle/`. 6.4.1

[40] The WU-FTPD Development Group. Wu-ftpd. `http://www.wu-ftpd.org/`. 4.4

[41] Y. Y. Haimes. *Risk Modeling, Assessment, and Management*. Wiley, 2004. 3.4.2

[42] Curtis P. Haugtvedt, Paul M. Herr, and Frank R. Kardes, editors. *Handbook of Consumer Psychology*. Psychology Press, 2008. 5.1.1

[43] M. Howard and S. Lipner. *The Security Development Lifecycle*. Microsoft Press, 2006. 8.3

[44] 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. 2.1, 2.2, 7.1

[45] 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. 2.1, 7.1

[46] Michael Howard. Personal communication, 2005. 5.5

[47] Double Precision Inc. Courier-IMAP sever. `http://www.courier-mta.org/imap/`. 4.3

[48] Red Hat Inc. Redhat security advisories. `http://www.redhat.com/support/errata/index.html`. 2.3

[49] Red Hat Inc. redhat.com— downloads. `http://www.redhat.com/apps/downloads/`. 2.3.2

[50] Chris F Kemerer. An empirical validation of software cost estimation models. *Commun. ACM*, 30(5):416–429, 1987. 5.3

[51] Barbara Kitchenham, Shari Lawrence Pfleeger, and Norman Fenton. Towards a framework for software measurement validation. *IEEE Trans. Softw. Eng.*, 21(12):929–944, 1995. 5.1

[52] Barbara Kitchenham, Shari Lawrence Pfleeger, and Norman Fenton. Reply to: Comments on "towards a framework for software measurement validation". *IEEE Transactions on Software Engineering*, 23(3):189, 1997. 5.1

[53] David John Leversage and Eric James Byres. Estimating a system's mean time-to-compromise. *IEEE Security and Privacy*, 6(1):52–60, 2008. 7.2

[54] R. Likert. A technique for the measurement of attitudes. *Archives of Psychology*, 22(140):5–55, June 1932. 5.3.2

[55] B. Littlewood, S. Brocklehurst, N. Fenton, P. Mellor, S. Page, D. Wright, J. Dobson J. McDermid, and D. Gollman. Towards operational measures of computer security. *Journal of Computer Security*, 2(2/3):211–230, 1993. 7.2

[56] M. Y. Liu and I. Traore. Properties for security measures of software products. *Applied Mathematics and Information Science (AMIS) Journal*, 1(2):129–156, May 2007. 5.1.3, 5.6

[57] Michael Yanguo Liu. *Quantitative security analysis for service-oriented software architectures*. PhD thesis, University of Victoria, BC, Canada, 2008. 5.1.3, 5.6

[58] N. Lynch and M. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, September 1989. 3.1, 3.2.1

[59] Bharat B. Madan, Katerina Goseva-Popstojanova, Kalyanaraman Vaidyanathan, and Kishor S. Trivedi. Modeling and quantification of security attributes of software systems. In *DSN*, pages 505–514, 2002. 7.2

[60] Bharat B. Madan, Katerina Goseva-Popstojanova, Kalyanaraman Vaidyanathan, and Kishor S. Trivedi. A method for modeling and quantifying the security attributes of intrusion tolerant systems. *Perform. Eval.*, 56(1-4):167–186, 2004. 7.2

[61] Gervase Markham. Reducing attack surface. `http://weblogs.mozillazine.org/gerv/archives/2007/02/reducing_attack_surface.html`. 5.5

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

[63] J. McHugh. Quality of protection: Measuring the unmeasurable? Invited Talk at the ACM CCS Workshop on Quality of Protection, October 2006. 8.3

[64] Miles A. McQueen, Wayne F. Boyer, Mark A. Flynn, and George A. Beitel. Time-to-compromise model for cyber risk reduction estimation. In *ACM CCS Workshop on Quality of Protection*, September 2005. 7.2

[65] Miles A. McQueen, Wayne F. Boyer, Mark A. Flynn, and George A. Beitel. Quantitative cyber risk reduction estimation methodology for a small SCADA control system. In *HICSS*, 2006. 7.2

[66] Austin C. Melton, David M Gustafson, James M. Bieman, and Albert L. Baker. A mathematical perspective for software measures research. *Software Engineering Journal*, 5(5):246–254, 1990. 5.1

[67] Manoel G. Mendonça and Victor R. Basili. Validation of an approach for improving existing measurement frameworks. *IEEE Trans. Softw. Eng.*, 26(6):484–499, 2000. 5.3

[68] Microsoft. Microsoft security bulletin ms03-007. `http://www.microsoft.com/technet/security/bulletin/MS03-007.mspx`. 5.5

[69] Microsoft. Microsoft security bulletin ms04-011. `http://www.microsoft.com/technet/security/bulletin/MS04-011.mspx`. 5.5

[70] Microsoft. Microsoft security bulletin ms05-039. `http://www.microsoft.com/technet/security/bulletin/MS05-039.mspx`. 5.5

[71] MITRE. Common vulnerabilities and exposures. `http://cve.mitre.org/`. 2.3

[72] MITRE. CWE - common weakness enumeration. `http://cwe.mitre.org/`. 5.4.1

[73] Yukio Miyazaki and Kuniaki Mori. Cocomo evaluation and tailoring. In *ICSE '85: Proceedings of the 8th international conference on Software engineering*, pages 292–299, Los Alamitos, CA, USA, 1985. IEEE Computer Society Press. 5.3

[74] Sandro Morasca, Lionel C. Briand, Victor R. Basili, Elaine J. Weyuker, and Marvin V. Zelkowitz. Comments on "towards a framework for software measurement validation". *IEEE Transactions on Software Engineering*, 23(3):187–188, 1997. 5.1

[75] David M. Nicol. Modeling and simulation in security evaluation. *IEEE Security and Privacy*, 3(5):71–74, 2005. 7.2

[76] NIST. CWE - common weakness enumeration. `http://nvd.nist.gov/cwe.cfm`. 5.4.1, C

[77] NIST. National vulnerability database CVSS scoring. `http://nvd.nist.gov/cvss.cfm`. 5.4.2

[78] NIST. National vulnerability database home. `http://nvd.nist.gov/`. 5.4.1

[79] 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. 7.2

[80] Joseph Pamula, Sushil Jajodia, Paul Ammann, and Vipin Swarup. A weakest-adversary security metric for network configuration security analysis. In *QoP '06: Proceedings of the 2nd ACM workshop on Quality of protection*, pages 31–38, New York, NY, USA, 2006. ACM. 7.3

[81] Sergey Poznyakoff. GNU cflow. `http://www.gnu.org/software/cflow`. 4.2.1

[82] Debian Project. Debian linux security information. `http://www.debian.org/security/`. 2.3

[83] The ProFTPD Project. Bugzilla main page. `http://bugs.proftpd.org/`. 5.4.2

[84] The ProFTPD Project. The ProFTPD project home. `http://www.proftpd.org/`. 4.4, 5.4.2, 5.4.2

[85] Peter H. Rossi, James D. Wright, and Andy B. Anderson, editors. *Handbook of Survey Research*. The Academic Press, New York, NY, USA, 1983. 5.3, 5.3.2

[86] Andrea Saltelli, K. Chan, and E. M. Scott, editors. *Sensitivity Analysis*. John Wiley and Sons, 2000. 4.5

[87] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. In *Proceedings of the IEEE*, volume 63, pages 1278–1308, 1975. 5.6

[88] Stuart Edward Schechter. *Computer Security Strength & Risk: A Quantitative Approach*. PhD thesis, Harvard University, 2004. 7.2

[89] N.F. Schneidewind. Methodology for validating software metrics. *IEEE Transactions on Software Engineering*, 18(5):410–422, 1992. 5.1

[90] Bruce Schneier. Attack trees: Modeling security threats. *Dr. Dobb's Journal*, 1999. 7.2

[91] Mu Security. What is a security analyzer. `http://www.musecurity.com/solutions/overview/security.html`. 8.3

[92] 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. 5.3, 5.3.2

[93] Mariana Sharp, Jason Sawin, and Atanas Rountev. Building a whole-program type analysis in Eclipse. In *Eclipse Technology Exchange Workshop at OOPSLA*, pages 6–10, 2005. 6.4.1

[94] Frank Swiderski and Window Snyder. *Threat Modeling*. Microsoft Press, 2004. 8.2.1

[95] Jr. Thomas J. Bouchard. Unobtrusive measures: An inventory of uses. *Sociological Methods and Research*, 3(4):267–300, 1976. 5.1.1

[96] Carnegie Mellon Univeristy. Computing facilities - school of computer science - carnegie mellon. `http://www.cs.cmu.edu/~help/`. 2.3.2

[97] Rayford B. Vaughn, Ronda R. Henning, and Ambareen Siraj. Information assurance measures and metrics - state of practice and proposed taxonomy. In *Proc. of Hawaii International Conference on System Sciences*, 2003. 1

[98] 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. 7.2

[99] F. Wang and R. Uppalli. SITAR: A scalable intrusion-tolerant architecture for distributed services, 2003. 7.2

[100] Lingyu Wang, Tania Islam, Tao Long, Anoop Singhal, and Sushil Jajodia. An attack graph-based probabilistic security metric. In *22nd Annual IFIP WG 11.3 Working Conference on Data and Applications Security*, 2008. 7.3

[101] Lingyu Wang, Anoop Singhal, and Sushil Jajodia. Measuring the overall security of network configurations using attack graphs. In *DBSec*, pages 98–112, 2007. 7.3

[102] Lingyu Wang, Anoop Singhal, and Sushil Jajodia. Toward measuring network security using attack graphs. In *QoP '07: Proceedings of the 2007 ACM workshop on Quality of protection*, pages 49–54, New York, NY, USA, 2007. ACM. 7.3

[103] Information Week. Windows 2000 security represents a quantum leap. `http://www.informationweek.com/834/winsec.htm`. 2.2

[104] E.J. Weyuker. Evaluating software complexity measures. *IEEE Transactions on Software Engineering*, 14(9):1357–1365, 1988. 5.1

[105] Wikipedia. Sasser (computer worm). `http://en.wikipedia.org/wiki/Sasser_(computer_worm)`. 5.5

[106] Jeffrey M. Wooldridge. *Econometric Analysis of Cross Section and Panel Data*. The MIT Press, Cambridge, MA, USA, 2002. 5.2.3

[107] Horst Zuse. Support of experimentation by measurement theory. In *Proceedings of the International Workshop on Experimental Software Engineering Issues: Critical Assessment and Future Directions*, pages 137–140, London, UK, 1993. Springer-Verlag. 5.1