A Formal Approach To Practical Network Security Management

Sudhakar Govindavajhala

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
By the Department of
Computer Science

November 2006

Abstract

When a system administrator configures a network so it is secure, he understands very well the users, data, and most importantly the intent—what he is trying to do. However, he has a limited understanding of the mechanisms by which components interact and the details of each component. He could easily misconfigure the network so a hacker could steal confidential data. In addition to this complexity, about one hundred new security vulnerabilities are found each week, which makes it even more difficult to manage the security of a network installation—because of the large number of program vulnerabilities and challenging time constraints. Even professional administrators find this a difficult (impossible) task. How does one enable the system administrator to securely configure the network with a limited understanding of its components, program bugs and their interactions?

The solution is a security analysis framework that modularizes information flow between the system administrator, security expert and the bug expert. The administrator specifies what he is trying to do, the security expert specifies component behavior, the bug expert specifies known bugs. We developed a rule based framework—Multihost, Multi-stage, Vulnerability Analysis (MulVAL)—to perform end-to-end, automatic analysis of multi-host, multi-stage attacks on a large network where hosts run on different operating systems. The MulVAL framework has been demonstrated to be modular, flexible, scalable and efficient. We used the framework to find serious configuration vulnerabilities in software from several major vendors for the Windows XP platform.

Acknowledgments

This dissertation represents a significant milestone in my life. I would not have reached this point without the support, guidance, and encouragement of many people and organizations along the way. I would like to acknowledge a few of them here.

I owe a lot to the Princeton University community; thank you to everyone at Princeton for providing me with the opportunity, encouragement, and motivating environment.

Thanks to my advisor Andrew, for his guidance through out my stay at Princeton. He has been very patient, always available, encouraging, and inspiring. I admire Andrew for his clarity of thought, opinion on various things in life, insightful thoughts at crisis times and his ability to come up with incredible ideas. I would like to thank Ed Felten for the numerous insightful discussions during the last five years, especially in handling critical situations. I would like to thank Siva Raj Rajagopalan for his enthusiasm and insightful discussions. A good part of the work in this thesis is inspired by his work. I would like to thank Jennifer Rexford for her great enthusiasm, especially in helping me learn about related work. I would like to thank Brian Kernighan for the incisive feedback on an earlier draft of the thesis. (Incidentally, the first programming textbook I read was authored by Brian.) I would like to thank David Walker on insightful ideas on presentation of my ideas.

I would like to thank Larry Peterson, the current Chair of the department, for enthusiastically helping me resolve various administrative issues and identifying funding sources towards the end of the program. Thanks to Melissa Lawson, the graduate coordinator, for helping me negotiate the academic requirements and keeping me on track. Thanks to the technical "csstaff" staff—Jim Roberts, Scott Karlin, Joe Crouthamel, Steve Elgersma, Chris Tengi, Chris Miller, Paul Lawson, Brian Jones, and Chris Sanchez for keeping the systems running and for being patient with me while I bombarded them with

lots of support requests. I appreciate the time Joe Crouthamel and Chris Miller spent on teaching me aspects of system and network administration. I would like to thank Anthony Scaturro, Leila Shahbender, and Kevin Graham at Office of Information Technology for giving me valuable feedback about the vulnerability analysis tool. I would like to thank Donna O'Leary, Ginny Hogan, Michele Brown and Jennifer Widdis for helping me with administrative issues.

I would like to thank Prasad Rao for valuable discussions on using the XSB Prolog system. Thanks to the folks on various mailing lists such as linux-user (Linux discussion list in the department), unix-list (Princeton Unix Group), and csgrad (Graduate Students in the department list). The members of these discussions lists have been very patient with some of my less thoughtful questions and have been a valuable resource to get some tough technical questions answered.

I would like to thank the persons I shared the Security Lab (418A) and other offices for providing me with a pleasant place to work and accommodating situations like my overflowing desk. The list includes: David Penry, Limin Jia, Wu Qiang, Lujo Bauer, Spyros Triantafyllis, Ram Rangan, Jason Blome, Manish Vachharajani, Neil Vachharajani, Jason Lawrence, Yilei Shao, Seshadri Comandur, Brent Waters, and Jordan Boyd-Graber. Thanks to Alex Halderman for numerous discussions and encouragement. I would like to thank Vasanth Bala for encouragement when things were not really working out. I would like to thank David August and his students for enthusiastically helping me get acclimatised to Princeton after my arrival in USA. David even drove me (and Ram Rangan) to Udipi restaurant in my first week to make me feel at home.

My work has been supported by ARDA grant NBCHC030106, DARPA grant F30602-99-1-0519, and by New Jersey Commission on Science and Technology and Princeton University. Thank you for making this work possible.

I would like to thank my parents, brother and my wife for the support they have always given me in whatever I do. I owe a thank you to a large number of individuals and organizations that have helped me. Here are some names in no particular order: Varugis Kurien, Adrian Oney, Brandon Baker, Scott Field, Wayne Boyer, Xiaolan Zhang, John Lambert, Miles McQueen, Michael Steiner, Josyula Rao, Ruoming Pang, Daniel Dantas, Xinming Ou, and Matt Thomlinson.

To my wife, my brother and my parents for all the support

Contents

Chapter 1

Introduction

1.1 What is security management?

When a system administrator configures a network to keep it secure, he or she must consider the users, data, services, servers and most importantly the *policy*—which data should be accessible to which principals. But even the most experienced system administrator finds it daunting to keep track of the details of every component, all the mechanisms by which components interact and what data should be accessible to which principals. He could easily misconfigure the network so a hacker could steal confidential data. How does one enable a system administrator to securely configure the network with a limited understanding of its components and their interactions?

About one hundred new security vulnerabilities are found per week. It becomes even more difficult to manage the security of a network installation in the presence of a large number of security weaknesses in software under challenging time constraints. It is difficult to answer questions like *Is this bug relevant on my network? What is the best work-around?* In a large network, the diversity of software is high. When a new vulner-

ability is reported, it is quite likely that the vulnerability advisory is pertinent to some installed software on the network. Thus, with a high likelihood the vulnerability adversary is relevant to his network. Any security bug on the network could have network-level consequences. A diligent system administrator will have to perform a security analysis of the whole network each time a bug is reported—a cumbersome and error-prone task.

The costs of having a security vulnerability and taking actions to close it are high. Using a tool to help in these tasks will reduce errors, save costs by eliminating redundant actions, and streamline actions by prioritizing between multiple bugs and actions. It also gives the administrator a chance to plan ahead ("what if I find a problem with the web server?"). It also provides an opportunity to do rigorous risk estimation.

I define *Security Management* as the processes to enable the system administrator to configure the network securely with a limited knowledge of the components and the program vulnerabilities in these components. One can use sound software engineering practices to guarantee that an individual program is secure—yet be vulnerable to an attack if the file system permissions allow anyone to overwrite the executable or if an untrusted user is allowed to be a part of the trusted groups. In this thesis, we study the problem of ensuring that programs are configured correctly and that they do not wrongly use the operating system security primitives—even in the presence of program vulnerabilities. Most of the security management issues are currently performed manually in an ad hoc manner. In this thesis, we describe a framework to formalize management to introduce measurement, streamline the remediation process, and reduce mistakes and costs.

1.2 Why is security management hard?

The security of a network depends on two orthogonal elements: individual program security and host and network configuration. Methods to improve individual program security in improve individual programs security include using safe programming languages where possible, reducing trusted computing bases, using sound cryptographic primitives, and ensuring length-checked input fields. Tremendous advances have been made in software engineering—the art and science of writing secure programs. One can design and implement a very secure program—but still be vulnerable to an attack if the file system permissions allow an untrusted user to overwrite the secure program (or a kernel mode component of the operating system). The untrusted user can just replace the secure program with arbitrary code of his choice. There are host wide configuration elements—like group memberships, firewall rules, networked file system configurations—which can effect the security of all programs.

We classify security vulnerabilities into two classes—*program vulnerabilities* and *configuration vulnerabilities*. Program vulnerabilities are security problems caused in a single program by poor software engineering. Configuration vulnerabilities are security holes caused by configuration issues such as firewall rules, file system permissions and group assignment.

1.2.1 Configuration vulnerabilities

At installation time, programs modify configuration elements that affect the security of all users and programs of the host. In particular, an installation program might open an attack path through a previously installed program. For example, we have noticed that in Windows XP, many kernel mode driver programs set file protection that lets any member

of `Power Users` group overwrite the kernel-mode executable. If the installation of a program adds an untrusted user `joe` to the `Power Users` group, the host becomes prone to an attack by the untrusted user `joe` through the already installed kernel mode drivers.

Memberships of groups have to be carefully controlled. Adding an untrusted user to a trusted group can result in a security breach. However, it is difficult to determine what the trusted groups on a host are. For example, each Windows XP host has an built-in group `Power Configuration Operators`. A system administrator might be interested in determining the security consequences of adding a user to this group. I could not find documentation for the security behavior of this group. It would be very difficult for an administrator to determine if membership of this group is equivalent being a member of the `Administrators` group. To answer this question, one will have to look at each resource that `Power Configuration Operators` has access to and see if it is used by a privileged part of the operating system. Since the number of resources is large, answering this question is difficult.

As we will demonstrate later in the thesis, managing the security of a single host itself is hard—we have found serious holes in professionally managed machines. On a large network—where the number of configuration elements and their interactions between various elements is large—managing the security is even harder. System administrators are forced to *securely* configure hosts running different operating systems to inter-operate. For example, the administrators of the Princeton University network will have to ensure that users can access their files from Windows, Linux and Solaris hosts. These operating systems have different semantics and one needs to be careful to ensure that the differing semantics are mapped properly. There is no framework to reason about the correctness configuration issues and we address that problem in this thesis.

Vendor and software Name	Description	Common Name	Resources
Adobe Acrobat, Create Suite, Illustrator, InDesign, Pagemaker, Pagemaker Plus Photoshop, Premiere, and Version Cue various versions	Multiple vulnerabilities have been reported in multiple Adobe products that could let local malicious users obtain elevated privileges or execute arbitrary code. No workaround or patch available at time of publishing. There is no exploit code required.	Adobe Multiple Product Privilege Elevation or Arbitrary Code Execution CVE-2006-0525	Security Focus, ID: 16451, January 31, 2006

Figure 1.1: Sample CERT advisory, from US CERT bulletin SB06-033. I found these vulnerabilities using the tool set described in this thesis. CERT's advisory is based on these findings.

1.2.2 Program vulnerabilities

In addition to the configuration issues discussed above, security attacks are made possible because of a large number of security holes in individual programs. The United States Computer Emergency Readiness Team (US-CERT) releases a weekly compilation of new program vulnerabilities, exploits, trends, and malicious code that have been recently reported. Occasionally, they also update information about already known computer security risks—where they incorporate a new understanding of the computer security risks. A sample entry in the weekly bulletin is shown in figure 1.1. A typical weekly bulletin contains around 92 advisories regarding various security threats, as shown in table:

ID	Release date	Number of items
SB06-026	January 26, 2006	65
SB06-019	January 19, 2006	81
SB06-012	January 12, 2006	108
SB06-005	January 5, 2006	115
Average		92

When an administrator discovers a security vulnerability in his network, he is forced to take remediation measures quickly. Exploit codes for bugs become available shortly after a bug advisory has been published. The exploit is often available before the software vendor releases a patch. For example, a vulnerability was found in the WMF (Windows Media Format) handling engine in December 2005; the exploit code was widely available two weeks before a patch was available [41].

When a new vulnerability advisory is reported, it is quite likely that the advisory could affect the security of the network. The system administrator needs to identify if the bug is relevant to his network. He needs to identify the machines that are using the affected software and identify which individual installations are affected. He needs to determine if an adversary could exploit the bug. An adversary cannot exploit the bug if the affected module is disabled or is hidden behind a firewall. If the adversary can exploit a bug, the administrator will have to determine the network-level consequences. Then, the administrator needs to determine what is the best remediation measure. It is very common that a vulnerability advisory explains the details of security vulnerabilities in multiple products. Given the large number of vulnerabilities reported, it is very common that the administrator has multiple problems to attend to, but he cannot identify which are more important problems. There is no mechanism to prioritize the remediation actions.

To summarize, program vulnerability management is hard because of complex semantics for components, a large number of bug advisories, and limited response time. The procedures administrators adopt are ad hoc, cumbersome and error-prone.

1.2.3 Lack of quantitative risk measurement

The national critical infrastructure is at risk from malicious attacks through the public network because of reliance on networked control systems for the management of the infrastructure. For example, the national power grid is monitored by highly distributed Supervisory Control and Data Acquisition (SCADA) systems that attackers potentially may exploit to cause widespread power outages by remote control through the Internet from anywhere in the world. To combat this potential threat, the operators of control systems need the ability to more easily measure the current risk and the amount of risk reduction achieved by countermeasures. Similarly, the ability to quantify the risk in an enterprise network is valuable.

The community has been working on quantifying various aspects affecting security like user's skill level (*is this user likely to open email attachments?*), the perceived attacker's skill level, and the importance of the program affected by the bug. However, the risk estimations do not have a formal model for the software environment (in particular the operating system behavior) and the adversarial behavior. As a result the estimations tend to be ad hoc and error prone. For example, when a bug is reported in a library file, the vulnerability scoring systems ignore the surrounding software context like what programs are using this library, the privilege level of the programs using this library, and dependencies between various libraries. In an enterprise network too, it valuable to quantify the risk reduction achieved by actions of the administrator. The mechanisms to quantify the risk in current network posture are little understood.

1.3 What is the solution?

As we discussed in the previous section, configuration issues are orthogonal to software engineering issues. Professional system administrators and developers have a difficult time in understanding the security semantics of the operating system. Even in the absence of program vulnerabilities, we found that professionally managed networks have serious configuration holes. Program vulnerabilities make it even harder for the administrator to manage the security of a network. When a new vulnerability is reported, it is hard to determine if it is relevant on a network. It is even harder to understand the network-level consequences of a bug. The adversary could adopt an attack path using multiple vulnerabilities and misconfigurations in multiple software. A local (response) action by the system administrator like adding or deleting a firewall rule, or adding or deleting a user from a group can have global effects. It is a cumbersome and error prone task to reason about the transitive effects of multiple vulnerabilities and misconfigurations.

The solution we propose to these problems is an end-to-end scanning and analysis framework that assists the administrator. This solution is based on the insight that even though the individual components of a network are complex, each component has a very well defined deterministic behavior and a limited number of mechanisms by which it interacts with other components. By incorporating knowledge of different components such as operating system, users and groups, firewall rule sets and networked file systems, the tool could help the administrator in secure configuration. In addition to configuration bugs, there could be program vulnerabilities—bugs internal to the software. We use off-the-shelf vulnerability scanners to recognize the existence of already known bugs. We designed and built a configuration scanner to obtain some information not available from off-the-shelf products. Our framework computes the transitive closure of the effects

of configuration and program vulnerabilities over the whole network to see which principal can access what data. The framework flags accesses not allowed by the systems administrator as attack paths.

Although many programs are installed on any machine, we can do security analysis and find a large number of previously unknown configuration holes by asking simple questions like *who is using a program?*, and *who is allowed to modify a program and its configuration?*, *who are the users of the machine?*, and *what each user is allowed to do on each host?*. As we will show later in the thesis, we found **new** configuration vulnerabilities in the *Macromedia Dreamweaver* program from Macromedia Inc. and *Simple Service Discovery Protocol* and *Universal Plug and Play* programs from Microsoft Corporation by using this black-box approach.

We do not have to understand the programs' details to do this analysis. In the case of vulnerabilities, just information like whether a bug can be remotely exploited and whether it is a privilege-escalation bug or denial-of-service bug is sufficient for our analysis. Despite the large number of program vulnerabilities, the effects of the vulnerabilities can be classified into a small number of categories. The effects of a remotely exploitable privilege-escalation vulnerability in a *Simple Mail Transfer Protocol* server is the same as the one in the *HyperText Transfer Protocol* server—in both the cases the adversary gains control of the user account of the programs. Similarly, the security effects of bugs in *at* and *cron* daemons in Unix are similar—both give you access to the administrative account of the machine. A large amount of analysis is feasible by just taking a black-box approach to the programs.

We can analyze the network by just analyzing the configurations. This black-box approach to perform a complete network security analysis is very different from traditional program vulnerability analysis. In program vulnerability analysis, static/dynamic analy-

sis of the binary or source of the program has been proved to be effective. In our analysis, we do not analyze individual program binary or source code, but only information such as, who is allowed to modify the program executable, who is using it, what CERT vulnerability advisories apply to it and what is the formal semantics of those advisories.

1.4 Contributions

A central contribution of this thesis is the development of new techniques to reason about the correctness of configurations. This thesis develops a scalable logic-programming approach to automatically identify how the adversary can leverage multiple vulnerabilities to launch a multi-stage attack on a network. The contributions of this thesis are as follows:

- A formal model of the behavior of the Windows operating system (chapter 4).
- A formal model of the behavior of the Unix operating system (chapter 5).
- A framework to integrate the models of various components in a network such as operating systems, firewalls, networked file systems (chapter 5). This contribution is joint work with Xinming Ou and Andrew Appel and these results have been published [35, 34].
- A framework to automatically recognize how an adversary can launch mutli-stage attacks exploiting weak configuration and program vulnerabilities (chapters 5 and 4).
- A first step towards quantitative risk estimation (chapter 7).
- A first step towards analysis of potential attack scenarios (chapter 5).

- A design for improved vulnerability scanners that separate scanning and analysis phases, and significantly reduces the trusted computing base of current vulnerability scanners (chapter 6).

- A practical demonstration of our approach on the Windows operating system shows that software from several major vendors has serious configuration problems (chapter 7).

A key contribution of our work is the adoption of Datalog as the modeling language to integrate information from various sources. We showed that declarative specification and evaluation can overcome the scalability problems with previous approaches. Declarative specification permits the model to be clean, thus making it easier to debug our model. It allows us to cleanly separate specification from implementation details. This thesis demonstrated how one can integrate information from the following sources to perform an end-to-end network security analysis:

- Model of behavior of various components of the network like firewalls, networked file systems and operating systems.

- Formal specification of software vulnerabilities.

- Formal specification of the effects of exploiting a software vulnerability.

- Configuration information from a host.

- Output of readily available vulnerability scanners.

- Output of network infrastructure management tools.

- Formal security policy, specified by the administrator.

Implementation of the techniques presented in this thesis yielded a tool that administrators and developers could use to verify the software configuration of a single host (chapters 4 and 7). In fact, we used the tool to find serious vulnerabilities in software from major commercial vendors. The thesis also gives us a better understanding of current scanning technologies and proposes solutions to address their shortcomings.

1.5 Thesis organization

In chapter 2, I discuss how system administrators spend tremendous efforts in configuration and vulnerability management. I introduce the notion of a *security expert* and a *bug expert*. In chapter 3, I give an exhaustive overview of the Windows security model. I show how privilege escalation attacks are possible because of poor application of the security model. In chapter 4, I discuss how one can formally describe the Windows security model. I discuss how it is possible to automatically recognize privilege-escalation attacks in a Windows host made possible by poor configuration. In chapter 5, I show how to extend the Windows model to reason about multi-host, multi-stage attacks involving hosts running Windows and Unix, networked filesystems, and firewalls. I discuss how the framework can be used to plan for potential attack scenarios too. In chapter 6, I discuss the advances made in building vulnerability scanners and propose a design for better scanner to address the shortcomings of the current scanners. In chapter 7, I discuss how we used my tool to find privilege-escalation attacks on a single Windows host. I then discuss how my tool has found multi-stage attacks on a real network. I also discuss how the tool might be used to perform quantitative risk analysis in the future. I summarize the contributions in chapter 8.

Chapter 2

Background and Related Work

In this chapter, we outline the different challenges a system administrator has to face. An administrator will have to deal with the reality that many programs are insecure in their default installation. He will have to expend immense resources in configuring firewalls and certain individual programs. We discuss these issues in section 2.1. We then discuss the challenges of dealing with a large number of security bugs in section 2.2. We discuss recent advances in program vulnerability recognition in section 2.4.

2.1 Configuration Management

It is quite well known that many programs are insecure in their default installation and that the administrator has to expend significant efforts towards securing the configuration. These insecurities are different from the buffer overflow vulnerabilities that are typical—thousands of which are found per year. We now give some examples to illustrate that configuration is a significant problem in computer security. To make matters

worse, administrators and developers do not have tools that can study the overall impact of configuration across different programs. This thesis is an attempt to solve this problem.

Oracle A vulnerability advisory released in 2002 discusses some problems with Oracle 9i Application Server [7]:

- Exposing sensitive information
- Letting anonymous users deploy certain applications
- Poor access control on sensitive resources
- Using well-known default passwords
- Allowing remote command execution without authentication
- Using world-readable temporary files—thus exposing sensitive data
- Administrative interface using no authentication by default

Pete Finigan discusses how Oracle Database Server in its default configuration can lead to a large number of privilege-escalation attacks [19].

Novell Groupwise WebAccess Novell GroupWise WebAccess is an easy-to-use messaging system that offers a wide range of powerful communication and collaboration capabilities. It lets one send and receive mail messages, appointments, tasks, notes and attached files over the web. An advisory was released in 2003 that discusses how one could gain unauthorized access to a vulnerable server because of the poor configuration file. The configuration file allowed any user to access any file by default [42].

Applications for the Windows platform As we will show later in the thesis, the access control model of Windows is more general and complex than that of Unix. Because of the complexity of the model, the application and operating system developers need to be careful in using operating system model. In this chapter we will discuss how poor application of operating system model in commercial software from Adobe, Macromedia, Microsoft, AOL, IBM, Symantec, and Trend Micro results in privilege escalation.

Network firewall management A network firewall is a device that can filter network packets based on various attributes like the protocol, port, source, and destination address. Network firewalls are widely used to limit the ability of users and attackers to send network packets to parts of the network. With a large number of firewalls, it very quickly becomes difficult for an administrator to manage the firewall rules. There exist a large number of tools that can manage firewall rule sets automatically [23, 24, 4, 6].

2.2 Vulnerability management

In addition to configuration management, dealing with software vulnerabilities on network hosts poses a great challenge to network administration. With the number of vulnerabilities discovered each year growing rapidly, it is very difficult for system administrators to keep the software on their network machines free of security bugs. The different tasks for system administrators to manage their network's security are: vulnerability recognition, host configuration, network configuration, understand operating system semantics, and understand adversaries.

Does the advisory apply to software on my system? The crux of the problem is that there is no framework to automatically analyze the effects of a vulnerability advisory. In

the wake of new vulnerabilities, assessment of their security impact on the network is important in choosing the right countermeasures: patch and reboot, reconfigure a firewall, dismount a file-server partition, and so on. One of the daily chores of a system administrator is to read bug reports from various sources (such as CERT and BugTraq) and understand which reported bugs are actually security vulnerabilities in the context of his own network. Vulnerability reports are written in an ad hoc manner like this:

OpenSSH 3.x, 4.x; Red Hat Fedora Core3 & Core4	A vulnerability has been reported in 'scp' when performing copy operations that use filenames due to the insecure use of the 'system()' function, which could let a malicious user obtain elevated privileges.	CVE-2006-0225

An administrator will have to manually check if this report is relevant on his network— that is, if any version between 3.0 and 4.999 of OpenSSH is installed on any machine in the administrator's system— and then do further research to understand its implications. Typical vulnerability reports sent to discussions lists that are written in natural language are hard to read; the manual process of recognizing existence of a vulnerability in a host is labor intensive and error-prone. There is a need to automate the process of determining if vulnerable software is installed on a host.

Can the adversary reach the vulnerability? Upon recognizing that a vulnerability report is indeed relevant to his network, the administrator will have to understand the host and network configurations to see if the vulnerability can be exploited. If the affected client program is disabled (host configuration), then an adversary cannot exploit it. If the vulnerability is in a particular module of a program and the module is disabled, then the adversary cannot exploit it (program configuration). If the network uses a firewall

to block access to the affected server program, an external adversary cannot exploit it (network configuration). It is a tedious process for the administrator to understand the host and network configuration. For examples, firewall rule sets are notoriously difficult for a human to understand.

What happens when a bug is exploited? The result of exploiting a bug could be one of: denial of service, loss of confidentiality, loss of integrity, or privilege escalation. It is very hard to automatically understand the effects of a *data corruption* (loss of integrity) or a *user name leaked* (loss of confidentiality) bug. The effects of a denial-of-service bug can easily be quantified as a loss of service. The result of a privilege-escalation bug is easily quantified—the adversary obtains privileges of the vulnerable program. An adversary could use the additional privileges obtained to launch further attacks. About 80% of reported vulnerabilities are either privilege escalation or denial of service. The ability to model privilege-escalation and denial-of-service attacks yields substantial benefits in current systems. This experience in modeling these attacks is useful in extending the model to other attacks in the future.

Results of a privilege-escalation exploit When a privilege-escalation bug in a program is exploited, the adversary gets control of the privilege level of the vulnerable program. In Windows, a compromise of a program would result in the adversary gaining the privileges of the principal using the program. In Unix, a compromise of a program would result in the adversary gaining the privileges of the principal running the program only if the program is *not* a *setuid* program. In Unix, a compromise of a *setuid* program would result in the adversary gaining the privileges of the owner of the program file. In Windows, a compromise of most *service* programs results in a complete control of the host. Let us consider the *Remote Procedure Call* server programs in Windows and Linux. A

compromise of this program in Windows gives the adversary (among other things), the ability to shut down the host. A compromise of the same program in Linux does not give the adversary the ability to shut down the host.

The system administrator needs to have a very good understanding of details of the operating system to determine the damage after a privilege-escalation attack. A typical system administrator may not have a detailed understanding of the operating system semantics. In today's world, system administration is a highly specialized task. A web server administrator knows the intricacies of a web server from one vendor, but not necessarily that of the database or operating system or even web server products from other vendors. A Unix administrator knows the intricacies of Unix, but not necessarily the intricacies of Windows. An Oracle database administrator knows the details of configuration and semantics of Oracle database server, but not that of rival products from Microsoft or IBM or Sybase. Still, it is very common that an enterprise's business flows are critically dependent on the interaction of applications across different expert domains (web server, application server, database, and operating system). How does one guarantee that there are no gaps across the different expert domains? In practice is it infeasible for even a professional system administrator to understand the security semantics of all the programs on the network. Even if the administrator does know the semantics of all the programs on the network, how do we know that the administrator has not made a mistake in the cumbersome details?

2.3 Roles of security expert and bug expert

Determining the impact of exploiting a privilege-escalation bug requires a detailed understanding of the operating system semantics. It is hard for a typical system administrator

to have detailed understanding of the operating system security semantics. A system administrator really understands only the local elements of the network—the users and data. He also understands very well his intent—the local network security policy. It is hard for him to deal the complexity posed by the complex semantics of the individual components—like operating systems. We introduce the notion of a *security expert*—a principal who understands the detailed security semantics of the operating system. The operating system designer would be an ideal candidate for the security expert. For the system described in this thesis, I acted the role of a security expert.

Both the system administrator and the security expert cannot predict which program will have the next publicly reported vulnerability and what will be its consequences. We introduce the notion of a *bug expert* which is an abstraction for the bug reporting community. The work described in this thesis is about enabling the modularity of information flow between the system administrator, the security expert, and the bug expert.

2.4 Program vulnerability recognition

Each time a security is released, a diligent system administrator will have to determine if the bug affects his network. Traditionally, this determination is done manually—the system administrator has to watch the vulnerability discussion lists, read and the understand the natural language descriptions of the bugs and see if installations on his network are affected. As we discussed in chapter 1, about a hundred program vulnerabilities are reported each week. Even on a small network with about a hundred hosts, this manual examination of network is infeasible in practice. Recently, the bug-reporting community has started to provide these kinds of information in formal, machine-readable formats so that a program can automatically recognize the vulnerability.

2.4.1 The OVAL language and scanner

The Open Vulnerability Assessment Language (OVAL) [49] is an XML-based language for specifying machine configuration tests. When a new software vulnerability is discovered, an OVAL definition can specify how to check a machine for its existence. The OVAL schema supports multiple platforms—Windows, Solaris, HP-UX, and Linux in particular. A vulnerability is defined as a boolean combination of elementary tests on a host. Each elementary tests properties such as operating system version, architecture version, software version, file permissions and network servers listening for incoming connections. A sample vulnerability definition can be found in figure 2.1.

A OVAL definition can be fed to an OVAL-compatible scanner, which will conduct the specified tests on the host and report the result. Currently, Mitre Corporation provides OVAL vulnerability definitions for the Windows, Red Hat Linux and Solaris platforms. OVAL-compliant scanners are available for Windows and Red Hat Linux platforms. Mitre Corporation's OVAL vulnerability definitions have been created since 2002 and new definitions are being submitted and reviewed on a daily basis. Any principal can submit a new vulnerability advisory to be added to Mitre's OVAL Database. When the OVAL board at Mitre Corporation accepts the submission, the advisory is added to Mitre's OVAL bug database. As of January 31, 2005, the number of OVAL definitions for each platform is:

Platform	Submitted	Accepted
Microsoft Windows	543	489
Red Hat Linux	203	202
Sun Solaris	73	57
Total	819	748

```
<definition id="OVAL864" class="vulnerability">
      <reference source="CVE">CVE-2003-0542</reference>
      <criteria>
            <software operation="AND">
                  <criterion test_ref="rrt-206"
                        comment="Red Hat Enterprise 3
                              is installed" negate="false" />
                  <criterion test_ref="rvt-304"
                              comment="httpd version is less
                              than 2.0.46" negate="false"
                  />
            </software>
      </criteria>
      <tests>
            <rpmversioncompare_test id="rvt-304" >
                  <name datatype="string" operator="equals">
                        httpd
                   </name>
                  <tested_version operator="equals">
                        2.0.46
                  </tested_version>
            </rpmversioncompare_test>
            <rpminfo_test id="rrt-206">
                  <name operator="equals">
                        redhat-release
                  </name>
                  <version operator="pattern match">
                        ^3.S
                  </version>
            </rpminfo_test>
      </tests>
</definition>
```

Figure 2.1: A sample (Linux) OVAL definition

For example, we ran the OVAL scanner on one machine using the latest OVAL definition file and found the following vulnerabilities: [1]

```
VULNERABILITIES FOUND:
OVAL Id     CVE Id
-------------------------
OVAL2819    CAN-2004-0427
OVAL2915    CAN-2004-0554
OVAL2961    CAN-2004-0495
OVAL3657    CVE-2002-1363
-------------------------
```

Besides producing a list of discovered vulnerabilities, the OVAL scanner can also output detailed machine configuration information in the System Characteristics Schema. Some of this information is useful for reasoning about multistage attacks. For example, the protocol and port number a service program is listening on, in combination with the firewall rules and network topology expressed, helps determine whether an attacker can send a malicious packet to a vulnerable program.

The security community is converging towards using OVAL schema for vulnerability recognition. OVAL-compliant vulnerability scanners and definitions are available from vendors such as ThreatGuard, Red Hat, and Qualys.

[1] CVE is a list of standardized names for vulnerabilities and other information security exposures. CVE aims to standardize the names for all publicly known vulnerabilities and security exposures [14].

2.4.2 Vulnerability effect

One can find detailed information about the vulnerability effects from OVAL's web site[2]. For example, the OVAL description for the bug `OVAL2961` is:

> Multiple unknown vulnerabilities in Linux kernel 2.4 and 2.6 allow local users to gain privileges or access kernel memory, as found by the Sparse source code checking tool.

This informal short description highlights the effect of the vulnerability—how the vulnerability can be exploited and the consequence it can cause. If a machine-readable database were to provide information on the effect of a bug such as *OVAL2961 is only locally exploitable*, one could formally prove properties like *if all local users are trusted, then the network is safe from remote attacker*. Unfortunately, OVAL provides the information about the effect of a vulnerability only in natural language (English), not in a format with a formal semantics. Fortunately, the ICAT database [33] classifies the effect of a vulnerability in two dimensions: exploitable range and consequences.

- exploitable range: *local*, *remote*
- consequence: *confidentiality loss*, *integrity loss*, *denial of service*, or *privilege escalation*

A *local* exploit requires that the attacker already have some local access on the host. A *remote* exploit does not have this requirement. A typical local-access vulnerability is a buffer overrun in a local privileged program like the operating system kernel or a weak protection on a sensitive operating system or application file. An adversary requires unprivileged shell access to the host to exploit the vulnerability. A typical remote-access

[2] http://oval.mitre.org

vulnerability is a buffer overrun in a program that listens on the network waiting for incoming requests. Two most common exploit consequences are *privilege escalation* and *denial of service*. Currently all OVAL definitions have corresponding ICAT entries (the two can be cross-referenced by CVE Id). We recommend that the OVAL and ICAT be merged into a single database that provides both kinds of information. We use the above classification in determining the effects of exploitation of a vulnerability.

2.5 Related Work

There is a long line of work on automatic vulnerability analysis. Kuang formalizes security semantics of Unix as a set of rules, and conducts search for ways a system can be broken into based on those rules [3]. The Computerized Oracle and Password System (COPS) is a freely-available, reconfigurable set of programs and shell scripts that enable system administrators to check for possible security holes in their UNIX systems. The COPS tool includes the following components:

- **file.chk** and **dir.chk** checks to ensure that important files and directories such as `/etc/passwd, .profile, /etc/rc, /, /bin` and `/bin` are not world-writeable.
- **pass.chk** checks for poor password choices.
- **group.chk** and **passwd.chk** check for problems with the password and group files such as empty lines and null passwords.
- **cron.chk, rc.chk** checks to ensure that none of the programs that are started as a part of the system boot reference world-writeable files.

The COPS tool also incorporated the Kuang rule set. NetKuang is extended the rule set in Kuang by considering impact of configuration across the networks, such as the contents of `.rhosts` file. When these tools were authored, configuration was the major problem for network security and not software vulnerabilities. Thus, the focus is on identifying configuration weaknesses and not on integrating software vulnerabilities into the model.

Levitt and Templeton proposed a *requires* and *provides* model for computer attacks [46]. One attack provides capabilities that support the next attack, which in turn may provide new capabilities to support following attacks. It is not clear if the model has been implemented.

Ritchey and Amman proposed using model checking for network vulnerability analysis [39]. Sheyner, et. al extensively studied attack-graph generation based on model-checking techniques [43]. In MulVAL, instead of model-checking, we adopt a logic-programming approach and use Datalog in the modeling and analysis of network systems. The difference between Datalog and model-checking is that derivation in Datalog is a process of accumulating true facts. Since the number of facts is polynomial in the size of the network, the process will terminate efficiently. Model checking, on the other hand, checks temporal properties of every possible state-change sequence. The number of all possible states is exponential in the size of the network, thus in the worst case model checking could be exponential. However, in network vulnerability analysis it is normally not necessary to track every possible state change sequence. For network attacks, one can assume the *monotonicity property* — gaining privileges does not hurt an attacker's ability to launch more attacks. Thus when a fact is derived stating that an attacker can gain a certain privilege, the fact can remain true for the rest of the analysis process. Also, if at a certain stage an attacker has multiple choices for his next step, the order in which he car-

ries out the next attack steps is irrelevant for vulnerability analysis under the monotonicity assumption. While it is possible that a model checker can be tuned to utilize the monotonicity property and prune attack paths that do not need to be examined, model checking is intended to check rich temporal properties of a state-transition system. Network security analysis requires only a small fraction of model-checking's reasoning power. And it has not been demonstrated that the approach scales well for large networks.

Amman et. al proposed a graph-based search algorithms to conduct network vulnerability analysis [1]. This approach also assumes the monotonicity property of attacks and has polynomial time complexity. The central idea is to use an *exploit dependency graph* to represent the pre- and postconditions for exploits. Then a graph search algorithm can "string" individual exploits and find attack paths involves multiple vulnerabilities. This algorithm is adopted in Topological Vulnerability Analysis (TVA) [25], a framework that combines an exploit knowledge base with a remote network vulnerability scanner to analyze exploit sequences leading to attack goals. However, it seems building the exploit model involves manual construction, limiting the tool's use in practice. In MulVAL, the exploit model is automatically extracted from the off-the-shelf vulnerability database and no human intervention is needed. Compared with a graph data structure, Datalog provides a declarative specification for the reasoning logic, making it easier to review and augment the reasoning engine when necessary.

Datalog has also been used in other security systems. The Binder [18] security language is an extension of Datalog used to express security statements in a distributed system. In D1LP, the monotonic version of Delegation Logic [29], Datalog is extended with delegation constructs to represent policies, credentials, and requests in distributed authorization. The success of Datalog in Binder led us to examine its use in vulnerability analysis, and its successful application as a security language in Binder and now in our

work to network security analysis is a convincing demonstration that it is an excellent language for security analysis.

Recent works by Ramakrishnan and Sekar [37], and the one by Fithen et al [20] consider vulnerabilities on a single host and use a much finer grained model of the operating system than ours.The goal is to analyze intricate interactions of components on a single host that would render the system vulnerable to certain attacks. The result of this analysis could serve as attack methodologies to be added as interaction rules in MulVAL. Specifically, it is possible that one can write an interaction rule that expresses the attack pre and postconditions without mentioning the details of how the low-level system components interact. These rules can then be used to reason about the vulnerability at the network level. Thus the work on single-host vulnerability analysis is complementary to ours.

MulVAL leverages existing work to gather information needed for its analysis. OVAL provides an excellent baseline method for gathering per-host configuration information [49]. Also, research in the past ten years has yielded numerous tools that can manage network configurations automatically [23, 24, 4, 6]. Although these works do not directly involve vulnerability analysis, they provide a good abstraction for the network model, which is used in MulVAL and simplifies its reasoning process.

Intrusion detection systems have been widely deployed in networks and extensively studied in the literature [15, 32, 27]. Unlike IDS, MulVAL aims at detecting potential attack paths *before* an attack happens. The goal of the work is not to replace IDS, but rather to complement it. Perhaps, an administrator could use our system to identify strategically important locations on the network and deploy an intrusion detection system to detect attacks on these choke points. Having an a priori analysis on the configuration of a network is important from the defense-through-depth point of view. Undoubtedly, the more problems discovered before an attack happens, the better the security of the network.

A major difference between MulVAL and the previous works is that MulVAL adopts Datalog as the modeling language, which makes integrating existing bug databases straightforward. Datalog also makes it easy to factor out various information needed in the reasoning process, which enabling us to leverage off-the-shelf tools and yield a deployable end-to-end system.

2.6 Summary

Security management of a large network is hard because for two reasons:

- The semantics of the components is complicated.
- The number of details the administrator has to keep track of is large.

It is hard for a typical system administrator to know the nuances of components as well as an expert like the operating system kernel designer or Computer Emergency Response Team (CERT). Neither the administrator nor the security expert can predict the future buffer overflows that will be reported. For efficient network management, it is important to effectively leverage strengths of the system administrator, the security expert and the bug expert. In this thesis, I describe a framework to effectively modularize the information flow between these three principals.

Chapter 3

Introduction to Windows

Previous work has shown that it is possible to analyze a network comprising Unix hosts, networked file systems and firewalls to check if an adversary could leverage ubiquitous program vulnerabilities to launch a multi-step attack [34, 35]. We now illustrate that automatic configuration analysis is feasible and that even for a single host, this analysis can produce useful results. In particular, we built a model to analyze the configuration for the Windows platform. In chapter 7, we show that this is useful for analysis of configuration of even a single host—the model uncovered previously unknown serious security bugs.

Microsoft Windows NT, Windows 2000 and Windows XP use a general model to control access to resources. Unix has a simple access control model with three privileges given to users, groups, and others for operations on just a few kinds of objects (such as files and directories). In contrast, Windows attaches access-control lists (prioritized "allow" and "deny" by groups) comprising up to 30 different privileges for operations on about 15 different kinds of objects [10]. For example, on a "service" object one can have the privilege "choose what program.exe is run to effectuate the service."

3.1 Windows Objects

In Unix, one has to deal with various operating system objects like files, directories, threads and processes. One uses primitives like locks, message queues and semaphores for interprocess communication. One uses sockets for network programming. Standard textbooks provide excellent introduction to Unix and how to use these objects [2, 44]. In addition to these mechanisms, Windows provides other primitives, the most important of which are the *registry*, *services* and *Windows Management Instrumentation*. We now discuss these primitives briefly. For a more detailed treatment, we refer the reader to the textbook by Russinovich *et al.* [40].

3.1.1 Registry

The Windows registry is a centralized hierarchical database to store configuration information for the operating system and applications and services running under Windows. It is the repository for both system-wide and per-user settings. It is a vital resource of the operating system—the operating system does not boot if the registry is corrupted. In fact, the most common reason for the Windows operating system not to boot is a corrupted registry. The registry stores a wide range of configuration settings, from boot parameters to user desktop settings to program settings. Many of the Control Panel applets, command-line tools, and Microsoft Management Console (MMC) plug-ins that ones uses each day perform some of their functions by reading, editing, or adding registry subkeys or entries.

Because Windows provides the services of the registry, each individual application does not have to maintain its configuration in application specific configuration files. This frees the application developer from the hassle of designing and implementing application specific configuration files. Thus, the developer can use his time only in development

Key	Value
Current version	1.7.12
Geckover	1.0.1
Install Directory	C:\Program Files\Mozilla Firefox
PathToExe	C:\Program Files\Mozilla Firefox\firefox.exe
Description	Mozilla Firefox (1.0.7)
Uninstall Log Folder	C:\Program Files\Mozilla Firefox\uninstall
Plugins	C:\Program Files\Mozilla Firefox\Plugins
YahooPluginPath	C:\Program Files\Yahoo!\Shared\npYState.dll

Figure 3.1: Some sample registry keys

tasks. In contrast to Windows, Linux does not have the concept of a global database. Each program uses ad-hoc formats specific to itself; the net result is that each programmer has to expend effort in developing custom file parser and modifiers. Another disadvantage of this approach is that configuration analysis programs like the one described in this thesis are difficult to develop.

Each entry in the registry is indexed by a key. The keys are hierarchical in structure, just like files and directories. `A\B` refers to the key `B` that is a subkey of the key `A`. It is very common for programs to store application specific information under the root key `HKEY_LOCAL_MACHINE` (`HKLM`). Each registry key entry in the registry has a security descriptor that determines who can perform what access to the registry entry. We list some registry keys installed by the Mozilla browser in figure 3.1.

Registry key security. The operations that can be controlled by access-control on registry keys are: reading a key, writing a key, deleting a key, enumerating subkeys, adding a subkey, requesting notification for changes to this key or its subkeys. We discuss these in detail in section 3.2.5.

3.1.2 Services

Every operating system has a mechanism to start processes at system startup, providing functionality not tied to any particular user. For example, when the operating system boots up, one would want the network programs and the web server program to to start automatically, even if no user has logged on. Windows services are similar to Unix daemon processes, but more general. Some example services are:

- **Task scheduler** service is used to run a program at a designated time. The Unix equivalent of this service is `cron`.
- **Uninterruptible Power Supply** service is used to monitor the battery status of an UPS power supply through a serial port.
- **Windows Time Service** maintains date and time synchronization on all computers using Network Time Protocol.
- **Print Spooler** service queues and manages print jobs locally and remotely.
- **DHCP Client** service manages network configuration by registering and updating IP addresses and Domain Name Server (DNS) names.

One can control the following attributes for a Windows service:

- **Process sharing** controls whether the started service runs in its own process or shares the process with other services. All services that shares processes run under a shared auxiliary process.
- **Start type** controls whether the process starts during system startup or is started on demand by a user.

- **Error control** specifies the action to take if the service fails to start. Actions include logging, restarting the system in recovery mode, failing the startup and ignoring the error.
- **Binary path** specifies the path to the service executable
- **Account name** specifies the name of the account under which the service should run. This is usually one of `Local System`, `Local Service` and `Network Service`.
- **Dependencies** specifies the services that must be started before this service can be started.

Service security. The operations that can be controlled by access-control on services are: starting a service, stopping a service, reading the configuration of a service, modifying the configuration of a service, querying the status of a service, enumerate other services that are dependent on this service, pause or continue a service. We discuss these in detail in section 3.2.5.

3.2 Windows Security Overview

There are three pieces of information that are needed to make an access control decision:

1. Who is the principal requesting the access?
2. What are the intentions of the principal (specified in the request)?
3. What is the protection on the object to be accessed?

Windows uses the notion of *security identifiers* to identify principals. A *token* is a structure that stores the authorization attributes of a principal—such as whether he is a super user and the groups the user is a member of. A *security descriptor* is a per-object data structure maintained by the operating system that stores an object's security settings. An *access control strategy* is the algorithm the system uses to determine whether a requested access should be granted. We now briefly describe how Windows implements these notions.

3.2.1 Security Identifiers

Windows users *security identifiers (SIDs)* to identify various entities that perform actions in a system. A SID could represent n user, or a built-in account (like `Administrator`, and `Local System` accounts), or local and domain groups, or local computers, or domains, or domain members. A SID is a variable length binary value that contains information about the structure format, an authority number identifying the agent that issued the SID, a variable number of sub-authority values that identify trustees relative to the issuing authority. When a SID is displayed in clear text, each SID carries an *S* prefix, and its various components are separated with hyphens: *S-1-5-21-346327843-89743984-384343-1128*

SIDs are long and Windows takes care to generate random values for each SID. For our discussion the details of the SID structure are not important and it suffices to know that except for a few well-known SIDs, SIDs are globally unique numbers. Some well-known standard SIDs are:

- `S-1-1-0`. This SID represents the group `Everyone`.
- `S-1-5-11`. This SID represents the group `Authenticated Users`.

- `S-1-5-18`. This SID represents the `Local System` account, the account under which all operating system processes run. The closest equivalent to this account in Unix is the user with *userid* of 0. However, one cannot log on to system under this account.

- `S-1-5-19`. This SID represents the `Local Service` account. This account is used to run services that do not need administrative privileges and do not need access to the network.

- `S-1-5-20`. This SID represents the `Network Service` account. This account is used to run services which do not require administrative privileges, but need access to the network.

- `S-1-5-32-544`. This SID represents the `Administrators` group.

3.2.2 Account privileges

Windows provides a flexible access control model where one the owner of a resource can specify the level of access each user has. This model is useful in protecting access to a single object. However, sometimes users perform operations that have a system-wide impact. For example, the ability to shut down a system or change the time of a system has system-wide impact. The capability to perform these actions will have to be controlled carefully. Windows uses the notion of *privileges* to achieve this purpose. When a user logs onto a host, after authentication, the system identifies the privileges associated with the user and stores this information in the kernel as a part of the the process control block for the user's shell.

We now briefly discuss some of the important privileges in Windows, and refer the reader to the documentation for the details [11]. The various privileges in Windows are:

`SeAssignPrimaryTokenPrivilege` (replace a process-level token, described in section 3.2.3), `SeAuditPrivilege` (generate security audits), `SeDebugPrivilege` (debug arbitrary programs), `SeLoadDriverPrivilege` (load/unload device drivers), `SeChangeNotifyPrivilege` (skip directory traversal access checks, needed to receive notifications of changes to files and directories), `SeLockMemoryPrivilege` (lock pages in memory), `SeRemoteShutdownPrivilege` (force shutdown with a remote system), `SeBackupPrivilege` (open arbitrary files for reading bypassing any security checks), `SeRestorePrivilege` (open arbitrary files for writing bypassing any security checks), `SeSecurityPrivilege` (control and view audit messages and other functions), `SeShutdownPrivilege` (shut down the host), `SeTcbPrivilege` (act as a part of the operating system), `SeTakeOwnershipPrivilege` (take ownership of arbitrary files and objects), and `SeSystemtimePrivilege` (change the system time).

Super privileges There are a large number of privileges whose possession will enable the adversary to obtain complete control of the host. The `SeBackupPrivilege` and `SeRestorePrivilege` privileges let the process bypass the access checks on a file. Anyone with this privilege will be able to read and write any file (in particular the kernel) respectively. In fact, even the `SeBackupPrivilege` might be sufficient to give unlimited access to the system. In certain environments, depending on the operating system configuration, this privilege might allow the program to open a memory mapped file and read off passwords from the memory. The `SeLockMemoryPrivilege` can be exploited for denial-of-service attacks on a system. The `SeDebugPrivilege` enables a user open *any* process on the system, ignoring the security descriptor on the process and launch further attacks. The `SeTakeOwnershipPrivilege` enables a

user to take ownership of any object even if the user is otherwise not allowed access. The `SeLoadDriverPrivilege` can be used to load and unload device drivers, which run in kernel space—thus compromising the kernel. The `SeCreateTokenPrivilege` can be used to generate arbitrary user accounts with arbitrary group membership and privilege assignment. The `SeTcbPrivilege` allows the adversary to act as a part of the operating system. Using this privilege he could create a user shell (technically, a logon session) that includes the SIDs of more privileged users or groups and then obtain unauthorized access to their resources.

SeChangeNotifyPrivilege is a privilege that is usually granted to all users. This privilege allows a user to register for notification of changes to a particular file or directory. This privilege also allows a user to skip the access checks on the parent directories when trying to access a file or a directory. This feature allows accesses on deeply nested files to be efficient—by skipping access checks on the parent directories[1]. However, this feature means a restrictive access control setting on a parent directory is not sufficient to prevent access to the file or directory. We used our tool to discover that certain files in the operating system's *System Restore* directory have poor access control. As we will discuss in chapter 7, an adversary with `SeChangeNotifyPrivilege` may be able to leverage this weak access control to launch a subtle attack on the operating system through System Restore.

When a user logs onto the system, the system determines the privileges associated with the user and adds the list of privileges to the process attributes ("token") for the user's shell program. Each process inherits the attributes and hence the privileges from

[1]This does not result in security problems because when one creates a file, the operating system (optionally) automatically copies appropriate security descriptors from its parents. It takes more time to create files, and security descriptors are larger because of inherited permissions from parents, but accesses to files are quicker.

its parent. A process might temporarily disable or permanently remove privileges from its attributes. A system administrator can use the Local Security Policy Editor in the Administrative Tools folder of the Control Panel to assign privileges and rights to groups and accounts.

3.2.3 Token

A *token* is a per-process (per thread in some cases) data structure, maintained in the kernel as a part of the process control block, that contains the security information. It stores information regarding the user account, a list of account privileges for the user account, a list of SIDs representing the user, groups the user belongs to, the session identifier and other security related information associated with the process or thread. A token is created when a user logs on to a system and is attached to the initial process (typically `userinit.exe`) that is started on behalf of the user. A child process inherits its parent's token. When a process makes a certain request (like opening a file in write mode or opening a service to configure its properties), the kernel consults the token of the calling process to determine the privileges of the process. Tokens are not a fixed size data structure.

A *restricted token* is a special kind of token where some of the SIDs are marked "restricted". In section 3.2.6, we describe the access control algorithm—where given a process token, object protection, and requested access—Windows makes a decision whether to allow the access. When an access control decision is made using a restricted token, then the access control decision is made twice. The first time, it is made using the normal algorithm. The second time, the access control decision is made using the SIDs that are marked restricted. The access is granted only if both scans of the access control list grant the requested access.

3.2.4 Security Descriptor

A *security descriptor* is a per-object data structure identifying who can perform what action of the object. A security descriptor can be set on objects like processes, threads, semaphores, sections, waitable timers, registry keys, files and services. A security descriptor consists of the following attributes: a revision number (version of the security model), flags (controlling inheritance characteristics), owner SID, group SID a discretionary access-control list (specifying who has what access to the object) and a system access-control list (specifying which attempted operations by which users should be logged). For our discussion only the owner SID and discretionary access-control list (DACL) are relevant.

An *access-control list* (ACL) is a list of zero or more *access control entries* (ACEs) which say who is allowed what access to an object. A simplified Windows ACE would look like:

+/-	**Trustee**	**Mask**	Flags
+	Alice	`write`	INHERIT_ONLY
-	Bob	`read`	INHERITED
+	Everyone	`read`	NO_INHERIT

The first column in the ACE describes whether it is a positive or negative ACE. A positive ACE grants specified access and a negative ACE denies it. The second column indicates the target user or group (physically represented by a SID) and the third column specifies the permissions in question (physically represented as a 32-bit mask). The fourth column lists the flags that control how an ACE of a parent object (directory) propagates to its child objects (files).

3.2.5 ACE access mask format

Each Access Control Entry has a 32-bit mask that specifies the set of permissions associated with the entry. This 32-bit mask is interpreted differently, depending on the object type. The layout of various access masks is as shown in figure 3.6. These permissions are classified into the following three classes:

- **Object specific permissions.** Some permissions are specific to an object. For example, the file append permission `FILE_APPEND_DATA` is only meaningful for files. The permission `SERVICE_START`—one that allows a principal to start a service—is not meaningful for files and registry keys. The object specific permissions for files, directories, registry keys, and services are shown in figures 3.2, 3.3, 3.4, and 3.5 respectively.

- **Standard permissions.** Some permissions are common to all objects. These are:

  - `DELETE` To delete an object.
  - `READ_CONTROL` To read the security descriptor.
  - `WRITE_DAC` To write the security descriptor.
  - `WRITE_OWNER` To write the owner of the resource.
  - `SYNCHRONIZE` To wait on object handle.

  This standard set of permissions allows one to treat objects polymorphically. One would want to use this polymorphism to have a single default access-control list that can apply to all objects. However, one cannot have a default access-control list apply to different kinds of objects because certain permissions are object-specific. To solve this problem, Windows introduces the idea of *generic permissions*.

FILE_READ_DATA	0x0001	Read data from the file
FILE_WRITE_DATA	0x0002	Write data to the file
FILE_APPEND_DATA	0x0004	Append data to the file
FILE_EXECUTE	0x0020	The right to execute a file

Figure 3.2: File access rights

FILE_ADD_FILE	0x0002	Create a file in the directory.
FILE_ADD_SUBDIRECTORY	0x0004	Create a subdirectory.
FILE_DELETE_CHILD	0x0040	Delete a directory and all the files in it.
FILE_LIST_DIRECTORY	0x0001	List the contents of a directory
FILE_WRITE_DATA	0x0002	Create a file in the directory
FILE_APPEND_DATA	0x0004	Create a subdirectory
FILE_TRAVERSE	0x0020	Traverse a directory

Figure 3.3: Directory access rights

- **Generic permissions.** For each object type, these four permissions—`GENERIC_READ, GENERIC_WRITE, GENERIC_EXECUTE, GENERIC_ALL`—are mapped to a set of its standard and object-specific access rights. For example, the `GENERIC_READ` file permission is mapped onto `FILE_READ_DATA, FILE_READ_ATTRIBUTES, SYNCHRONIZE, READ_CONTROL,` and `FILE_READ_EA`. The `GENERIC_READ` registry permission is mapped onto `KEY_ENUMERATE_SUB_KEYS, KEY_NOTIFY, KEY_QUERY_VALUE,` and `READ_CONTROL`.

KEY_CREATE_SUB_KEY	0x0004	Create a subkey of a registry key.
KEY_ENUMERATE_SUB_KEYS	0x0008	Enumerate subkeys of a registry key
KEY_QUERY_VALUE	0x0001	Read the value of the key
KEY_SET_VALUE	0x0002	Set a value for the registry key
KEY_NOTIFY	0x0010	To request notification for changes in a key

Figure 3.4: Registry access rights

Access Right	Needed to
SERVICE_CHANGE_CONFIG	Change the configuration.
SERVICE_ENUMERATE_DEPENDENTS	Enumerate all dependent services.
SERVICE_INTERROGATE	Ask service to report its status immediately.
SERVICE_PAUSE_CONTINUE	Pause or continue the service.
SERVICE_QUERY_CONFIG	Query the service configuration.
SERVICE_QUERY_STATUS	Query the status of the service
SERVICE_START	Start the service.
SERVICE_STOP	Stop the service.
SERVICE_USER_DEFINED_CONTROL	Send user defined code to the service.

Figure 3.5: Specific access rights for service.

Bit	Flag
0-15	Object type specific rights
16	DELETE
17	READ_CONTROL
18	WRITE_DAC
19	WRITE_OWNER
20	SYNCHRONIZE
28	GENERIC_ALL
29	GENERIC_EXECUTE
30	GENERIC_WRITE
31	GENERIC_READ

Figure 3.6: Access mask format

3.2.6 Determining access

After an adversary has gotten hold of a process, he tries to access resources from this privilege level. We now describe the algorithm the Windows kernel uses to determine whether an access should be allowed. In making this decision, the kernel considers the following inputs:

- The authorization attributes for the principal requesting access. This information is available by looking at the process token of the process requesting access.
- The intentions specified in the request.

- The security settings for the object to be accessed. The protection level of the object is expressed in the Discretionary Access Control List (DACL) of the object.

No ACL implies no protection If an object does not have a discretionary access control list, then any access is permitted on the object. Such a DACL is called a *Null DACL*.

SeTakeOwnershipPrivilege privilege gives write-owner access The privilege SeTakeOwnershipPrivilege in the caller's token gives WRITE_OWNER access to any resource. With WRITE_OWNER permission, one can change the owner SID of a resource to one of the SIDs in the caller's process token. (Technically, the SID in the process token will have to be marked as having the potential for being an owner.) After obtaining the ownership of a resource, the adversary will be able to get full control of the resource by launching further attacks, as described below. To summarize, the SeTakeOwnershipPrivilege will give the adversary complete control over all resources on a host, thus resulting in system-wide compromise.

Owner always gets access The owner of a resource always gets WRITE_DAC access. The owner can use the WRITE_DAC permission to reset the ACL to give an arbitrary entity arbitrary access. Thus, the owner of a resource can always get full access to the resource.

Consult the Access Control List If none of the previous rules apply, then the kernel consults the access control list. Each Access Control Entry (ACE) in the access-control list is examined from first to last looking for an entry that denies or allows the action. An ACE is processed if the ACE is an access-deny or access-allowed ACE and the SID in the ACE matches a SID in the caller's access token. If it is an access-denied ACE, then

the access is denied. If the ACE is an access-allowed ACE, then the access is allowed provided the process token is not a restricted token (refer section 3.2.3). If the ACE is an access-allowed ACE and if the process is a restricted token, then the system rescans the ACL's ACEs looking for ACEs with access matches for the access the user is requesting and a match of the ACE's SID with any of the callers restricted SIDs. Only if both scans of the ACL grant the requested access right is the user granted access to the object. If the end of the list is reached without a matching ACE, the request is denied.

3.3 Privilege escalation

It is possible that weak configuration on a file or registry key's security descriptor can allow an adversary to modify the resource state. (We will later show in chapter 7 that this indeed is the case.) Then, the adversary can inject corrupted data into a higher privileged process reading the corrupted data. Sometimes, the data corruption presents the adversary with an immediate opportunity to attack. If the adversary is able to overwrite a driver file, then the adversary can corrupt the kernel and hence get complete control of the host. If an adversary can write to a program used in the operating system startup, then the adversary can completely compromise the integrity of the host and all its users. Alternatively, the adversary will try to write to a dynamically linked library that is loaded by a more privileged process. If the adversary can write to a registry key storing the executable or library name to be executed under certain conditions, it provides the adversary with another avenue of attack. For example, when a user logs on to a host, the operating system looks at the value in the key `HKLM\SOFTWARE\Microsoft\WindowsNT\CurrentVersion\Winlogon\Userinit` and executes the file name referenced as the first program running on behalf of the user. Sometimes, a registry key stores the name of a

library that is loaded by processes. If a process loads or executes the file references by a registry key, and if the adversary can write to the registry key, then the adversary can compromise the process. If the adversary can inject data into a process executing at a higher level privilege level, he can immediately gain access to the higher privilege level.

For any object, a `WRITE_DAC` permission can be used to change the security descriptor of the object and then take control of the object. A `FILE_WRITE` permission can be used overwrite a file and then take control of the principal executing the file or loading the library. A `KEY_WRITE` can be used to overwrite the contents of the registry key and take control of whoever trusts this registry key. A `SERVICE_CHANGE_CONFIG` can be used to overwrite the attributes of the service. In particular, the adversary could configure a malicious program to be executed when the service is started. He could also configure the user account which runs the program to be an administrator. If the adversary could cause the service to be restarted (say by rebooting the machine), the malicious program runs as an administrator. After compromising the administrative account, the adversary can get complete control of the host and any principal trusting the host.

3.4 Summary

The number of types of objects in Windows is large. The most important object types used by adversaries in attacking Windows are files, registry keys, and services. Windows uses a complex access control model to protect access to objects. In traditional Unix, there is no mechanism to specify that a particular user (who is not already the file owner) gets read access to a file. One can only specify this at a granularity of the current group the user belongs to. In contrast, in Windows it is possible to specify that a particular user gets read access to a resource. Similarly, in Unix it is not possible to specify that a

user can only append a file and not overwrite its existing contents. In Windows, one can granularly specify that a user can perform a particular operation such as append a file, change the security settings of a resource, start/stop a service, and get notified when a file content changes. Thus, Windows uses a flexible and general access control model to control access to resources.

The downside to this general model is that even professional software developers do not understand the model correctly and wrongly apply the model. A wrong security descriptor on an object provides the adversary to launch a privilege escalation attack. Thus software from major vendors has serious privilege-escalation vulnerabilities because of wrong application of access control model. In the next chapter, we will show how to model the semantics of Windows formally. In chapter 7, we discuss how the model finds privilege escalation vulnerabilities.

Chapter 4

Formal modeling of Windows

In this chapter, we demonstrate how to model the access control semantics of Windows (Windows NT, Windows 2000 and Windows XP) as described in the last chapter. We use the formal model of access control semantics of Windows to automatically identify methods by which an adversary can increase the privileges he holds. We show in section 7.1 how we used our framework to find serious security bugs in configuration of software from several major vendors.

4.1 Datalog overview

Our system adopts Datalog as the modeling language for elements in the analysis. We found that declarative semantics is sufficient for encoding the interactions of operating systems, network firewalls, and file systems and for modeling attacks using program vulnerabilities. Datalog allows us to cleanly specify the semantics, thus it is easy to determine if we made a mistake in understanding the operating system model. Datalog

has the additional advantage that evaluation of all facts implied by a Datalog program is theoretically polynomial in complexity and efficient in practice.

Syntactically, Datalog is a subset of Prolog [9] with limited forms of clauses. The reasoning rules in our system are declared as Datalog clauses. A *literal*, $p(t_1, \ldots, t_k)$ is a predicate applied to its arguments, each of which is either a constant or a variable. In the formalism of Prolog, a variable is an identifier that starts with an upper-case letter. A constant is one that starts with a lower-case letter. Let $L_0, \ldots, L_n$ be literals. A reasoning rule in Datalog is represented as a Horn clause:

$$L_0 \ \texttt{:-}\ L_1, \ldots, L_n$$

Semantically, it means if $L_1, \ldots, L_n$ are true then L_0 is also true. The left-hand side is called the *head* and the right-hand side is called the *body*. A clause with an empty body is called a *fact*. A clause with a nonempty body is called a *rule*. A Datalog program is a set of facts and reasoning rules to infer from these facts. The execution of a Datalog program infers from these facts using the reasoning rules. The significant differences between Datalog and Prolog are:

- **Data constructors.** Datalog does not have data constructors; each of the arguments to a predicate is either a constant or a variable. In Prolog, a data constructor can be supplied as an argument. One cannot encode a list as an argument to a Datalog predicate; the argument to a Prolog predicate can be a data constructor that encodes a list.

- **Declarative semantics.** Datalog has pure declarative semantics. The order in which clauses appear in a Datalog program does not affect its logical meaning and the evaluation result. In Prolog, such order is important and affects the result

of evaluation [9], due to the depth-first search strategy and side-effect of operators like *cut*.

- **Negation.** Datalog does not have negation and one cannot express rules like *Alice will buy a mango if she cannot find an orange*. Adding a new fact to a Datalog program cannot invalidate any fact that can be derived from the program. (Technically, the negation operator in Prolog is implemented using the cut operator.)

- **Polynomial Termination.** Prolog is a Turing-complete language—computation may never terminate. In contrast, the complexity of determining whether a literal is implied by a Datalog program is polynomial in the size of the program [16].

Datalog has been used as a security language for expressing access control policies [18]. The efficiency of Datalog and existing off-the-shelf Datalog evaluation engines [48, 38] makes it readily usable in practice. Datalog is attractive for the formal model of the security analysis described in this dissertation because it gives us a clear specification of the semantics of network components such as operating systems and networked filesystems.

4.2 Modeling the Windows Access Control algorithm

We now show to model the semantics of Windows access control. We adopt Datalog for the model. We use certain constructs that are not in pure Datalog, As we show in section 4.4, the usage of these constructs does not adversely affect the running time of our program.

4.2.1 Object protection

An **access control entry** is encoded as the primitive predicate `ace( aceType(Type), aceRights(RightsList), Sid)` that specifies that an access control entry of type `Type` (one of `ACCESS_ALLOWED_ACE_TYPE` or `ACCESS_DENIED_ACE_TYPE`) grants or denies to the entities represented by the identifier `Sid` the rights specified in the list `RightsList`. An example usage is:

```
ace(aceType('ACCESS_ALLOWED_ACE_TYPE'),
      aceRights(['FILE_WRITE_DATA']),
      sid('S-1-5-21-854245398-1637723038-725345543-1003')).
```

The predicate `dacl(AclList)` encodes a **discretionary access control list (DACL)**, where `AclList` is a list of access control entry predicates, storing the entries in the same order as they appear in the security descriptor. In case the object does not have any protection ("null DACL"), we encode this as `dacl(null)`. If the object has an DACL of length zero, then the DACL is encoded as `dacl([])`. All other DACLs are encoded as `dacl([HeadAce | ACLTail])` where `HeadAce` is a predicate that describes the first access control entry and `ACLTail` is a list of predicates for subsequent access control entries. A sample DACL predicate is:

```
dacl([
   ace(aceType('ACCESS_ALLOWED_ACE_TYPE'),
      aceRights(['FILE_WRITE_DATA']),
      sid('S-1-5-21-854245398-1637723038-725345543-1003'))
   |
   [  ace(aceType('ACCESS_ALLOWED_ACE_TYPE'),
      aceRights(['FILE_READ_DATA']),
```

```
        sid('S-1-5-21-854245398-1637723038-725345543-1003'))
    ]
  ])
```

A **security descriptor** is encoded as the predicate

```
securityDescriptor(Owner, Dacl)
```

where `Owner` represents the security identifier of the owner and `Dacl` is a predicate that encodes the discretionary access control list.

When a kernel makes an access control decision, the decision will require the ability to decide what a single access control entry means. The predicate `checkACE(Result, AceEntry, RequestedAccess, SidsList)` is the predicate that models the decision making at a granularity of a single access control entry. `Result` is one of `allowed` or `denied`, `AceEntry` is an access control entry predicate, `RequestedAccess` is the access requested and `SidsList` is a list of security identifiers of the groups in the process token. The predicate `checkACE(Result, AceEntry, Access, SidsList)` means that an elementary access control decision, using the access control entry `AceEntry`, for a request `Access` by a principal whose list of security identifiers of the groups in the process token is `SidsList`, is `Result` (one of `allowed`, `denied`).

```
checkACE(allowed,
        ace( aceType('ACCESS_ALLOWED_ACE'), AceRights, Sid),
        Access, SidsList) :-
        accessInAceMask(Access, AceRights),
        sidInGroup(Sid, SidsList).
```

```
checkACE(denied,
        ace( aceType('ACCESS_DENIED_ACE), AceRights, Sid),
        Access, SidsList) :-
        accessInAceMask(Access, AceRights),
        sidInGroup(Sid, SidsList).
```

The predicate `sidInGroup(Sid, SidList)` recursively searches the `SidList` to see if `Sid` is present:

```
sidInGroup(Sid, [ Sid | _  ]).
sidInGroup(Sid, [ _ | Tail]) :-
        sidInGroup(Sid, Tail).
```

The predicate `checkAccessList(Result, RequestedAccess, Acl, SidsList)` models the algorithm the kernel uses to decide whether an access control list `Acl` allows or denies `RequestedAccess` to a principal with `SidsList` as the list of security identifiers of the groups in the process token. This predicate examines the `Acl` from first to last and unifies the `Result` variable to `allowed` or `denied` accordingly. If there is no access control on the object, then the request is granted. If the end of the list is reached, then access is denied. Formally, we write this as:

```
checkAccessList(allowed, Access, dacl(null), SidsList).


checkAccessList(Result, Access,
  dacl(acl([AclHeadEntry| AclTail])), SidsList) :-
      (
        checkACE(Result, AclHeadEntry, Access, SidsList);
```

```
        checkAccessList(Result, Access,
                        dacl(acl(AclTail)), SidsList)
      ).

% An empty access control list denies access
checkAccessList(denied, Access,
  dacl(acl([])), SidsList)
```

In the formalism of Prolog, an identifier starting with _ (like `_RequestedAccess`) is an anonymous variable, and it can bind to any value. So, in the above rule the predicate `checkAccessList(allowed, _RequestedAccess, dacl(null), _SidsList)` means that it does not matter what the requested access is and what SIDs are a part of the process token of the calling process.

4.2.2 Process credentials

When a principal access an object, the kernel looks up the process token of the process making the request to determine its credentials: the user, the privileges, the groups the user belongs to and the restricted groups the user belongs to. The predicate `processToken(UserSid, Privileges, Groups, RestrictedGroups)` encodes the credentials of the process requesting the access. `UserSid` is the user identity on behalf of whom the process runs, `Privileges` is the set of privileges (like `SeTakeOwnershipPrivilege`, `SeSystemtimePrivilege`) the process holds, `Groups` is the set of groups the user belongs to and `RestrictedGroups` is the set of restricted groups the user belongs to. A sample process token looks like:

```
processToken( 'S-1-5-21-1214440339-507921405-1060284298-500',
```

```
    privileges([
      'SeBackupPrivilege',
      'SeChangeNotifyPrivilege',
      'SeSystemtimePrivilege'
    ]),
    groups(['S-1-1-0', 'S-1-2-0',
            'S-1-5-11', 'S-1-5-32-544']),
    restrictedGroups([])
  )
```

4.2.3 Modeling access check

The predicate `windowsAccessCheck(Result, ObjectProtection, RequestedAccess, RequestingToken)` models the algorithm the kernel uses in determining whether to permit an `RequestedAccess` access to an object with protection `ObjectProtection` by a process with token `RequestingToken`. The variable `Result` is instantiated to `allowed` or `denied` accordingly. In section 3.2.6, we described the algorithm the kernel uses to make an access control decision. We now formally describe the algorithm.

No ACL implies no protection. If a file does not have an Access Control List ("Null DACL"), then any access is permitted on the file. The formal rule is:

```
windowsAccessCheck(allowed,
        securityDescriptor( Owner, dacl(null)),
        RequestedAccess,
        RequestingProcessToken).
```

SeTakeOwnershipPrivilege privilege gives write-owner access. The privilege SeTakeOwnershipPrivilege in the caller's token gives WRITE_OWNER access to any resource. With WRITE_OWNER permission, one can change the owner SID of a resource to one of the SIDs in the caller's process token. (Technically, the SID in the process token will have to be marked as having the potential for being an owner.)

After obtaining the ownership of a resource, the adversary will be able to get full control of the resource by launching further attacks, as described below. Similarly, as we discussed in section 3.2.2, any super privilege will give the adversary complete control over all the resources of the host. In our model, we encode these multi-step attacks as a single step as follows:

```
windowsAccessCheck(allowed, SecDescriptor, RequestedAccess,
        processToken(Owner, PrivList, GroupSids,
                     TokenRestrictedSids)
) :-
hasSuperPrivilege(true, PrivList).
        %Check if token has a ‘‘ super’’ privileges
```

Owner always gets access. The owner of a resource always gets WRITE_DAC access. The owner can use the WRITE_DAC permission to reset the ACL to give an arbitrary entity arbitrary access. Thus the owner of a resource always get full access to the resource. This is expressed as:

```
windowsAccessCheck(allowed,
        securityDescriptor(Owner, Dacl),
        RequestedAccess,
        processToken(Owner, PrivList, GroupSids,
```

```
                    TokenRestrictedSids).
```

Consult the Access Control List. We now need to evaluate the access control list from first to last, trying to see if we come across an entry that allows or denies the access. Formally, we write this as:

```
windowsAccessCheck(allowed,
        securityDescriptor(ObjectOwner, dacl(Acl)),
        RequestedAccess,
        processToken(ProcessOwner, PrivList, Groups,
                    TokenRestrictedSids)
)  :-
       checkAccessList(allowed, RequestedAccess, Acl, Groups),
      (  processIsNotRestricted(TokenRestrictedSids) ;
           /* ; is the Prolog OR operand. */
         processIsRestricted(TokenRestrictedSids),
         checkAccessList(allowed, RequestedAccess,
                Acl, TokenRestrictedSids)
      ).
```

Everything else is denied. If none of the above rules match, then the access cannot be allowed and hence access is denied. We model this as:

```
windowsAccessCheck(denied, SecurityDescriptor,
                  RequestedAccess, ProcessToken) :-
        not windowsAccessCheck(allowed, SecurityDescriptor,
                           RequestedAccess, ProcessToken).
```

Atomic permissions The predicates that we discussed above—`windowsAccessCheck`, `checkAccessList` and `checkACE`—take as argument the access requested. In our model, the `RequestedAccess` argument is an atomic permission like `FILE_READ_DATA` and `FILE_READ_ATTRIBUTES`. In practice, one requests more than one permission, like `FILE_READ_DATA` and `FILE_WRITE_DATA`—to read and write a file. To check for such nonatomic permissions, we will have to call the corresponding access check function on each atomic permission requested. This is a deviation from the way the kernel implements the algorithm. However, this deviation is functionally equivalent when used with atomic predicates. The kernel implementation avoids calling the function more than once for nonatomic permission by doing some clever bit manipulation. Since it is not straightforward to do bit manipulation in Datalog, in our model access check can only be called on an elementary permission like `FILE_READ_DATA`. This makes our reasoning rules easier to write and debug and thus increases the assurance of our system. However, Windows allows a principal to request more than one permissions—like all of `FILE_READ_DATA, FILE_WRITE_DATA, DELETE, READ_CONTROL, WRITE_DAC`—simultaneously. To model the behavior of Windows in our model, when a principal requests both `FILE_WRITE_DATA` and `WRITE_DAC`, we invoke the predicate `windowsAccessCheck` twice—the first time with `FILE_WRITE_DATA` and the second time with `WRITE_DAC`.

4.3 Modeling privilege escalation

We use the predicate `resource(Type, Name, Dacl)` to identify various resources on a host. `Type` indicates the type of the resource—it is one of `service`, `registry`

and `file`. `Name` identifies the resource and `Dacl` is the protection on the resource. By scanning the host, one could generate the list of all the resources on a machine.

The predicate `userToken(Principal, Token)` identifies that the principal `Principal` gets the token `Token` when he logs in. We generate this predicate for each user on the machine.

`canWrite(Principal, resource(Type, Name, Dacl))` is a derived predicate that specifies that principal `Principal` can write the resource of type `Type` (`service`, `registry`, `file`) identified by `Name` and with a security descriptor `Dacl`.

If the adversary has `WRITE_DAC` or `GENERIC_WRITE` permissions, he can write to the resource. We write this formally as:

```
canWrite(Principal, resource(Type, Name, Dacl)) :-
        userToken(Principal, ProcessToken),
        windowsAccessCheck(allowed, Dacl,
                           'WRITE_DAC', ProcessToken).

canWrite(Principal, resource(Type, Name, Dacl)) :-
        userToken(Principal, ProcessToken),
        windowsAccessCheck(allowed, Dacl,
                           'GENERIC_WRITE', ProcessToken)
```

If the adversary has `FILE_WRITE_DATA` permission on a file, he could overwrite the file. We write this as:

```
canWrite(Principal, resource(file, Name, Dacl)) :-
        userToken(Principal, ProcessToken),
```

```
        windowsAccessCheck(allowed, Dacl,
                           'FILE_WRITE_DATA', ProcessToken).
```

If the adversary has `KEY_SET_VALUE` permission on a registry key, he could overwrite the contents of the key. We write this as:

```
canWrite(Principal, resource(registry, Name, Dacl)) :-
        userToken(Principal, ProcessToken),
        windowsAccessCheck(allowed, Dacl,
                           'KEY_SET_VALUE', ProcessToken).
```

If the adversary has `SERVICE_CHANGE_CONFIG` permission on a service, he could reconfigure the service. We write this as:

```
canWrite(Principal, resource(service, Name, Dacl)) :-
        userToken(Principal, Token),
        windowsAccessCheck(allowed, Dacl,
                           'SERVICE_CHANGE_CONFIG', Token);
```

`trusts(Principal, Resource)` is a predicate that specifies that `Principal` trusts `Resource`. If a principal executes code in a file, he trusts the file. Since any one who can configure a service can get administrative access to a machine indirectly, an administrator should trust any service resource. We write this as:

```
trusts(Administrator, resource(service, Name, Dacl)).
```

If a principal `Target` trusts a `resource(Type, Name, Dacl)` and if a principal `Attacker` can write to this resource, then the adversary `Attacker` can launch a privilege escalation to `Target`. This is formally encoded as:

```
execCode(Attacker, Target) :-
        canWrite(Attacker, Resource),
        trusts(Target, Resource).
```

4.4 Discussion

In the formal description of Windows, we used two constructs that are not allowed in pure Datalog: negation and lists. But the use of negation in this program has a well-founded semantics [21]. The complexity of a Datalog program with well-founded negation is polynomial in the size of input [17]. Similarly, when we use lists, we are not constructing new data structures from them. In fact, each time a list is consulted to make a decision, the size of the list is decreased. In the absence of lists, the program terminates in polynomial time. Since our usage of lists is well-founded, our program terminates in polynomial time.

We used our tool to see how software from various vendors is configured in the default installations. We found that unprivileged users on a Windows XP host can obtain administrator privileges through misconfigurations. We discuss the detailed findings of our automatic analysis tool in section 7.1. These misconfigurations can be classified as follows:

- **Files.** Some vendors' software is vulnerable to a traditional file-system-based Trojan-horse vulnerability. The executable files in the software distribution are configured to allow an untrusted guest user to overwrite the files; thus a guest user can introduce malicious code into the executables.

- **Registry.** Each registry key has a security context attached to it controlling access to the key. Some registry keys store sensitive information like the path to the exe-

cutable acting as an user's shell, the library to be loaded by a program, the identity of an operating system object[1] etc. If an adversary can overwrite the contents of a sensitive key with the path of his library or executable, he could cause his code to be executed [47, 28]. The standard configuration of software from several vendors allows an untrusted guest user to overwrite sensitive registry keys.

- **Services.** Several vendors poorly apply the Windows access control model to their services; a common mistake is to assign the `SERVICE_CHANGE_CONFIG` permission indiscriminately to untrusted users. This permission allows a principal to set both the executable and the account under which the service runs. Using this permission, an untrusted guest user could cause an evil program to run as an administrator.

[1] Windows identifies certain operating system objects ("classes") by globally unique identities like *4D36E96B-E325-11CE-BFC1-08002BE10318*

Chapter 5

Analyzing multi-stage attacks

Single-host privilege escalations are useful to the adversary only if he has access to a host. Typically, adversaries adopt a sequence of attack steps to reach a final target. To obtain access to sensitive resources, besides Windows hosts, adversaries have to attack other components in the network such as Unix hosts, networked filesystems, and database systems. The number of components in a large network is high, complexity of component interaction is high, the semantics of components are nontrivial, and sometimes the components have known security bugs. Even professional system administrators have difficulty in managing the security of a small network with a couple of hundred users. How does administrator protect the network from mistakes he could make? On a large network, the problem quickly becomes intractable.

In chapters 3 and 4, we discussed the security model of the Windows operating system. We described how one can automatically analyze a single host to determine if privilege escalations are possible because of bad configuration. In this chapter we show that the solution to the large network management problem is a rule-based expert system that integrates configuration scanning, vulnerability scanning and automatically

determines the transitive closure of all security attacks on the network. After determining all possible accesses, the framework identifies accesses that are not allowed by the administrator—these are security attack paths. The tool we built is called Multihost, Multistage Vulnerability Analyzer (MulVAL).

MulVAL reasoning rules specify semantics of different kinds of exploits, compromise propagation, and multi-hop network access. The MulVAL rules are carefully designed so that information about specific vulnerabilities is factored out from the data generated by vulnerability scanners. The interaction rules characterize general attack methodologies (such as "Trojan Horse client program"), not specific vulnerabilities. Thus the rules do not need to be changed frequently, even if new vulnerabilities are reported frequently. We have predicates in our system that model each of the following properties:

- Vulnerabilities
- Exploit propagation
- User and data binding
- User behavior
- Network behavior
- Host configuration
- Security policy

In the rest of this chapter, we describe in detail various components of our system.

5.1 The MulVAL interaction rules

5.1.1 Vulnerability rules

`vulScannerOutput(Host, VulId, ProgramPath)` is a predicate specifying that a vulnerability with identity `VulId` has been found in the program located at the path `ProgramPath` on host `Host`.

`vulProperty(VulId, ExploitRange, ExploitConsequence)` is a predicate specifying whether the vulnerability `VulId` can be exploited locally or remotely and what happens when the vulnerability is exploited. `ExploitRange` is one of `local` or `remote`. `ExploitConsequence` is one of `privilegeEscalation`, `denialOfService`, `confidentialityCompromised` and `integrity compromised`.

`dependsOn(Host, ProgramPath, LibraryPath)` is a predicate that specifies on the host `Host`, the program at the path `ProgramPath` uses the library at path `LibraryPath`.

`vulExists(Host, Program, ExploitRange, ExploitConsequence)` is a derived predicate that specifies that a vulnerability exists in a host along with information about whether it is remotely exploitable and its consequences. We derive this predicate as follows:

```
vulExists(Host, Program, ExploitRange, ExploitConsequence) :-
        vulProperty(VulId, ExploitRange, ExploitConsequence),
        vulScannerOutput(Host, VulId, Program).
```

If there is a security bug in a library used by a program, then the program could be vulnerable. We express this as:

```
vulScannerOutput(Host, VulId, ProgramPath) :-
```

```
        vulScannerOutput(Host, VulId, LibraryPath),
        dependsOn(Host, ProgramPath, LibraryPath).
```

We need to capture transitive dependencies—if library `Library1` uses `Library2` and library `Library2` uses `Library3`, then `Library1` uses `Library3`. We write this formally as follows:

```
dependsOn(Host, Library1, Library3) :-
        dependsOn(Host, Library1, Library2),
        dependsOn(Host, Library2, Library3).
```

5.1.2 Exploit modeling rules

The details of exploiting each vulnerability are unique. For example, the specific details of exploiting a buffer overflow bug in *Simple Mail Transfer Protocol (SMTP)* server would be different from a buffer overflow bug in *Hypertext Transfer Protocol (HTTP)* server. However, there are lots of common traits between these bugs. For example, both these programs are network servers, where they listen over the network for incoming requests. Thus, an adversary can attempt to exploit these bugs over the network. Typically a buffer overflow bug results in crashing the server program (an example of *denial-of-service* attack), or capturing the server program's privilege (*privilege-escalation* attack). When answering the question *what happens after a successful attack,* the details of the specific attack become unimportant, and it is sufficient to know whether the server program crashed or has been completely taken over to determine the overall effects of a successful attack.

The mechanisms of attacking different client programs are surprisingly similar. For example, even though *Microsoft Word* is very different from *Adobe Acroread*, the mech-

anism an adversary would use to attack vulnerabilities in these programs is similar, like sending a malicious file that triggers a buffer overrun bug within the application to take over the process running on behalf of the target. A similar technique is employed in exploiting a vulnerability in a Web browser, an e-mail client, or an instant messenger client. To facilitate these exploits, the adversary will deliver a malicious web page to a vulnerable web browser or its components (such as Internet Explorer and Java Virtual Machine plugin of Firefox browser), or a malicious e-mail to an e-mail client (such as Pine, Mutt, Outlook Express) or send a malicious instant message to the user using a vulnerable client (such as Skype, Yahoo! Messenger, and MSN Messenger).

When a vulnerability is reported on vulnerability reporting mailing lists, typically the advisory contains information like whether the vulnerability is remotely exploitable or locally exploitable and whether the vulnerability results in a denial of service or privilege escalation. Such information is also readily available in machine-readable databases on the Internet [33]. We will show in this thesis that such meta-information is valuable in conducting a transitive closure of all bugs on a network to understand how an adversary can launch a multi-stage multi-host attack on the network.

In this section, we describe how we model different kinds of exploits. We introduce several predicates that are used in the exploit rules.

`execCode(P, H, UserPriv)` is a predicate specifying that principal `P` can execute arbitrary code with privilege `UserPriv` on host `H`.

`netAccess(P, Src, Dest, Protocol, Port)` is a derived predicate specifying that principal `P` can send packets from machine `Src` to port `Port` on machine `Dest` through protocol `Protocol`.

`networkService(Host, Program, Protocol, Port, Priv)` is a predicate specifying that a service program `Program` is running on host `Host` at privilege

level `Priv`. This program is serving requests over the network by listening on port `Port` of protocol `Protocol`.

`setuidProgram(H, Program)` is a predicate specifying that `Program` is a setuid program on host `H`. In a Unix system, when a setuid is executed, it runs at the privilege of the owner of the program[1].

Remote privilege escalation Suppose that an adversary is trying to attack a network server program. He will be able to successfully attack the program only if the following conditions are met:

- A vulnerable program is listening on the network.
- The vulnerability can be remotely exploited.
- The network configuration allows the adversary to send a malicious packet(s) to the port the program is listening on.

If the bug is a privilege escalation bug, upon on successful attack, the adversary gets hold of the network server's process, thus getting hold of the userid running the network server process. We write this formally as:

```
%% Remote network server attack
execCode(Attacker, Host, ProgramUserId) :-
        vulExists(Host, Program, remoteExploit,
                privilegeEscalation),
        networkService(Host, Program, Protocol,
```

[1] A common use of setuid programs is a situation wherein a program is run by various users, but still has to keep track of statistics or data over different runs. Examples include programs like GNOME Mahjongg (the program has to maintain a list of highest scores across different users) and `passwd` (a program to change user passwords that are stored in a common file).

```
                Port, ProgramUserId),
        netAccess(Attacker, _AttackerSource,
                Host, Protocol, Port).
```

In the formalism of Datalog, an identifier starting with _ (like `_AttackerSource`) is an anonymous variable, and it can bind to any value. So, in the above rule the predicate `netAccess(Attacker, _AttackerSource, Host, Protocol, Port)` means that it does not matter from which host the `Attacker` launches the attack.

Local privilege escalation In Unix, when a setuid program is executed, then the program executes with the privileges of the user owning the program—typically `root`. As we discussed in chapter 3, certain programs in Windows are executed in a privileged context. To attack these privileged programs, the adversary first obtains access to some privilege level on the target host, and then exploits the vulnerability in the privileged program. An adversary can attack a privileged program if:

- The privileged program is vulnerable.
- The vulnerability can be locally exploited.
- The adversary already has (shell) access to the host.

Upon successful attack, the adversary gets hold of the privileges of the privileged program. We encode this formally as:

```
%% Local attack against a privileged program
execCode(Attacker, Host, ProgramUserId) :-
        vulExists(Host, Program, localExploit, privilegeEscalation),
        privilegedProgram(Host, Program, ProgramUserId),
        execCode(Attacker, Host, _SomeUser).
```

Kernel. On many occasions, programming flaws in the kernel of the operating system resulted in both locally exploitable and remotely exploitable vulnerabilities [30, 45]. Hence, we model the operating system kernel as both a network service running as `root`, and a local privileged program. That is, the consequence of exploiting a privilege-escalation bug in kernel (either local or remote) will result in a compromise of the administrative account of the machine and hence the whole system.

Remote-exploit-client An adversary can attack a client program if the following conditions are met:

- The `Program` is vulnerable to a remote exploit.
- The `Program` is client software with privilege `Priv`.
- The `Attacker` is some principal that originates from a part of the network where malicious users may exist.

The consequence of the exploit is that the attacker can execute arbitrary code with privilege `Priv`. We encode this formally as:

```
execCode(Attacker, Host, Priv) :-
  vulExists(Host, VulID, Program),
  vulProperty(VulID, remoteExploit, privEscalation),
  clientProgram(Host, Program, Priv),
  malicious(Attacker).
```

5.1.3 File access

After exploiting a client or server program, the adversary gets hold of a certain privilege level on a host. Among other things, one of the things the adversary could try is to tamper

with critical operating system files—like the password file, kernel program, and programs in the system directory. Or the adversary could try to steal confidential data from users' directories. Thus we need to model file access semantics.

In traditional Unix, the permissions that can be placed on an object are *read*, *write* and *execute*. When a principal is granted a permission to a file object, the operating interprets the permissions as follows:

- **read** Permission to read the file. For directories, this means permission to list the contents of the directory.
- **write** permission to write to (change) the file. For directories, this means permission to create and remove files in the directory.
- **execute** permission to execute the file (run it as a program). For directories, this means permission to access files in the directory.

There are three categories of users who may have different permissions to perform any of the above operations on a file object: the file's owner, the file's group and everyone else.

`localFileProtection(Host, User, Access, FilePath)` is a predicate specifying that the `User` on machine `Host` can have specified `Access` to the file `FilePath`.

`accessFile(Principal, Host, Access, Path)` is a predicate specifying that `Principal` can access the file specified by `Path` on `Host`. `Access` can be one of `read`, `write` and `execute`.

If an attacker `Attacker` can execute code on machine `Host` as a user `User`, then he can access whatever files `User` can access. We encode this formally as:

```
accessFile(Principal, Host, Access, Path) :-
        execCode(Principal, Host, User),
        localFileProtection(Host, User, Access, Path).
```

5.1.4 Trojan horse programs

A Trojan horse is a malicious program that is disguised as legitimate software—one that masquerades as a benign program. The program may look useful or interesting (or at the very least harmless) to an unsuspecting user, but is actually harmful when executed. It could be an otherwise useful software that has been corrupted by a cracker inserting malicious code that executes while the program is used. For example, it is very common for intruders to break-in to a system and replace the *Secure Shell* executable with a malicious executable that provides the functionality of *Secure Shell*, but in addition captures the user's passwords and uploads it to a global server. A Trojan horse program may also even install a back-door program on the compromised system that allows the adversary to enter the host at a later time.

An adversary usually introduces a Trojan horse program as follows: he obtains certain privileges on a host and then replaces legitimate programs with a Trojan-horse version. When a user executes such programs, the attacker obtains the privileges of the user. A special case of Trojan-horse is where the adversary replaces an executable file used in operating system boot process. In that case, the adversary can gain complete control of the host. We model this as follows:

```
execCode(Attacker, Host, User) :-
        accessFile(Attacker, Host, write, Path),
        not privilegedProgram(Host, Path),
```

```
        localFileProtection(Host, User, exec, Path).

execCode(Attacker, Host, Owner) :-
        accessFile(Attacker, Host, write, Path),
        privilegedProgram(Host, Path),
        fileOwner(Host, Path, Owner).
```

Suppose that an adversary has the capability to overwrite a file. To determine whether overwriting this file is useful in a Trojan-horse attack, we need to determine a) if file is an executable file and b) if the target does indeed execute the file. In determining the answers to these questions, we make certain conservative approximations.

We assume that any file (in the system directories) that is marked executable is indeed an executable. There could be files that are marked executable, but they are merely used to store data (like statistics)—so overwriting these files does not result in a Trojan-horse attack. For example the file `/etc/ppp/options` file in a Fedora Core 4 machine is marked executable, but it is merely a configuration file and is not directly executed. It is very hard to automatically understand when a file is indeed an executable and the strategy of checking if `execute` permission is granted is a good heuristic. In practice, we found that very few data files are mistakenly marked as executable files. Thus, it is reasonable to assume that any file (in the system directories) that is marked executable is indeed executable.

If an adversary can tamper with or overwrite an executable program, a successful Trojan-horse attack results only after the target actually executes the program. It is really hard to determine the circumstances under which an executable file is indeed executed. For example, it is hard to answer questions like *"who are using the grep program"* and *"when is the executable /usr/bin/zipinfo used"*. We assume that a file that is marked

executable is executed by the owner of the file. If an executable file is owned by an administrator, then we assume that all users *will* use the file. This is conservative yet pragmatic given that our approach is to treat programs as black boxes.

5.1.5 Networked File Systems

A networked file system allows sharing a file system between many computers, so that one could easily access files from all of them. In cluster environments where there a large number of identical machines, it is a hassle for the administrator to maintain each machine. The solution adopted is that the administrator maintains a single host and then lets the changes propagate to other hosts automatically using networked file systems.

After getting certain access on a host, one way an adversary can propagate the attack to other hosts is to use the networked file systems. The adversary corrupts the file on the client share, and the file system copies the changes back on to the server share. From the server share, the corrupted file can automatically propagate to other clients mounting the share. There a large number of networked file systems, like *Andrew File System (AFS)*, *Network File System (NFS)* and *Server Message Block (SMB, Samba)* . Of these, we show here how to model the *Network File System (NFS)*. We hypothesize that modeling other networked file systems should be similar.

NFS protocol is based on *Remote Procedure Call (RPC)*. NFS is assigned an RPC protocol number of 100003. `nfsExportInfo(Server, Path, Access, Client)` `nfsMounted(Server, ServerPath, Access, Client, ClientPath)` are predicates that are generated by looking at the NFS configuration file on the client side. However, if an administrative account is compromised on the client side, then the attacker can mount any files. We encode this as:

```
nfsMounted(Anything) :-
        execCode(P, Client, root).
```

If files are mounted on the client side using the NFS protocol, and if the adversary can access files on the client side, then the adversary can access the files on the server side. We encode this as:

```
accessFile(P, Server, Access, SrvPath) :-
        nfsExportInfo(Server, Path, Access, Client),
        nfsMounted(Server, SrvPath, Access, Client, ClntPath),
        netAccess(P, Client, Server, rpc, 100003).
        accessFile(P, Client, Access, ClntPath).
```

5.2 Modeling the network

An attacker can propagate a multi-stage multi-host by sending malicious packets over the network to compromise vulnerable server programs. The ability of the attacker to send malicious packets depends on how the network infrastructure is configured. It is very common to use *network firewalls* to allow or block certain network packets, depending on various characteristics. Thus, the system administrator can achieve certain goals like *block all Internet access from this class of hosts*, *block all programs that use UDP traffic*, *do not allow any traffic on port 23 in TCP protocol*.

Packet flow is controlled by switches, routers and firewalls. Modeling the elements of the network infrastructure is important in determining the options an adversary has in attacking a network. The problem of modeling network infrastructure to determine what kind of activities are allowed is well studied [6, 4]. We run network analysis tools which produce host to host reachability information. We abstract this information as a set of

host access control lists (hacl). We feed the hacl into our analysis. A host access control list specifies all accesses between hosts that are allowed by the network. It consists of a collection of entries of the following form:

```
hacl(Source, Destination, Protocol, DestPort).
```

One can use these abstracted entries to determine the network access an adversary has as follows:

```
netAccess(P, H2, Protocol, Port) :-
    execCode(P, H1, Priv),
    hacl(H1, H2, Protocol, Port).
```

If a principal `P` has access to machine `H1` under some privilege and the network allows `H1` to access `H2` through `Protocol` and `Port`, then the principal can access host `H2` through the protocol and port. This allows for reasoning about multi-host attacks, where an attacker first gains access on one machine inside a network and launches an attack from there.

5.3 Policy specification

The analysis we described earlier can be used to compute which user can access what resource and which user can obtain what privileges. An administrator can access all the resources of his domain and can obtain the privileges associated with his any user in his domain. This is normal behavior. On the other hand, a situation where a untrusted user can access confidential files or can obtain unwanted privileges is a real problem. We need a mechanism to specify which accesses are allowed. In our framework, the system administrator specifies allowed behavior through a security policy.

5.3.1 Policy specification

The security policy specifies which principal can access what data. Each principal and data is given a symbolic name, which is mapped to a concrete entity by the binding information discussed in section 5.3.2. Each policy statement is of the form

`allow(Principal, Access, Data).`

The arguments can be either constants or variables (variables start with a capital letter and can match any constant). Following is an example policy:

```
allow(Everyone, read, webPages).
allow(user, Access, projectPlan).
allow(sysAdmin, Access, Data).
```

The policy says anybody can read `webPages`, `user` can have arbitrary access to `projectPlan`. And `sysAdmin` can have arbitrary access to arbitrary data. Anything not explicitly allowed is prohibited.

The policy language presented in this section is quite simple and easy to make right. However, the MulVAL reasoning system can handle more complex policies as well (see section 5.6).

5.3.2 Binding information

Principal binding maps a principal symbol to its user accounts on network hosts. For example:

```
hasAccount(user, projectPC, userAccount).
hasAccount(sysAdmin, webServer, root).
```

Data binding maps a data symbol to a path on a machine. For example:

```
dataBind(projectPlan, workstation, '/home').
dataBind(webPages, webServer, '/www').
```

The binding information is provided manually.

5.4 Analysis Algorithm

The analysis algorithm is divided into two phases: *attack simulation* and *policy checking*. In the attack simulation phase, all possible data accesses that can result from multistage, multi-host attacks are derived. This is achieved by the following Datalog program.

```
access(P, Access, Data) :-
  dataBind(Data, H, Path),
  accessFile(P, H, Access, Path).
```

That is, if `Data` is stored on machine `H` under path `Path`, and principal `P` can access files under the path, then `P` can access `Data`. The attack simulation happens in the derivation of `accessFile`, which involves the Datalog interaction rules and data tuple inputs from various components of MulVAL. For a Datalog program, there are at most a polynomial number of facts that can be derived. We execute our Datalog programs in off-the-shelf Datalog engine XSB [38]. The engine guarantees (through its tabling mechanism) that each fact is computed only once. Hence, the attack simulation phase is polynomial.

In the policy checking phase, the data access tuples output from the attack simulation phase are compared with the given security policy. If an access is not allowed by the policy, a violation is detected. The following Prolog program performs policy checking.

```
policyViolation(P, Access, Data) :-
  access(P, Access, Data),
  not allow(P, Access, Data).
```

This is not a pure Datalog program because it uses negation. But the use of negation in this program has a well-founded semantics [21]. The complexity of a Datalog program with well-founded negation is polynomial in the size of input [17]. In practice the policy checking algorithm runs very efficiently in the XSB environment (see section 7.5).

5.5 Hypothetical analysis

One important usage of vulnerability reasoning tools is to conduct "what if" analysis. For example, the administrator would like to ask *"Will my network still be secure if two CERT advisories arrive tomorrow?"*. After all, an important purpose of using firewalls is to guard against *potential* threats. Even there is no known vulnerability in the network today, one might be discovered tomorrow. Analysis that can reveal weaknesses in the network under hypothetical circumstances is useful in improving security. Performing this kind of hypothetical analysis is easy in our framework. We introduce a predicate `bugHyp` to represent hypothetical software vulnerabilities. For example, following is a hypothetical bug in the web service program `httpd` on host `webServer`.

```
bugHyp(webServer, httpd,
               remoteExploit, privEscalation).
```

The fake bugs are then introduced into the reasoning process.

```
vulExists(Host, VulID, Prog) :-
```

```
   bugHyp(Host, Prog, Range, Consequence).

vulProperty(VulID, Range, Consequence) :-
   bugHyp(Host, Prog, Range, Consequence).
```

The following Prolog program will determine whether a policy violation will happen with two arbitrary hypothetical bugs.

```
checktwo(P, Acc, Data, Prog1, Prog2) :-
  program(Prog1),
  program(Prog2),
  Prog1 @< Prog2,
  cleanState,
  assert(bugHyp(H1, Prog1, Range1, Conseq1)),
  assert(bugHyp(H2, Prog2, Range2, Conseq2)),
  policyViolation(P, Acc, Data).
```

The two `assert` statements introduce dynamic clauses about hypothetical bugs in two programs (Prolog backtracking will cycle through all possible combination of two programs.). The policy check is conducted with the existence of the dynamic clauses. If no policy violation is found, the execution will back track and another two hypothetical bugs (in two different programs) will be tried. `@<` is the term comparison operator in Prolog. It ensures a combination of two programs is tried only once. If there exist two programs whose hypothetical bugs will break the security policy of the network, the violation will be reported by `checktwo`. Otherwise the network can withstand two hypothetical bugs.

5.6 Discussion

Coverage. We model privilege escalation attacks and denial-of-service attacks. We do not model vulnerabilities whose exploit consequence is confidentiality loss or integrity loss. The ICAT database does not provide precise information as to what confidential information may be leaked to an attacker and what information on the system may be modified by an attacker. ICAT statistics show that 84% of vulnerabilities are labeled with only privilege escalation or denial of service, the two kinds of exploits modeled in MulVAL. It seems in reality privilege-escalation bugs are the most common target for exploit in a multistage attack.

More complex policies. The two-phase separation in the MulVAL algorithm allows us to use richer policy languages than Datalog without affecting the complexity of the attack simulation phase. The MulVAL reasoning system supports general Prolog as the policy language. Should one need even richer policy specification, the attack simulation can still be performed efficiently and the resulting data access tuples can be sent to a policy resolver, which can handle the richer policy specification efficiently.

No policy? Because the attack simulation is *not* guided by or dependent on the security policy, it is possible to use MulVAL without a security policy; the system administrator may find useful the raw report of who can access what. However, the policy is useful in filtering undesirable accesses from harmless accesses.

Hypothetical analysis. Is it really possible that a real network can survive two general hypothetical bugs? Networks are large and complex. A large number of security bugs are reported in software and hence in any large network system administrators have to

manage a large number of software bugs. For example, as we describe in section 7.2.1, a standard vulnerability scanner found several bugs in a professionally managed network used by hundreds of users. The system administrators explained that they were aware of the security bugs, but hypothesized that exploiting the bug is difficult and that the vulnerable program is accessible only from a couple of trusted hosts on the internal network. Hence, fixing these security bugs was not the most important issue they had to attend to. Sometimes, removing a security bug network from a network results in disruption of business processes and the immediate risk of exploitation is low. In these scenarios, an administrator may decide that the cost of fixing the security bug is not worth the cost and may instead of deploy a intrusion detection system to monitor the vulnerable software. Administrators also follow a "defense in depth" strategy where they prepare for circumstances where a particular software may be compromised. In such situations, it is very useful for the administrator to have a tool that can answer questions such as "what happens if a particular host is compromised".

Chapter 6

Vulnerability scanning

In this chapter I describe the state of the art in vulnerability scanning and how improvements can be made. In section 6.1, I describe the evolution of vulnerability scanning technology. In section 6.2, I describe a state-of-the-art framework developed by the Mitre Corporation to concisely specify the properties of a bug. This framework can be used to quickly identify the existence of a bug on a network. In section 6.3, I show how one specifies a vulnerability in the framework. In section 6.4, I outline the drawbacks of current vulnerability scanners, even state-of-the-art ones. In that section, I outline the design of a next generation scanner to addresses these shortcomings.

6.1 Evolution of vulnerability scanning technology

An important component of a network's security management is to understand the current known vulnerabilities in software installed on the network and determine whether any of the hosts in the administrator's domain are vulnerable. To obtain information about the existence and exploit details of current known vulnerabilities, an administrator subscribes

to vulnerability discussion and information dissemination mailing lists such as BugTraq, US-CERT's Cyber Security Bulletins, and Full Disclosure mailing list. In figure 6.1, we show an advisory in Ethereal software that was discussed on many discussion lists on 24 April 2003. This advisory details information such as the affected versions of the program, the affected versions of the operating system distribution that deliver the vulnerable software, how the adversary can exploit the vulnerability, and how the administrator can get the updated versions of the software that fix this vulnerability.

The advisory mentions the details of the vulnerability in natural-language format and not in a format with formal semantics. The main drawback of this approach is that the process of determining if a vulnerability exists on a given host requires manual intervention, which is a slow and error-prone process. On large networks, the process of manual discovery of vulnerabilities is infeasible because of the large number of machines and limited number of system administrators. There is a need to consolidate management of vulnerabilities, from discovery to real-time reporting to managed security monitoring reports. This consolidation can be classified into the following stages:

- **Manual:** This is the most primitive phase, where an administrator reads the vulnerability advisory and manually checks information such as the version of the program and the operating system, the configuration options for the program (like whether module `mod_perl` of Apache web server is enabled). Examples include opening the vulnerable program and clicking the "About" tab in the case of applications with graphical interfaces, checking the values of registry keys through `Regedit`, and using `winver` command in Windows to open a graphical session listing various operating system parameters.

```
---------------------------------------------------------------------
Red Hat Security Advisory

Synopsis: Updated ethereal packages fix security vulnerabilities
Advisory ID: RHSA-2003:076-01
Issue date: 2003-04-23
Updated on: 2003-04-23
Product: Red Hat Linux
Keywords:
Cross references:
Obsoletes: RHSA-2002:290
CVE Names: CAN-2003-0081 CAN-2003-0159
---------------------------------------------------------------------

1. Topic:

Updated ethereal packages are now available which fix a format
string bug and a heap-based buffer overflow.

2. Relevant releases/architectures:

Red Hat Linux 7.2 - i386, ia64
Red Hat Linux 7.3 - i386
Red Hat Linux 8.0 - i386
Red Hat Linux 9 - i386

3. Problem description:

Ethereal is a package designed for monitoring network traffic on
your system. Ethereal 0.9.9 and earlier allows remote attackers
to cause a denial of service (crash) and possibly execute
arbitrary code via carefully crafted SOCKS packets. Additionally,
a heap-based buffer overflow in the NTLMSSP code for Ethereal
0.9.9 and earlier allows remote attackers to cause a denial of
service and possibly execute arbitrary code. Users of Ethereal
should update to the erratum packages containing Ethereal version
0.9.11 which are not vulnerable to these issues.

4. Solution:

Before applying this update, make sure all previously released
errata relevant to your system have been applied.
...
```


Figure 6.1: Sample vulnerability advisory on the Full Disclosure mailing list.

- **Scripting** Some of the repetitive tasks are scripted. Small scripts are written to test for the existence of specific bugs. These scripts can be run automatically with little human interference. To detect a new bug, one has to change the program that tests for the vulnerability.

- **Data-driven vulnerability scanning:** The main drawback of the previous stage is that any time a new vulnerability definition is out, the program that recognizes the bug has to be changed. This ad hoc programming can introduce errors. A better solution is to define a schema to describe bugs and then use a program that understands the schema to automatically recognize bugs. The advantage of this approach is that when a new vulnerability is reported, one just needs to write the specification for the vulnerability. It is much easier to get the specification right than to correctly write a script to identify the existence of the vulnerability.

Currently, the Open Vulnerability Assessment Language (OVAL) represents the most advanced stage of this evolution process. The schema has been developed by the Mitre Corporation and Mitre also released a reference implementation; other vendors have also released OVAL-compatible scanners. OVAL compatible vulnerability definitions are available for download at the website `http://oval.mitre.org` and from other vendors. We discuss in detail the OVAL schema and its reference implementation because it represents the state of the art in vulnerability scanning. We then discuss the limitations of current systems and how we designed better scanners.

6.2 Open Vulnerability Assessment Language

In section 2.4.1 of this thesis, we briefly introduced the idea of formal representation of security advisories and automatic recognition of security vulnerabilities. The OVAL

schema is a standard schema that one can use the specify the security vulnerabilities [49]. An OVAL vulnerability test is defined as a boolean combination of one or more elementary tests. An elementary test is used to query properties such as the existence of a file, the permissions on a file, the operating system version, the processes running on a host, the ports that are bound to network server programs and the version information of software installed on the host. An example OVAL vulnerability definition, when written in pseudo-code would read:

```
This host is vulnerable if
      -- The host runs Red Hat Linux 6.2 operating system
   and
      -- Apache 1.3.4 is installed on the host
   and
      -- Apache program listening on port 80
   and
      -- Config file /etc/apache.conf is world-writeable.
    or
      -- Config file /etc/httpd.conf is world-writeable.
```

6.2.1 Linux tests

The current definitions for the vulnerabilities and definition intrepreters on the Linux platform use the following tests: *file test*, *permission test*, *uname test*, *process test*, *network server test*, *rpminfo test* and the *rpmversion test*. We now briefly describe the functionality of each of these tests.

Red Hat Package Manager An important problem system administrators face in systems management is conflict management between several software products. Before installing a new software, an administrator will have to (manually) check that its prerequisite software is already installed. Installation of new software might replace a library file used by another software, which might render some of the already existing software unusable. In the worst case, the machine might be rendered unbootable. Operating system vendors provide mechanisms to automatically track the dependencies between various software and check for conflicts before software is installed. The *Red Hat Package Manager* is an open source application package management system that is used to maintain package specific information (such as version information, required libraries, installation scripts and files used by the package) and a database of all packages installed on a host, a list of files on the host and the programs that use the file. It is widely used by various distribution vendors for their Linux distributions. Each application package is supplied to the Red Hat Package Manager as a Red Hat Package Manager (RPM) package. To determine properties of an application installed on a host (such as version information, files required), one needs to just consult the RPM database.

File test. A file test is used to test the existence of a file. After determining the existence of the file, this test can be used to retrieve properties of the file such as owner, group, last time the file was accessed or accessed, the time the file was created and a checksum. OVAL schema uses `rft` as the prefix to identify a file test.

Permission test. A permission test is used to test the permission bits of a file. One can test whether a file is readable, writeable, or executable by either the owner or the group or the world. One can also test whether the setuid or setgid bits of a fie are turned on—these

bits imply that upon execution the program in the file runs on behalf of the owner of the file. OVAL schema uses `ret` as the prefix to identify this test.

Uname test. A uname test is used to obtain properties such as the machine hardware name (like i686), the host name, the operating system name (like Linux), the operating system build version (like "#1 Wed Aug 25 13:34:40 UTC 2004"), the operating system version (like 2.6.11-1.1369_FC4) and the processor type (like i686). OVAL schema uses `rut` as the prefix to identify this test.

Process test. A process test is used to obtain properties such as the command or program name to check, the amount of CPU time the process has consumed, the process ID of the process, the parent process's process ID, the scheduling priority of the process (adjusted using the `nice` command), the scheduling class, the start time of the process, the terminal (tty) on which the process was started and the user owning the process. OVAL schema uses `rct` as the prefix to identify this test.

Network server test. This test is used to check if a program is listening on the network, either for a new connection or as part of an ongoing connection. One can retrieve the name of the communication program, the internet protocol address of the network interface on which the program listens, the local port, the remote internet protocol address, the remote port, the transport layer protocol, the process identity of the process and the user account which owns the network server process. OVAL schema uses `rlt` as the prefix to identify this test.

Rpminfo test. This test checks the RPM header for a given RPM package. It is used to retrieve the architecture the package was packaged for, the epoch number for the RPM,

the release number for the RPM, and the version number for the software built in the RPM. OVAL schema uses `rrt` as the prefix to identify this test.

Rpmversion test. This test checks the installed RPM against a given epoch, version and release number to determine if the installed RPM is an earlier or later version compared to the affected version. A majority of the tests in the OVAL vulnerability database maintained by the Mitre Corporation use this test to determine the version of a program. One supplies the information for the first RPM that does not have the given security flaw and the interpreter checks if the software installed is vulnerable. OVAL schema uses `rvt` as the prefix to identify this test.

The above tests are supported by the standard reference implementation. In addition to these tests, there are other tests that are supported by the schema, but not by the standard reference implementation. These tests are used to obtain information such as the network interfaces on the host, information regarding the user's password, password aging and lockout.

6.2.2 Windows tests

The current definitions for the vulnerabilities and definition intrepreters on the Windows platform use the following tests: *registry test*, *file test* and the *metabase test*. We now briefly describe the functionality of each of these tests.

Registry test. The Windows registry test specifies a particular registry key (or keys) to test. One can test if the value stored in a registry key matches a regular expression. OVAL schema uses `wrt` as the prefix to identify this test.

File test. This test is used to test the file metadata information. One can retrieve the owner, the time of access, the creation time, time of last modification, the checksum of the file, type of the file (one of directory, standard file, and named pipe), The meta data associated with a file also contains the "version" of the file. One can use the file test to retrieve the version of the file. OVAL schema uses `wft` as the prefix to identify this test.

Metabase test. The Windows metabase is a database that is very similar to the registry. A metabase is used to store information for the `Internet Information Services` program. One can use this test to retrieve information stored in the specified metabase keys. OVAL schema uses `wmt` as the prefix to identify this test.

The standard reference implementation supports the above tests. In addition, the Windows schema supports a large number of other tests that are not implemented in the reference implementation. These tests are used to obtain information such as the account privileges of an account, the configuration settings available under active directory, the audit policy of the system (such as writing to the security log when a user logs on or off, when a user changes password and when a user executes a file), the audit policy on a given file or registry key (such as writing to the security log when a particular user opens a file for a read operation), the security descriptor that determines which principals can access a file or registry key, the group memberships of a given group, the network interfaces of the system, the account lockout policy, the password policy, the network ports open on a host, the properties of processes on the system, and the information regarding the file system.

6.3 Example of formal vulnerability specification

We now show how to formally encode the vulnerability we discussed above. The vulnerability advisory released by the vendor is shown in figure 6.1. The vulnerability affects Ethereal 0.9.9 and earlier on operating systems Red Hat Linux 7.2 through 9 on `i386` architecture and Red Hat Linux 7.2 on ia64 architecture. Thus we need to write an OVAL test that tests the operating system, architecture and program version.

Testing for Operating System. The procedure to test the version of an operating system in a Red Hat Linux system is to check the version of a `redhat-release` RPM package. We show how to test for Red Hat Linux 9 system; testing for other versions is similar. The test is:

```
<rpminfo_test id="rrt-201" comment="Red Hat 9 is installed">
        <name operator="equals">redhat-release</name>
        <version datatype="int" operator="equals">9</version>
</rpminfo_test>
```

Testing for architecture. We show how to test for `i386` architecture. The test for `ia64` is similar.

```
<uname_test id="rut-201" comment="ix86 architecture">
        <machine_class operator="pattern match">^i.*86
        </machine_class>
</uname_test>
```

Testing program version The test to check for Ethereal versions 0.9.9 and earlier is:

```
<rpmversioncompare_test id="rvt-206">
        <name operator="equals">ethereal</name>
        <tested_version operator="equals">
                0.9.11
        </tested_version>
        <installed_version operator="equals">
               earlier
        </installed_version>
</rpmversioncompare_test>
```

Complete vulnerability definition. In figure 6.2, we show a complete vulnerability definition that uses the above tests as elementary tests. This definition is listed in the OVAL vulnerability database maintained by the Mitre Corporation, with the reference identifier being `OVAL54`. Besides the criteria needed to recognize a vulnerability, the complete definitions also incorporates other useful information such as the definition author, the dates, description.

6.4 Next generation scanners

The current state-of-the-art scanner suffers from two weaknesses: insufficient functionality and large trusted base. We now explain why these issues are problems and how we designed a better scanner.

```
<definition id="OVAL54" class="vulnerability">
        <affected family="redhat">
                <redhat:platform>Red Hat Linux 9</redhat:platform>
                <product>Ethereal</product>
        </affected>
        <contributors>
                <submitter organization="The MITRE Corporation">
                        Jay Beale
                </submitter>
        </contributors>
        <cveid status="CAN">2003-0081</cveid>
        <dates>
                <created date="2003-08-17"/>
                <modified date="2004-05-05">
                      Corrected syntax errors in the sql
                      verion of the definition.
                </modified>
                <status_change date="2004-03-25">INTERIM
                </status_change>
                <status_change date="2004-05-25">ACCEPTED
                </status_change>
        </dates>
        <description>
          Format string vulnerability in packet-socks.c
          of the SOCKS dissector for Ethereal 0.8.7 through 0.9.9
          allows remote attackers to execute arbitrary code via
          SOCKS packets containing format string specifiers
        </description>
        <status>ACCEPTED</status>
        <version>1</version>
        <criteria>
              <software operation="AND">
                        <criterion test_ref="rrt-201"
                              comment="Red Hat 9 is installed" />
                        <criterion test_ref="rut-201"
                              comment="ix86 architecture" />
                        <criterion test_ref="rvt-206"
                              comment="ethereal version
                                      0.9.9 and earlier/>
              </software>
        </criteria>
</definition>
```


Figure 6.2: Sample Vulnerability Definition in OVAL database (OVAL54).

6.4.1 Drawbacks of current vulnerability scanning technologies

Insufficient functionality. We show in chapter 7 how our tool found serious vulnerabilities in the configuration of software from major vendors for the Windows platform. These vulnerabilities were in the security configuration of certain Windows services and registry keys installed by various vendors. The current vulnerability scanners do not support scanning the security configuration of service objects in Windows. An OVAL scanner is meant to recognize already existing vulnerabilities. One has to supply the vulnerability definition to recognize the existence of a vulnerability. Thus an OVAL scanner can never recognize the existence of an unknown bug. We wrote a sophisticated scanner to query the configuration of services and other operating system objects. We then used the tool to find previously unknown vulnerabilities.

Scanners are too heavy. Another drawback of current scanners is that the collection of configuration data from the host is not separated from the analysis of the configuration data. The collection phase of the vulnerability scanner needs to collect certain privileged information and hence needs to run from an administrative account. The analysis phase involves understanding the vulnerability definition, consulting the configuration data from the collection phase and determining if the vulnerability does indeed exist on the host. Because collection of data is not separated from configuration analysis, both the collection and analysis phases need to run with administrative privileges. The net result it that the trusted computing base of the vulnerability scanner is much larger than it needs to be; it is well understood that this is a bad idea.

6.4.2 Next generation scanner

We built a new scanner that extends the functionality of an OVAL scanner to scan the configuration settings of Windows services and registry key's security context. Our scanner scans the entire registry and identifies registry keys whose data contain the name or path of an executable file or library. It is very common in Windows to use a registry key to store the file or library to invoke upon certain conditions being met. For example, when a user logs onto a host, the system should determine what is the first program that runs on behalf of the user. The path to this program is stored in the key `HKLM\SOFTWARE\Microsoft\WindowsNT\Current Version\Winlogon\Userinit`. If a sensitive key can be overwritten by the adversary, the adversary can make the key point to his executable content. The adversary then waits for the system to execute his content. Our tool identifies sensitive registry keys and for each sensitive registry key, our tool investigates the security descriptor to see if the key's contents can be overwritten by the adversary. Our tool enumerates the services on a Windows host and investigates the security descriptor on each service if the service presents the adversary with an opportunity to attack the system.

We modified the OVAL (Linux) scanner so that it works in two phases. The first phase needs administrative privileges and collects all the configuration data into a database. The second phase does the analysis of the configuration data and the vulnerability definition and determines if the vulnerability specified by the definition exists on the host. In this design, only the first phase needs administrative privileges, the second phase can run under a nonadministrative account. In figure 6.4.2, we show how the better scanner reduces the trusted computing base and the size of the configuration snapshot.

Another advantage of separating the configuration collection from analysis is that configuration collection can be run before information about a vulnerability is available.

Tests	Lines of code	Size of configuration snapshot
File and file permission test	250	20 MB
Process test	150	1 MB
Network server test	1	1 KB
Uname test	50	1 KB
RPM tests	2000	10 MB
Total	2451	31MB

Figure 6.3: We designed a two-phase scanner that separates configuration information collection from analysis. In contrast, the OVAL scanner has a trusted computing base of 17354 lines of code.

If configuration collection is run after a vulnerability is publicly known, there is the possibility that the adversary has already compromised the machine. If the adversary has already compromised the host, then he may be able to hijack the OVAL scanner, tricking the scanner to believe that the vulnerability does not exist. We suggest that the configuration collector be run periodically or each time a change is made to the host configuration and that the results be stored in a centralized server. When a vulnerability advisory is released, one check if the host is vulnerable by running the analyzer on the centralized server—thus not relying on the information provided by the host after a vulnerability is publicly known.

If one maintains configuration information on a centralized and trust-worthy server, then it is easy to maintain a historical database of the configuration of the host. When it is suspected that a host has been compromised, a historical database of host configuration can be used to perform an analysis of what the adversary could have done after compromising a host.

Chapter 7

Practical experience

In this chapter, we describe our experience with deploying our tool. Our tool found two classes of security attacks on real networks: attacks using configuration vulnerabilities and attacks leveraging interaction of multiple vulnerabilities. We first describe how our tool found previously unknown vulnerabilities in the configuration of a single host.

7.1 Single host configuration vulnerabilities

Our tool to analyze a Windows host comprises two phases: configuration collection and analysis. The configuration collector queries the security descriptors of all files, registry keys and services in the host. There are a large number of objects that are owned by the Administrator's account and whose security descriptor configuration prevents anyone other than an Administrator from modifying it. Such objects would not aid the adversary in a privilege escalation attack and can be safely ignored so that our analysis phase is efficient. Our configuration collector filters such objects and passes the configuration of the rest of the objects for closer analysis. The configuration collector can run without

the administrative privileges. The data from the configuration phase is fed to the analysis engine, which could potentially run on a different host. The analysis phase uses the formal model of the operating system encoded in Datalog to reason about the configuration data. The analysis phase identifies privilege escalation attacks against the host as described in chapter 4.

We used our tool to see how software from various vendors is configured in the default installation. Figure 7.2 shows how unprivileged users on a Windows XP host can obtain administrator privileges through several paths. Figure 7.1 shows how software from several vendors is configured in hosts in a professionally managed network. These results indicate that unprivileged users can gain administrator privileges through several paths. We hypothesize that even professional software developers and professional system administrators have a limited understanding of the semantics of the operating system. We suggest that developers and administrators use tools like ours to examine how software is configured. We found three classes of bugs: file system misconfigurations, registry misconfigurations and service misconfigurations.

7.1.1 Service misconfigurations

Several vendors poorly apply the Windows access control model to their services; a common mistake is to assign the SERVICE_CHANGE_CONFIG permission indiscriminately to services. The Windows XP documentation states, "...because this grants the caller the right to change the executable file that the system runs, it should be granted only to administrators" [13]. But that warning fails to explain clearly that permission to configure a service allows both setting the executable *and* selecting the account under which the service runs, e.g., change the "run-as" account to `Local System` [12, 28]. From `Local`

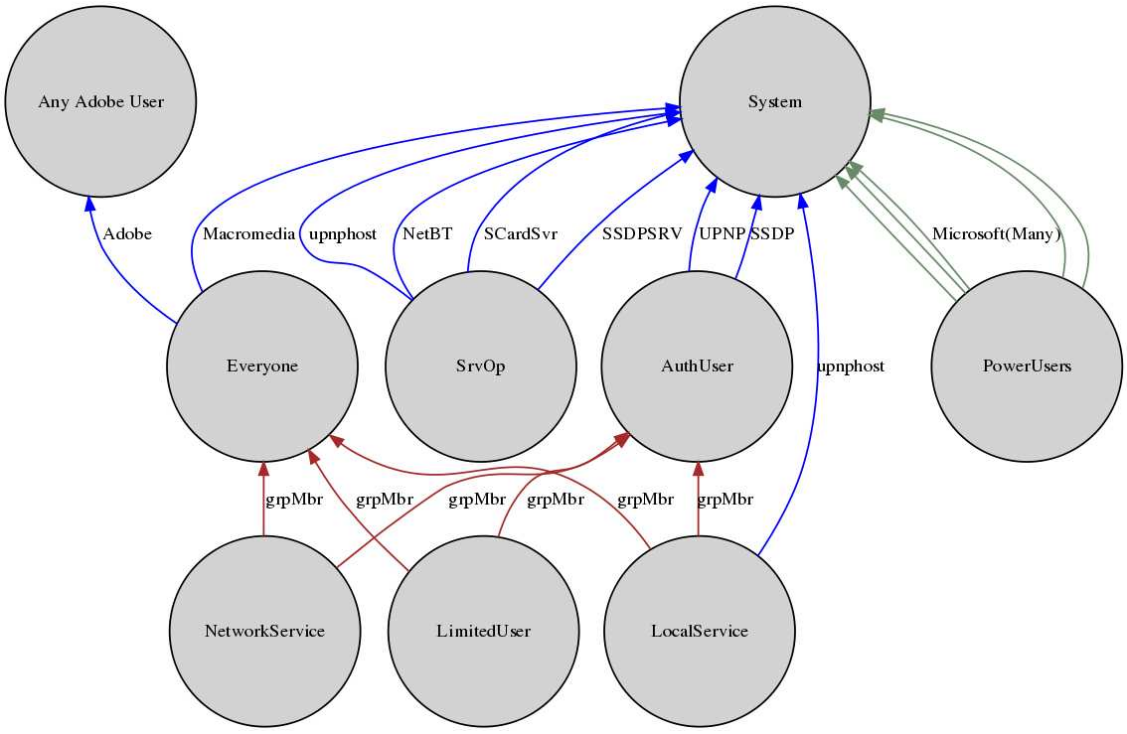

Figure 7.1: Privilege escalations in a single host of a network managed by full-time professionals. LimitedUser is an unprivileged user, NetworkService and LocalService are low-privileged accounts used to run some operating system programs. Everyone, SrvOp, AuthUser and PowerUsers are groups. The arcs labeled grpMbr show that the user is a member of the group. All other arcs show privilege escalations. There are about thirty escalation paths from PowerUsers to System. It can been seen from the graph that software running on even professionally managed hosts has serious problems in using the operating system's access control model resulting in serious privilege-escalation vulnerabilities.

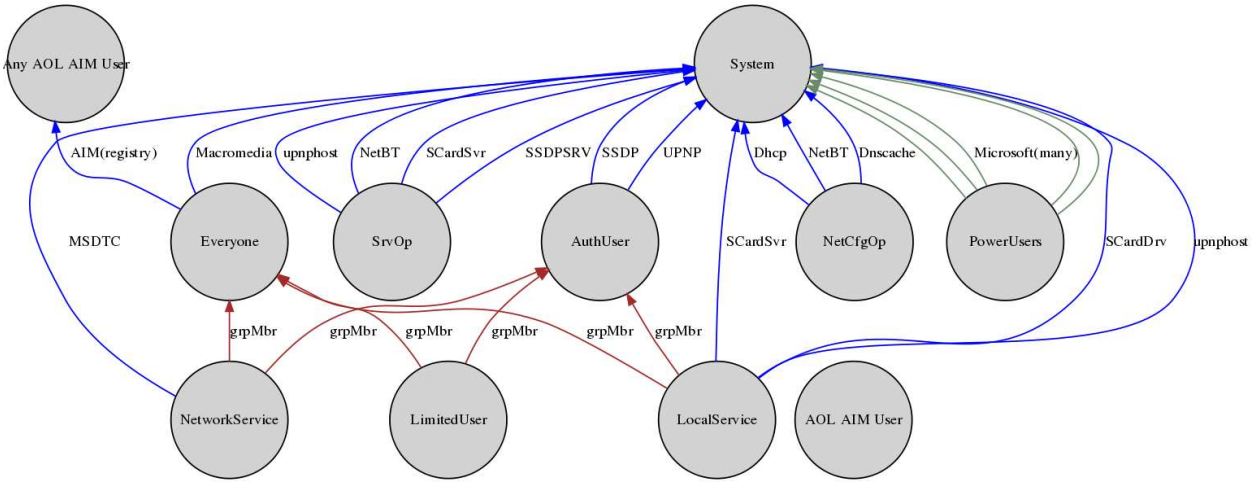

Figure 7.2: Privilege escalations found in a default configuration of WindowsXP, prior to Service Pack 2. LimitedUser is an unprivileged user, NetworkService and LocalService are low-privileged accounts used to run some operating system programs. Everyone, SrvOp, AuthUser, NetCfgOp, and PowerUsers are groups. The arcs labeled grpMbr show that the user is a member of the group. All other arcs show privilege escalations. There are about thirty escalation paths from PowerUsers to System. It can be seen from the graph that the Microsoft Windows' security model is complicated that even professional software developers at Microsoft have a difficult time in using the model correctly. In addition, we show vulnerabilities in AOL messenger software.

`System`, all things are possible (including installing password sniffers to launch further attacks in the guise of any ordinary user).

We now describe specific bugs we discovered; each example is marked with a bullet • and corresponds to one or more labeled arcs in the graph, marked in **boldface**.

• In the default configurations of Windows XP, the *SSDP Discovery Service* (**SSDP** in the graph) and the *Universal Plug and Play Device Host service* (**uPnP** in the graph) granted "permission to configure the service" to the `Authenticated Users` group. A normal unprivileged user is a part of the `Authenticated Users` group and hence a normal user can configure the executable and the account under which these services run. Then, the adversary needs to make the service reload the new configuration. He needs to wait for the service to be restarted (he could, for example, force the system

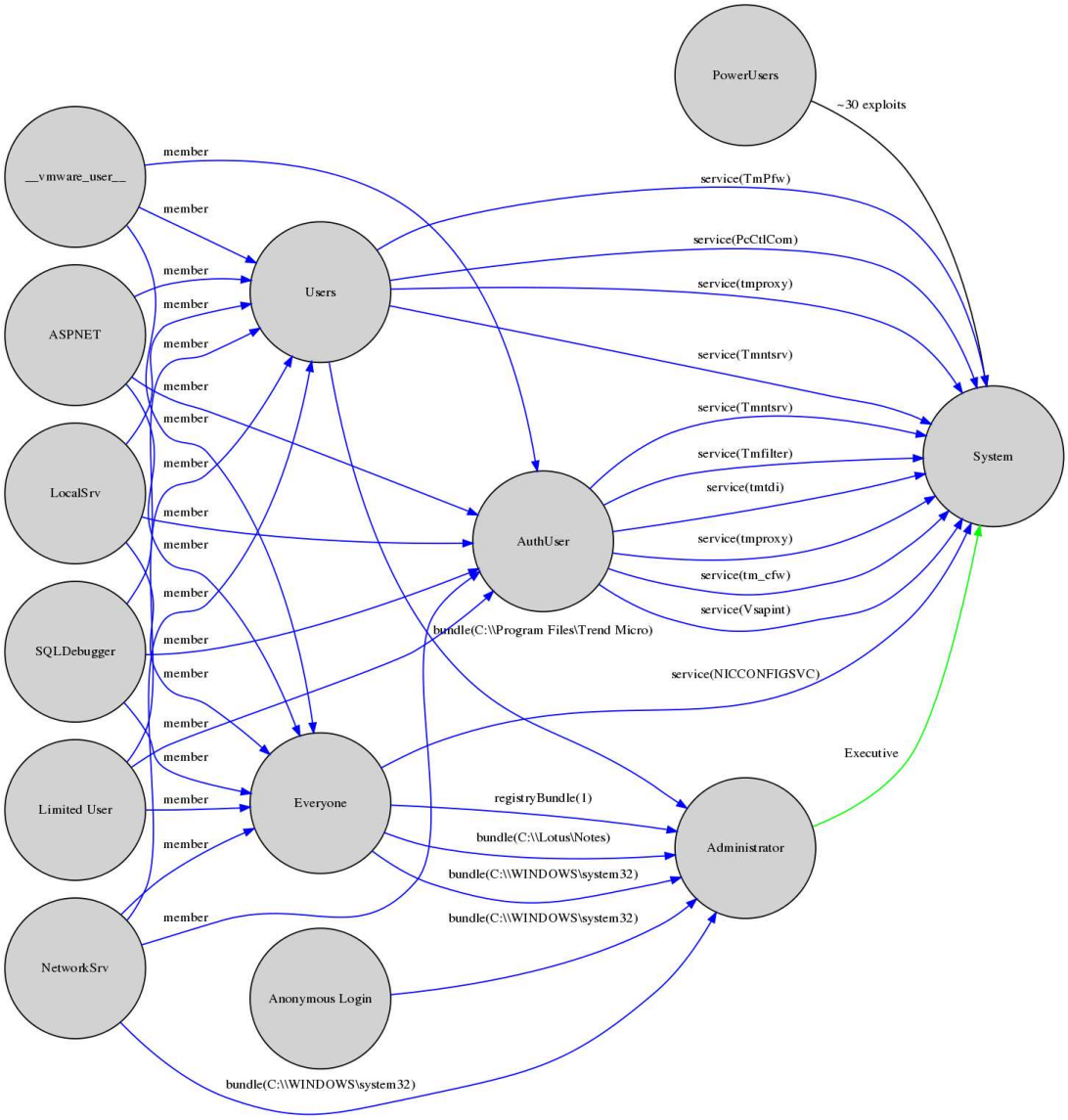

Figure 7.3: Privilege escalations found in a machine running just Lotus Notes, PC-Cillin antivirus, and VMWare. The vulnerable software is Lotus Notes and PC-Cillin antivirus. The circles in the extreme left column represent various users installed by the operating system or other software. The dark nodes are represent the administrative accounts. As one can see the privilege-escalation graph is dense. The escalations from the `LimitedUser` account are the most serious among the many shown on the graph. In the limited space available on this page, it is a significant challenge to make the figure show complete details and yet be readable. Sometimes, it is difficult to determine which software installed a particular a registry key or file or service.

administrator to reboot the machine by consuming too many resources so that the system is too slow to respond). We also noticed that usually when a principal is granted the SERVICE_CHANGE_CONFIG permission, he is also granted SERVICE_STOP and SERVICE_START permissions. One could use the Service Controller (`sc.exe`) to trivially reload the service in the new configuration as follows:

```
$sc config weakService binPath=c:\attack.exe obj=".\LocalSystem" \
            password=""
$sc stop weakService
$sc start weakService
```

Via the same SERVICE_CHANGE_CONFIG mechanism, the following (Windows XP Professional) access-control decisions give paths from `Local Service`, `Network Service`, `Network Configuration Operators`, and `Server Operators` to `Local System`: • The `Local Service` account has permission to configure the *Universal Plug and Play* service (**uPnP**), *Smart Card Services* (**SCardSvr**) and the *Smart Card Helper Service* (**SCardDrv**). • The `Network Service` account has permission to configure the *Microsoft Distributed Transaction Coordinator* service (**MSDTC**). • The `Server Operators` group has permission to configure **uPnP**, *Simple Service Discovery Protocol* (**SSDP**), *NetBios over TCP/IP* (**NetBT**), and *Smart Card Services* (**SCardSvr**). The `Network Configuration Operators` group has permission to configure the *Dynamic Host Configuration Protocol* (**DHCP**), **NetBT**, and **Dnscache** services. This defeats the principle of least privilege that was the motivation for creating `Local Service` and `Server Operators`. If the adversary were to find a buffer overflow bug in a program running as `Local Service`, this escalation path enables the adversary to take complete control of the host.

Finally, although Microsoft describes `Power Users` as "includes many, but not all, privileges of the Administrators group," [5, page 31] it is well known that • there are

many privilege-escalation paths from `Power Users` to `Local System`; we have found more than 20 with our tool.

Other vendors' software also has access-control configuration bugs in their services:
• The `Everyone` group was granted the permission to configure the *Macromedia Licensing Service*, installed by **Macromedia's** Dreamweaver program.

SERVICE_USER_DEFINED_CONTROL weakness. Windows allows each service to specify a custom control code to be sent to a service. The permission SERVICE_USER_DEFINED_CONTROL is used to control the set of users who can send control codes to the services. Control codes represent some general operations such as start, continue, pause, and stop. One way to create a new service in Windows is to use the `sc.exe` utility. Unfortunately, when an administrator creates a service using this utility, the `Authenticated Users` groups is granted this permission by default. This places the burden of setting the service's security context on the service creator. Developers do not understand the security implications well enough to do the right thing. This is a poor design decision made by the `sc.exe` utility of the operating system; it could lead to vulnerabilities. I do not know of services that use user defined control codes, but this is a weak point in the system.

7.1.2 Registry misconfigurations

Registry The Windows Registry is a global, hierarchical database, where entries are accessed by *keys*. Each registry key has a security context attached to it controlling access to the key. Some registry keys store sensitive information like the path to the executable acting as a user's shell, the library to be loaded by a program, and the identity of an

operating system object[1] . If an adversary can overwrite the contents of a sensitive key with the path of his library or executable, he could cause his code to be executed [47, 28]. • The standard configuration of **AOL** includes a registry entry binding the name of a DLL file to be loaded and executed (in some circumstances) by the AOL software. The access permissions permit any user to write this entry; the attacker can substitute the name of his own DLL and wait for some other AOL user to execute it. • We also found several weaknesses (potential security holes) in several registry keys from several vendors where an adversary could escalate his privileges.

7.1.3 File misconfigurations

In addition to Trojan horses via service configuration, some vendors' software is vulnerable to a more traditional kind of file-system-based Trojan-horse vulnerability: • The `Everyone` group has been granted the permission to write to 170 executable (.EXE and .DLL) files from **Adobe**. The adversary can write to these files and wait for a system administrator or other user to execute the files. • The `Everyone` group has been granted the permission to write to 103 files from the Trend Micro Internet Security 2006 virus scanner from **Trend Micro**. • The `Everyone` group has been granted the permission to write to 354 files in Lotus Notes program from **IBM**.

Program Name	Version	Vendor	Mechanism	#instances	Remarks
Lotus Notes	6.5.4	IBM	File	354	`Everyone`
VPN Client		Cisco	File	18	`Interactive`
PC-cillin 2006	14.10.23	Trend Micro	File	103	`Users`
PC-cillin 2006	14.10.23	Trend Micro	Service	6	`Authenticated`
Illustrator		Adobe	File	170	`Everyone`
Anti-virus		Symantec	File	6	`Everyone`
AOL Messenger		AOL	Registry	2	`Everyone`
Dreamweaver		Macromedia	Service	1	`Everyone`
Flash		Macromedia	File	1	`Everyone`
Windows XP		Microsoft	Service	2	`Everyone`
Windows XP		Microsoft	Service	3	`Local Service`
Windows XP		Microsoft	Service	1	`Network Service`
Windows XP		Microsoft	Registry	20	`Local Service`
Windows XP		Microsoft	Registry	20	`Network Service`
Windows XP		Microsoft	Service	1	`NetCfgOp`

Figure 7.4: A summary of vulnerabilities discovered by our tool. Major software vendors make mistakes in using the operating system access control. It is quite possible that there are a large number of open vulnerabilities in software that we have not tested. We suggest that developers use tools like the one we describe in this thesis to test their software. The column instances reflects the number of objects that lead to a privilege-escalation vulnerability. For example, Lotus Notes has 354 files that are writeable by any member of the `Everyone` group.

7.1.4 Summary of findings

7.1.5 Misconfigurations in System Restore.

It is common that when an administrator installs a program, he discovers that the installation has undesirable (sometimes disastrous) consequences. It is useful to be able to roll back the effects of software installation. Windows provides the *system restore* facility to provide a way to easily revert the system to a previously known (good) state. System Restore in Windows enables the administrator takes multiple snapshots of the system state and revert the these snapshots when required. If a software installation were to corrupt the system state, system restore can be used to go back to a previously known good state. By design only the administrators group is allowed to run the system restore utility.

System Restore stores its files in the `C:\System Volume Information` directory. This directory's ACL is set so that only the `Local System` can access the contents of this directory. This directory has a subdirectory with name similar to `_restore{34DA1123-3456-76ED-EDAB-1234567890AA}`[2]. On this directory, any member of the `Everyone` group has complete access. For each restore point, the system creates a directory like `RP10` and stores the data for each restore point into the `snapshot` subdirectory[3]. We found that the following files in the `snapshot` directory have weak access control, enabling any member of the `Everyone` group have complete access:

[1]Windows identifies certain operating system objects ("classes") by globally unique identities like *4D36E96B-E325-11CE-BFC1-08002BE10318*

[2]We conjecture that the hexadecimal number in the directory name is generated by an equivalent of the `guidgen.exe` program. This program generates a globally unique 128 bit hexadecimal number. Windows uses globally unique identifiers (GUIDs) to identify objects such as ActiveX classes and interfaces. However, it is not clear if this identifier is cryptographically strong. When an adversary gets limited access to a host, he may be able to leverage other attacks to predict this number.

[3]Windows XP. Version 5.1 (build 2600.xpclnt_qfe.010827-1803)

_REGISTRY_MACHINE_SAM

_REGISTRY_MACHINE_SECURITY

_REGISTRY_MACHINE_SOFTWARE

_REGISTRY_MACHINE_SYSTEM

_REGISTRY_USER_.DEFAULT

_REGISTRY_USER_NTUSER_S-1-5-18

_REGISTRY_USER_NTUSER_S-1-5-19

_REGISTRY_USER_NTUSER_S-1-5-20

_REGISTRY_USER_NTUSER_<usersid>

Usually all users are granted SeChangeNotifyPrivilege privilege, thus the adversary can skip the restrictive access checks on parent directories. The adversary's ability to access these files is only controlled by the access control settings on the files. The access control settings on the files allow any member of the Everyone group to write to the files. Thus an adversary who gains limited access to the host can corrupt the files. From the name of the files, we conjecture that these are the files backing up the registry for the machine and thus the adversary can corrupt the entire registry backup. If an administrator were to try to use *system restore* to restore to a previous checkpoint, the machine's registry can be compromised and hence the host. The adversary will be able to corrupt these files only if:

- He knows the complete path to the file. The adversary does not immediately know the complete path of the corrupted file because he cannot read the contents of the C:\System Volume Information directory. The adversary cannot read the contents of the C:\System Volume Information and hence cannot know the name of the restore directory. The restore directory has a name similar to _restore{34DA1123-3456-76ED-EDAB-1234567890AA}, generated by

using `guidgen.exe`. If the adversary can leverage other attacks to guess the name of the restore directory, he could guess the complete path to the file.

- When a user accesses a file, the permissions along the directory path need to be checked to ensure that the user is allowed access to the file. This could lead to severe run-time performance penalties for deeply nested files. The Windows solution to these penalties is to copy the security descriptors from parent directories to the child nodes of the filesystem, and have the kernel *optionally* skip access checks along on the directory path. The `SeChangeNotifyPrivilege` privilege in the process token controls this behavior. This privilege is typically enabled for all users because most applications in Windows break when this privilege is disabled. Thus the adversary can skip the restrictive permissions on the parent directories in the system restore directory. In effect, the only protection on the system restore registry snapshots is the security descriptors on the files that store the settings, which are too weak.

Though this attack is more difficult to launch than other attacks, this attack exposes the weaknesses of using Windows in high-assurance systems.

Determining the name of the restore directory. We conjecture that one can guess the restore directory name by trying to open the directory
`C:\System Volume Information\`
`_restore{34DAXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX}` and measure the time taken to get a failure. Applications break if the privilege `SeChangeNotifyPrivilege` is not granted to a user; hence this privilege is granted by default to all users. With this privilege turned on, when the kernel receives a call to open a file or directory, the kernel does not perform access checks along the directories in the path, but only performs the

access check on the *leaf node*. Thus, when any principal tries to open a file, the system will have to look up whether the file or directory exists—as opposed to getting an access-denied error on one of the parent directories. In this particular case, since the parent directory `C:\System Volume Information` always exists, this boils down to determining if the file `_restore{34DAXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX}` exists. In case the file does not exist, we would get an error in opening the file. It is intuitive that the more the characters that match the file name, the more the time taken to get the error. Given that modern kernels are highly optimized for speed, it is unlikely that the kernel deliberately slows down when a file name did not match. By measuring the time taken to get an error, it should be feasible to determine the number of characters in the current guess that are correct. One could then launch an incremental attack. We conjecture that the hierarchy (including the names of the files and directories) of the whole file system can be determined by any user of the Windows system—a potentially big problem. Previously, Andrew Griffiths has shown that in some Unix systems due to implementation of various system calls (`open` in particular), it becomes possible to test whether or not a file exists in a directory that is unreadable [22].

7.2 Network security analysis

7.2.1 A small real-world example

We ran our tool on a small network used by seven hundred users. We analyzed a subset of the network that contains only machines managed by the system administrators.[4] Our

[4]In this benchmark we did not model hundreds of user machines. We recommend that these should be modeled as we did "internet," as one machine. In this case, unlike "internet, " the host would have non-malicious users, but would be assumed to have many vulnerabilities. In our future work we plan to experiment with such models; at present we recommend our framework for networks of managed, not unmanaged, hosts.

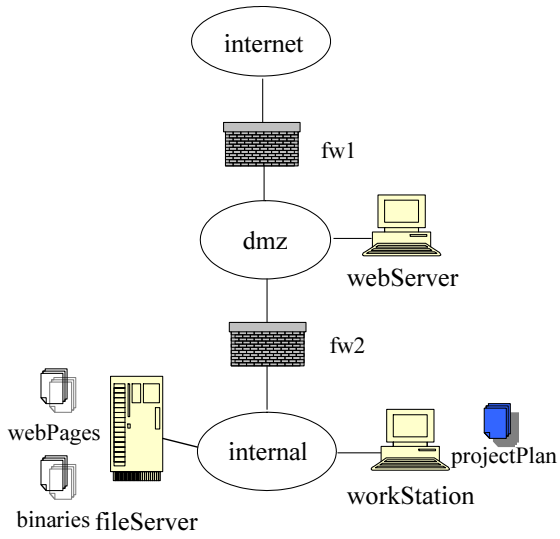

Figure 7.5: Network topology for the network discussed in section 7.2.1

tool found a violation of policy because of a vulnerability. The system administrators subsequently patched the bug.

Network topology. The topology of the network is very similar to the one in Figure 7.5. There are three zones (`internet`, `dmz` and `internal`) separated by two firewalls (`fw1` and `fw2`). The administrators manage the `webserver`, the `workStation` and the `fileserver`. The users have access to the public server `workStation` which they use for their computing needs. The host access control list for this network is:

```
hacl(internet, webServer, tcp, 80).
hacl(webServer, fileServer, rpc, 100003).
hacl(webServer, fileServer, rpc, 100005).
hacl(fileServer, AnyHost,  AnyProtocol, AnyPort).
hacl(workStation, AnyHost, AnyProtocol, AnyPort).
hacl(H, H, AnyProtocol, AnyPort).
```

Machine configuration The following Datalog tuples describe the configuration information of the three machines.

```
networkService(webServer , httpd, tcp , 80 , apache).
nfsMount(webServer, '/www', fileServer, '/export/www').

networkService(fileServer, nfsd, rpc, 100003, root).
networkService(fileServer, mountd, rpc, 100005, root).
nfsExport(fileServer, '/export/share', read, workStation).
nfsExport(fileServer, '/export/www', read, webServer).

nfsMount(workStation, '/usr/local/share', fileServer,
                                    '/export/share').
```

The `fileServer` serves files for the `webServer` and the `workStation` through the NFS protocol. There are actually many machines represented by `workStation`. They are managed by the administrators and run the same software configuration. To avoid the hassle of installing each application on each of the machines separately, the administrators maintain a collection of application binaries under `/export/share` on `fileServer` so that any change like recompilation of an application program needs to be done only once. These binaries are exported through NFS to the `workStation`. The directory `/export/www` is exported to `webServer`.

Data binding.

```
dataBind(projectplan, workStation, '/home').
dataBind(webPages, webServer, '/www').
```

Principals. The principal `sysAdmin` manages the machines with user name `root`. Since all the users are treated equally, we model one of them as principal `user`. `user` uses the `workStation` with user name `userAccount`. For this organization, the primary worry is a remote attacker launching an attack from outside the network. The attackers are modeled by a single principal `attacker` who uses the machine `internet` and has complete control of it. The Datalog tuples for principal bindings are:

```
hasAccount(user, workStation, userAccount).

hasAccount(sysAdmin, workStation, root).
hasAccount(sysAdmin, webServer, root).
hasAccount(sysAdmin, fileServer, root).

hasAccount(attacker, internet, root).
malicious(attacker).
```

Security policy The administrators need to ensure that the confidentiality and the integrity of users' files will not be compromised by an attacker. Thus the policy is

```
allow(Anyone, read, webPages).
allow(user, AnyAccess, projectPlan).
allow(sysAdmin, AnyAccess, Data).
```

Results We ran the MulVAL scanner on each of the machines. The interesting part of the output was that `workStation` had the following vulnerabilities:

```
vulExists(workStation, 'CAN-2004-0427', kernel).
```

```
vulExists(workStation, 'CAN-2004-0554', kernel).
vulExists(workStation, 'CAN-2004-0495', kernel).
vulExists(workStation, 'CVE-2002-1363', libpng).
```

The MulVAL reasoning engine then analyzed this output in combination with the other inputs described above. The tool did indeed find a policy violation because of the bug `CVE-2002-1363`—a remotely exploitable bug in the `libpng` library. The adversary creates a malicious image file that exploits the bug, then uploads this image to a popular website and waits for the user to download the picture from the site. When the user loads the image, the adversary obtains control of the user's account on `workStation` machine by launching the exploit. A reasoning rule in our framework for remote exploit derives that the `workStation` machine can be compromised. After obtaining control of the user's account on the host, it is trivial for the adversary to access `projectPlan` files because the user already has access to them. The reasoning rule for file access in our framework derives that the adversary can access `projectPlan` files. Thus the `projectPlan` data can be accessed by the attacker, violating the policy. Our system administrators subsequently patched the vulnerable `libpng` library.

One might be curious that there was only one vulnerability that contributed to the policy violation though the host `workStation` actually had four vulnerabilities. The other three bugs on the `workStation` are locally exploitable vulnerabilities in the kernel. Since only trusted users access these hosts, after patching the `libpng` bug our tool indicates the policy is no longer violated. These machines have uptimes in the order of months and upgrading the kernel would require a reboot. Patching these vulnerabilities would result in a loss of availability, which is best avoided. Our tool showed the administrators that they can meet the security goals without patching the kernel and rebooting the

workStation. We expect our tool to be useful in mission-critical systems like commercial mail servers serving millions of users and servers running long computations.

7.2.2 An example multistage attack

We now illustrate how our framework works in the case of multistage attacks. Let us consider a simulated attack on the network discussed in the previous example. Suppose the following two vulnerabilities are reported by the scanner:

```
vulExists(webServer, 'CVE-2002-0392', httpd).
vulExists(fileServer, 'CAN-2003-0252', mountd).
```

Both vulnerabilities are remotely exploitable and can result in privilege escalation. The corresponding Datalog clauses from ICAT database are:

```
vulProperty('CVE-2002-0392', remoteExploit, privEscalation).
vulProperty('CAN-2003-0252', remoteExploit, privEscalation).
```

The machine and network configuration, principal and data binding, and the security policy are the same as in the previous example.

Results The MulVAL reasoning engine analyzed the input Datalog tuples. The Prolog session transcript is as follows:

```
| ?- policyViolation(Adversary, Access, Resource).


Adversary = attacker
Access = read
Resource = projectPlan;
```

```
Adversary = attacker
Access = write
Resource = webPages;

Adversary = attacker
Access = write
Resource = projectPlan;
```

We show the trace of the first violation in figure 7.6. Here we explain how the attack can lead to the policy violation. An attacker can first compromise `webServer` by remotely exploiting vulnerability `CVE-2002-0392` to get control of `webServer`. Since `webServer` is allowed to access `fileServer`, the adversary he can then compromise `fileServer` by exploiting vulnerability `CAN-2003-0252` and become `root` on the server. Next he can modify arbitrary files on `fileServer`. Since the executable binaries on `workStation` are mounted on `fileServer`, their integrity will be compromised by the attacker. Eventually an innocent user will execute the compromised client program; this will give the attacker access to `workStation`. Thus the files stored on it would also be compromised.

One way to fix this violation is moving `webPages` to `webServer` and blocking inbound access from `dmz` zone to `internal` zone. After incorporating these counter measures, we ran MulVAL reasoning engine on the new inputs and verified that the security policy is satisfied.

```
|-- policyViolation(attacker,read,projectPlan)
 |-- dataBind(projectPlan,workStn,/home)
 |-- accessFile(attacker,workStn,read,'/home')
 Rule: execCode implies file access
    |-- execCode(attacker,workStn,root)
    Rule: Trojan horse installation
       |-- malicious(attacker)
       |-- accessFile(attacker,workStn,write,'/sharedBinary')
       Rule: NFS semantics
          |-- nfsMounted(workStn,'/sharedBinary',fileSrv,'/export',read)
          |-- accessFile(attacker,fileSrv,write,'/export')
          Rule: execCode implies file access
             |-- execCode(attacker,fileSrv,root)
             Rule: remote exploit of a server program
                |-- malicious(attacker)
                |-- vulExists(fileSrv,CAN-0252,mountd,remote,privEsc)
                |-- networkServiceInfo(fileSrv,mountd,rpc,100005,root)
                |-- netAccess(attacker,fileSrv,rpc,100005)
                Rule: multi-hop access
                   |-- execCode(attacker,webSrv,apache)
                   Rule: remote exploit of a server program
                      |-- malicious(attacker)
                      |-- vulExists(webSrv,CAN-0392,httpd,remote,prvEsc)
                      |-- networkServiceInfo(webSrv,httpd,tcp,80,apache)
                      |-- netAccess(attacker,webSrv,tcp,80)
                      Rule: direct network access
                         |-- located(attacker,internet)
                         |-- hacl(internet,webSrv,tcp,80)
                   |-- hacl(webSrv,fileSrv,rpc,100005)
             |-- localFileProtection(fileSrv,root,write,/export)
    |-- localFileProtection(workStn,root,read,/home)
 |-- not allow(attacker,read,projectPlan)
```

Figure 7.6: A sample attack tree showing a multi-stage attack on the network shown in figure 7.5. This trace provides a detailed chain of reasoning that explains why the security policy of the network is violated. Alternatively, the trace shows the detailed steps an adversary needs to perform to launch the attack. The hierarchy in the trace shows the dependencies between various steps of the attack. The trace is useful for the administrator to decide where to break the attack chain to foil the attack.

7.3 Attack graph or attack trace?

In figure 7.6, we represented the attack as an attack trace, where each step in the attack is shown sequentially. In contrast, in figures 7.1 and 7.2 we show all the attacks possible as a graph. This raises the question as to how one can efficiently represent all the privilege escalations possible in a network. What is the best way to efficiently represent all the privilege escalations possible in a network? Should one present the results as a list of all attack traces? Or should one present the results in a graph, where each edge represents an attack step from one location state to another. We note that one can compute one representation from the other. By interviewing the administrators who used our tool, we found that representing the graph as an edge makes it easy for them to visualize the attacks. More importantly, it efficiently represents the attacks.

Let us consider the escalations from `Limited Users` to `Authenticated Users`, `Network Service` to `Authenticated Users`, and then to `Local System` as shown in figure 7.1. These escalations have been represented by using four arcs: one each from `Limited Users` to `Authenticated Users` and from `Network Service` to `Authenticated Users` showing the group memberships and two from `Authenticated Users` to `Local System` showing the `SSDP` and `uPnP` services. If we were to show these escalations as attack traces, the representation would look like:

```
Limited User-->Authenticated User---(SSDP)--->Local System
NetworkService-->Authenticated User--(SSDP)-->Local System
Limited User-->Authenticated User---(uPnP)--->Local System
NetworkService-->Authenticated User--(uPnP)-->Local System
```

If there is another escalation from `Authenticated Users` to `Local System`, then the graph representation has only one extra arc, while the attack trace representation has two extra traces.

If there a large number of escalations possible, then we get efficient representation with attack graphs as against with attack traces. An attack trace is essentially a walk on the attack graph. It is well known that in a dense graph, the number of unique walks is super-exponential. Thus number of attack traces will be super exponential if there are a large number of escalations possible. On the other hand, the complexity of the graph is bounded by the number of atomic escalations possible. Our tool computes the total escalations possible and runs in quadratic time. Thus, the total number of escalations possible is a quadratic function. Hence, representing the output as an attack graph has quadratic complexity. Therefore, one should represent the output of our tool attack as an attack graph as opposed a list of escalation traces.

7.4 Quantitative bug analysis

System administrators require the ability to measure the risk of current network posture. A measurement of risk is valuable to determine the amount of risk reduction achieved by counter measures. Network administrators have an interest in measuring the security risk of running an operating system. When an administrator estimates the security risk, he adopts multiple measures. One measure is the number of bugs that are found in the operating system in a unit time. This measure is readily available by visiting vulnerability databases maintained by organizations like CERT and SANS. When an administrator reads a vulnerability advisory, he is interested in determining whether the bug can be remotely exploited. This information is usually readily available in the bug advisory. A

more important question on the administrator is the consequences of the bug—does it result in an administrative account compromise?

Unfortunately, for a typical system administrator, it is very hard to determine whether the bug results in an administrative account compromise. To determine if a bug can result in an administrative account compromise, the administrator will have to determine the program or library affected by the bug. This task is usually easy, because its mentioned in the vulnerability advisory. The administrator will then have to determine all the programs that are dependent on this library. If there is a bug in a file (library or program), then any program using the file is vulnerable. Similarly, we define that a registry key is vulnerable if the key contains the path to a vulnerable executable. The motivation is that some other program uses this registry key to determine which program to load.

In today's dynamic networks, the software environment evolves quickly and it is hard to determine what programs use what programs or libraries or registry keys. A bug results in a system-wide compromise if the program gets used during the operating system boot or while an administrator logs on to a host. We need to determine when a program gets used. Unfortunately, it is difficult to determine the conditions under which a program gets used. To determine when a program gets used, we adopted the conservative approach of tracing to determine when a file or registry key gets used.

In Windows, there are two important resources—files and registry keys. We want to monitor registry and file usage. One approach would be to modify the kernel by adding a tracing driver. The advantage of this approach is that we do not modify much state (registry keys, audit log settings, file and registry access control lists that control auditing) on the system. In a prior system, we were modifying state on the target machine and sometimes bugs in our code resulted in catastrophic failures, where sometimes I had to reinstall the operating system. Thus, as opposed to modifying state on the host, we would

prefer to introduce a tracing driver in the kernel. However, we were advised against it by an experienced kernel hacker [31]. We were advised that project could easily consume two full-time developers a year and one very easily would run into bugs in the kernel. The fact we do not have access to Windows source code will only make it difficult to pursue this path.

Instead, we took the approach of using already existing registry and file tracing tools. Thus, we decided on using usermode tracing tools to monitor registry and file usage. We used the *regmon* utility to trace the registry usage. Regmon has the ability to monitor registry accesses in the boot processes, before any user can login. We tried to use the *filemon* utility to monitor the file system accesses, but unfortunately, filemon does not have the ability to monitor the boot processes. It can be run only after a user has logged on. We decided to use the Windows auditing facility to see what user runs what program. We turned on the auditing of process execution—whenever any .exe program is executed, the system writes an entry to the audit log mentioning the path of the program executed and the privilege level of the process. This means that we cannot trace the usage of other types of executable content in Windows like the file types: `.cmd, .drv, .msc, .mof, .ocx, .sys, .tsp, .bat, .dos, .cpl`. Another problem we face is that there is no easy way to turn on auditing of uses of all dynamically-linked library (DLL) files. To audit DLL file accesses, we would have to locate all the DLL files in the file system and turn on auditing on each of the files. Since this involves changning the state of the file system on a large number of files, we ignored this approach for now. If our program modified the state on the target system, a bug in our program may leave the operating system in an inconsistent state. We decided not to trace the DLL usage at run time. Instead, we relied on static tools like *depends* written by Steve Miller to tell us what file uses what DLL.

From the output of the *regmon*, *filemon* and the system log, we determine who is using the file or registry key. By determining the users of a vulnerable resource, we identify the ultimate privilege level the adversary obtains by leveraging a vulnerability. We performed this experiment on a Microsoft Windows XP machine that is not completely patched. We got the following results:

Vulnerable file path	Exploit level
C:\Program Files\Common Files\Microsoft Shared\TextConv\mswrd632.wpc	User-level
C:\Program Files\Windows Media Player\wmplayer.exe	User-level
C:\Program Files\Windows NT\Accessories\wordpad.exe	User-level
C:\WINDOWS\system32\crypt32.dll	System
C:\WINDOWS\system32\hlink.dll	System
C:\WINDOWS\system32\inetcomm.dll	System
C:\WINDOWS\System32\Ntoskrnl.exe	System
C:\WINDOWS\system32\rpcrt4.dll	System
C:\WINDOWS\system32\shell32.dll	System

While the number data points is small, we conjecture that a majority of bugs in Windows would result in a system-wide compromise.

7.5 Performance and Scalability

We measured the performance of our Linux scanner on a Red Hat Linux 9 host (kernel version 2.4.20-8). The CPU is a 730 MHz Pentium III processor with 128MB RAM. We measured the performance of our Windows scanner on a Windows XP host. The CPU is a 2.2GHz Pentium IV processor with 512MB RAM. The analysis engine runs on a Windows PC with 2.8GHz Pentium 4 processor with 512MB RAM. We constructed

examples with configurations similar to the network in section 7.2, but with different numbers of web servers, file servers and workstations.

To analyze a network in the MulVAL reasoning engine, one needs to run the MulVAL scanner on each host and transfer the results to the host running the analysis engine. The scanners can execute in parallel on multiple machines. The analysis engine then operates on the data collected from all hosts. Since the functioning of the scanner is the same on various hosts, we measured the scanner running time on one host. We measured the running time for the analysis engine for real and synthetic benchmarks. The running times (in seconds) are as:

MulVAL Linux scanner		236 s
MulVAL Windows scanner		386 s
MulVAL reasoning engine	§7.2.1	0.08
	1 host	0.08
	200 hosts	0.22
	400 hosts	0.75
	1000 hosts	3.85
	2000 hosts	15.8

MulVAL scanner is the time to run the scanner on one (typically configured) Linux host; in principle, the scanner can run on all hosts in parallel. The benchmark §7.2.1 is the real-world 3-host network described in section 7.2.1. Each benchmark labeled "n hosts" consists of n similar Linux hosts, (approximately one third web servers, one-third file servers, and one-third workstations), with host access rules (i.e., firewalls) similar to §7.2.1. Our reasoning engine can handle networks with thousands of hosts in less than a minute.

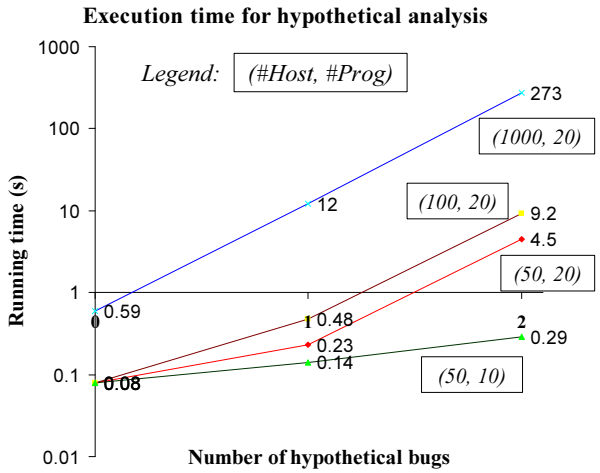

Figure 7.7: Hypothetical analysis. For a network of 1000 hosts running 20 kinds of installed software, analyzing security assuming the existence of any 1 unreported vulnerability takes 12 seconds.

A typical network might have a dozen kinds of hosts: many web servers, many file servers, many compute servers, many user machines. Depending on network topology and installed software (e.g., are all the web servers in the same place with respect to firewalls, and are they all running the same software?) it may be possible that each group of hosts can be treated as one host for vulnerability analysis, so that $n = 12$ rather than $n = 12,000$. It would be useful to formally characterize the conditions under which such grouping is sound.

To test the speed of our hypothetical analysis discussed in section 5.5, we constructed synthesized networks with different numbers of hosts and different numbers of programs. Each program runs on multiple machines. Since the hypothetical analysis goes through all combination of programs to inject bugs, the running time is dependent on both the number of programs and the number of hypothetical bugs. Figure 7.7 shows the performance with regard to different number of hosts, number of programs and number of injected bugs. The running time increases with the number of hypothetical bugs, because the analysis

engine will need to go through $\binom{n}{k}$ combinations of programs, where n is the number of different kinds of programs and k is the number of injected bugs. $k = 0$ is the case where no hypothetical bug is injected. The performance degraded significantly with the increase of k. But it still only takes 273 seconds for $k = 2$ on a network with 1000 hosts and 20 different kinds of programs. Since hypothetical analysis can be performed offline before the existence of a bug is known, it is not important to have fast real-time response time. The degraded performance is acceptable. Figure 7.7 shows our system can perform this analysis in a reasonable time frame for a big network.

The input size (measured by the number of lines) to the MulVAL reasoning engine is:

Data	Source[5]	hosts=200	=2000
Data Binding	sys admin	26	3004 lines
Policy	sys admin	3	3
Principal Binding	sys admin	10	10
HACL	Smart Firewall	342	3342
Scanner Output	OVAL/ICAT	1222	12022

Coverage Our system can reason about privilege escalation vulnerabilities and denial of service vulnerabilities. We cannot currently reason about confidentiality loss or integrity loss vulnerabilities. Overall, we could reason about 84% of the Red Hat Linux bugs reported in OVAL. The detailed statistics are (as of January 31, 2005):

OVAL definitions for Red Hat	202
Those with PrivEsc or only DoS	169
Coverage	84%

[5]The indicated "Source" shows what person or tool would provide the information in a real installation; for this benchmark measurement, we constructed the data synthetically.

Size of our code base To implement our framework on Red Hat platform, we adapted the OVAL scanner and wrote the interaction rules. The size of our code base is:

Module	Original	New
OVAL scanner	13484	668 lines
Interaction rules		393

The modularity and simplicity of our design allowed us to effectively leverage the existing tools and databases by writing about a thousand lines of code. We note that the small size and declarative style of our interaction rules makes them easy to understand and debug. The interaction rules model Unix-style security semantics. The rules are independent of the vulnerability definitions.

7.5.1 Scanning a distributed network

We measured the performance of running the MulVAL scanner in parallel on multiple hosts. We used PlanetLab, a worldwide testbed of over 500 Linux hosts connected via the Internet [36]. We selected 47 hosts in such a way as to get geographical diversity (U.S., Canada, Switzerland, Germany, Spain, Israel, India, Hong Kong, Korea, Japan). We were able to log into 39 of these hosts; of these, we successfully installed the scanner on 33 hosts.[6] We ran a script that, in parallel on 33 hosts, opened an SSH session and ran the MulVAL scanner. We assume that many hosts were carrying a normal workload, as we made no attempt to reserve them for this use. The first host responded with data in 1.18 minutes; the first 25 hosts responded within 10 minutes; the first 29 hosts responded within 15 minutes; at this point we terminated the experiment.

[6]Normally one needs root privileges to install the scanner; PlanetLab gives its users fake "root" privileges in a chroot environment; for production use of MulVAL, root privileges are advisable.

For a local area network, we expect fast and uniform response time. But for distributed networks, we recommend that scanning be done asynchronously. Each machine, either when its configuration is known to have changed or periodically, should scan and report configuration information. Then, whenever newly scanned data arrives or whenever new vulnerability data is obtained from OVAL or ICAT, the reasoning engine can be run within seconds.

Chapter 8

Conclusion

Despite the fact that business processes are critically dependent on smooth functioning of networks, today's networks are surprisingly fragile. Penetration testers, security experts and expert system administrators have been using ad hoc methods to analyze the security of a single host and, more generally, a large network. This thesis establishes that configuration errors are an important problem, and discusses techniques to formalize network security analysis. The thesis presents techniques to formally describe and reason about the (access control) semantics of operating systems. This thesis presents techniques to perform automated security analysis of configuration and program vulnerabilities of a large network. These techniques have found a large number of previously unknown serious bugs. In section 8.1, we describe reasons why people make mistakes in configuration. In section 8.2, we describe the contributions of this thesis. We discuss possible steps forward in section 8.3.

8.1 Reasons for configuration problems

We have shown that configuration bugs often cause security vulnerabilities. There are many structural causes for the mistakes that people make in software configuration. A large enterprise network is typically managed by multiple system administrators, wherein each administrator is responsible for specific functionality and where there is a limited overlap between the operational domains of different administrators. Interactions between administrators may be limited by various barriers such as different business processes, different administrative domains, different operating systems and technologies. The result is that each administrator configures his system independently. The global security behavior may in fact be dependent on the configurations of multiple hosts, as well as the dependencies between these hosts.

Operation of a network depends on the interactions of configurations across multiple boundaries, but network operators typically do not have access to configurations across boundaries. There is no way to guarantee that policies configured in his network will not conflict with rest of the network. Worse, the inconsistencies may result in security holes. It is hard to debug these configuration problems.

Professional system administrators and software developers are guided in their tasks by the behavior of software system as documented by the vendor. However, as we demonstrated in this thesis, often the vendors' documentation is obscure and sometimes even wrong. Software developers often don't understand operating-system security semantics and often can't predict how these will interact with customers' security configurations. To ensure that their programs always work, they tend to ask for too many privileges [8]. System administrators are forced to permit too much access to users because applica-

tions do not work otherwise. With this leakage of privileges, it is inevitable that there are security bugs.

Until now, there have been no tools or techniques developed to help administrators reason about the configuration of the network *as a whole*. It is very hard for an administrator, security expert, or penetration tester to determine the global effects of a potential configuration change. It is not surprising that configuration issues present the adversary with a very useful avenue to attack enterprise networks.

8.2 Contributions

This thesis explains new techniques to reason about the correctness of configurations. These techniques were incorporated into an automated tool to find serious weaknesses in configuration of the Windows operating system and programs running on the operating system. The tool found several *new* attacks against a standard Windows host. Thus, the dissertation work establishes that configuration vulnerabilities are an important avenue of attacks against common software systems.

This thesis demonstrates that end-to-end, automatic, efficient network security analysis is feasible for large real networks. We developed new techniques to perform multi-stage attacks on a network where an adversary leverages multiple weaknesses to incrementally increase the potency of attack. Our work identifies that the modularity of information flow between the security expert, bug expert and system administrator is crucial for a security analysis tool to be useful in practice. Our work establishes that the semantics of common operating systems, network environments and common failures for software can be modeled declaratively. Declarative specification allows us clean specification, yet efficient evaluation. In fact, we demonstrated that real network configurations

can be analyzed in polynomial time. Previous work had significant scalability problems making the systems unusable for more than a handful of hosts. The specification of semantics in previous work was obscure. By adopting a simple representation for specification, we showed how one can design a system whose correctness can be easily verified. Previous work could not conduct analysis over a heterogeneous network with multiple operating systems. The ability to conduct analysis over a heterogeneous network—as described in this thesis—is a significant advancement. Formal specification for typical operating systems is an important contribution of this thesis.

Quantitative analysis to determine the risk of current network posture is increasingly in demand. Formal specification is the first step in quantitatively analyzing the risk of current network posture. Using a formal specification of the operating system, this thesis (in section 7.4) presented techniques that use historical data to quantify the probability that a security bug results in a system-wide compromise. In the future, these techniques could be improved to determine the risk posture of the *entire* network.

A concern that an administrator worries about is how much unknown threats the network can withstand. It is difficult to forsee a situation in the near future where no more software holes are found in commonly used software. It is inevitable that more bugs will be found in software in the future; however the administrator cannot predict which software will have what bug in the future and when. In addition, there may be bugs that exist in a piece of software that are not reported publicly; such bugs are called zero-day bugs. Or a user account may be compromised with neither the user or administrator being aware of it. A concern administrators use in configuring hosts is how vulnerable the network is to unknown threats. That would enable an administrator to plan for potential emergencies. In this thesis, we describe techniques to rigorously understand what are the consequences of potential emergencies that can emerge in the future. Previously, there

is no formal analysis of potential attack scenarios. In fact, the techniques we describe in this thesis are efficient (see section 7.5 for further details).

8.3 Moving forward from lessons learned

We have a system that already does enterprise-scale analysis of networks of Unix and Windows hosts, and detailed configuration analysis of individual Windows hosts. Based on our experience in deploying these prototypes, we identified certain short-term research that needs to be done and and longer-term research problems that need to be investigated.

Response strategies. We deployed MulVAL on a professionally managed network and found several serious local privilege escalation vulnerabilities. In some cases, the administrators and developers were able to identify that changing the security descriptors of the vulnerable objects was sufficient to solve the security problem. For example, as we discussed in chapter 7, many security problems resulted because members of `Everyone` group were assigned the permission to "change configuration" (`SERVICE_CHANGE_CONFIG`) for certain services, the permission to write to a file or a registry key. In these cases, it was straightforward to program our tool to change the security descriptor to disallow untrusted users from modifying the resource or its properties. In general, administrators found it difficult to identify an appropriate remediation strategy. In a more complex environment, it is feasible that the administrator will have to choose between various alternatives such as disabling a service, disabling a user, patching a program, and adding a firewall rule. It is hard for the administrator to identify the best remedy. We plan on extending our framework to automatically identify least-cost remedies.

Quantitative risk estimation. The community has been working on quantifying various aspects affecting security such as user's skill level (*is this user likely to open email attachments?*), the perceived attacker's skill level, the importance of the program affected by the bug [26]. However, the risk estimations do not have a formal model for the software environment and the adversarial behavior. As a result the estimations tend to be ad hoc and error prone. For example, when a bug is reported in a library file, the vulnerability scoring systems ignore the surrounding software context like what programs are using this library and the privilege level of the programs using this library. We have done preliminary work on formal vulnerability risk analysis yielding results like *60% of security bugs found on the Windows platform result in system-wide compromise*. We plan on using the MulVAL framework to do rigorous quantitative risk estimation and vulnerability scoring system

Simple models for complex systems In this thesis work, we demonstrated that simple models can go a long way to reason about complex problems like security. We hypothesize that simple models can be used to solve other problems faced by system administrators and developers. For example, sometimes a system administrator installs a program, only to discover that the program is incompatible with the existing software . He cannot easily undo the effects of an aborted installation. An efficient framework to describe the program installation dependencies would enable one to test if a given operation will result in an inconsistent state before actually installing the software.

In the course of implementing this project, I encountered various instances where the compiler/linker complained about missing libraries. Sometimes, because of static linking, some libraries that are common to different components (such as standard input/output routines) are included in different components, resulting in conflicts during linking. A

framework to describe and reason about the program and library dependencies would make it easier for developers to avoid these errors.

8.4 Conclusion

This dissertation has established that (1) configuration errors are an important source of attacks against common software systems, (2) it is possible to construct tools to automatically analyze configurations, (3) efficient, end-to-end, automatic network security analysis is feasible for large networks, (4) it is possible for the administrator to rigorously plan for potential attack paths using unknown exploits, (5) it is possible to quantify the risk profile associated with a bug. The thesis has established that declarative specification of component behavior is the key to building an efficient and practical framework to perform network security analysis. The techniques developed in this dissertation can help resolve the tension between flexibility and complexity that exists in managing any large network.

Bibliography

[1] P. Ammann, D. Wijesekera, and S. Kaushik. Scalable, graph-based network vulnerability analysis. In *Proceedings of 9th ACM Conference on Computer and Communications Security*, Washington, DC, November 2002.

[2] M. J. Bach. *The Design of the UNIX Operating System*. Prentice Hall, 1986.

[3] R. Baldwin. Rule based analysis of computer security. Technical Report TR-401, MIT LCS Lab, 1988.

[4] Y. Bartal, A. J. Mayer, K. Nissim, and A. Wool. Firmato: A novel firewall management toolkit. In *IEEE Symposium on Security and Privacy*, pages 17–31, 1999.

[5] E. Bott and C. Siechert. *Microsoft Windows Security Inside Out: for Windows XP and Windows 2000*. Microsoft Press, 2003.

[6] J. Burns, A. Cheng, P. Gurung, D. Martin, Jr., S. R. Rajagopalan, P. Rao, and A. V. Surendran. Automatic management of network security policy. In *DARPA Information Survivability Conference and Exposition (DISCEX II'01)*, volume 2, Anaheim, California, June 2001.

[7] CERT. CERT advisory CA-2002-08. http://www.cert.org/advisories/CA-2002-08.html, Mar. 2002. web page fetched May 9, 2006.

[8] S. Chen, J. Dunagan, C. Verbowski, and Y.-M. Wang. A black-box tracing technique to identify causes of least-privilege incompatibilities. In *Proceedings of Network and Distributed System Security Symposium, 2005*, Feb. 2005.

[9] W. Clocksin and C. Mellish. *Programming in Prolog*. Springer-Verlag Inc. New York, 1987.

[10] M. Corporation. Access rights and access masks. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/secauthz/security/access_rights_and_access_masks.asp, Oct. 2005. web page fetched October 9, 2005.

[11] M. Corporation. ChangeServiceConfig. http://msdn.microsoft.com/library/default.asp?url=/library/e
us/secauthz/security/authorization_constants.asp, 2005.

[12] M. Corporation. ChangeServiceConfig. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/changeserviceconfig.asp, 2005.

[13] M. Corporation. Service security and access rights. http://windowssdk.msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/service_security_and_access_rights.asp, Oct. 2005. web page fetched October 9, 2005.

[14] T. M. Corporation. Introduction to CVE, the key to information sharing. http://cve.mitre.org/docs/docs-05/8-9-04_cve_intro_flyer.pdf, May 2006. Web page fetched on May 28, 2006.

[15] F. Cuppens and A. Mige. Alert correlation in a cooperative intrusion detection framework. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, page 202. IEEE Computer Society, 2002.

[16] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and expressive power of logic programming. In *IEEE Conference on Computational Complexity*, pages 82–101, 1997.

[17] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and expressive power of logic programming. *ACM Comput. Surv.*, 33(3):374–425, 2001.

[18] J. DeTreville. Binder, a logic-based security language. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, page 105. IEEE Computer Society, 2002.

[19] P. Finigan. Many ways to become DBA. http://www.insight.co.uk/files/presentations/Many%20ways%20to%20become%20DBA%20(Pete%20Finnigan).pdf, Oct. 2005. web page fetched May 9, 2006.

[20] W. L. Fithen, S. V. Hernan, P. F. O'Rourke, and D. A. Shinberg. Formal modeling of vulnerabilities. *Bell Labs technical journal*, 8(4):173–186, 2004.

[21] A. V. Gelder, K. Ross, and J. S. Schlipf. Unfounded sets and well-founded semantics for general logic programs. In *PODS '88: Proceedings of the seventh ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 221–230, New York, NY, USA, 1988. ACM Press.

[22] A. Griffiths. Syscall implementation could lead to whether or not a file exists. http://archives.neohapsis.com/archives/fulldisclosure/2003-q2/0052.html, Apr. 2003. web page fetched May 9, 2006.

[23] J. D. Guttman. Filtering postures: Local enforcement for global policies. In *Proc. IEEE Symp. on Security and Privacy*, pages 120–129, Oakland, CA, 1997.

[24] S. Hinrichs. Policy-based management: Bridging the gap. In *15th Annual Computer Security Applications Conference*, Phoenix, Arizona, Dec 1999.

[25] S. Jajodia, S. Noel, and B. O'Berry. Topological analysis of network attack vulnerabity. In V. Kumar, J. Srivastava, and A. Lazarevic, editors, *Managing Cyber Threats: Issues, Approaches and Challanges*, chapter 5. Kluwer Academic Publisher, 2003.

[26] P. M. Jeannette. An attack surface metric.

[27] S. T. King, Z. M. Mao, D. G. Lucchetti, and P. M. Chen. Enriching intrusion alerts through multi-host causality. In *The 12th Annual Network and Distributed System Security Symposium (NDSS 05)*, Feb. 2005.

[28] J. Lambert, M. Thomlinson, and V. Kumar. Microsoft Corporation, personal communication, July 2005.

[29] N. Li, B. N. Grosof, and J. Feigenbaum. Delegation Logic: A logic-based approach to distributed authorization. *ACM Transaction on Information and System Security (TISSEC)*, Feb. 2003.

[30] LinuxSecurity.com. SuSE: Kernel local root exploit. http://www.linuxsecurity.com/content/view/105569/112/, Dec. 2003. web page fetched May 9, 2006.

[31] J. Lorch, August 2005. Private communication.

[32] P. Ning, Y. Cui, and D. S. Reeves. Constructing attack scenarios through correlation of intrusion alerts. In *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, pages 245–254. ACM Press, 2002.

[33] N. I. of Standards and Technology. ICAT metabase. http://icat.nist.gov/icat.cfm, Oct. 2004. web page fetched on October 28, 2004.

[34] X. Ou. *A Logic Programming Approach to Network Security Analysis*. PhD thesis, Princeton University, 2005.

[35] X. Ou, S. Govindavajhala, and A. W. Appel. Mulval: A logic-based network security analyzer. In *14th USENIX Security Symposium*, 2005.

[36] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A blueprint for introducing disruptive technology into the internet. In *Proceedings of the 1st Workshop on Hot Topics in Networks (HotNets-I)*, Oct. 2002.

[37] C. R. Ramakrishnan and R. Sekar. Model-based analysis of configuration vulnerabilities. *Journal of Computer Security*, 10(1-2):189–209, 2002.

[38] P. Rao, K. F. Sagonas, T. Swift, D. S. Warren, and J. Freire. XSB: A system for efficiently computing well-founded semantics. In *Proceedings of the 4th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR'97)*, pages 2–17, Dagstuhl, Germany, July 1997. Springer Verlag.

[39] R. W. Ritchey and P. Ammann. Using model checking to analyze network vulnerabilities. In *2000 IEEE Symposium on Security and Privacy*, pages 156–165, 2000.

[40] M. E. Russinovich and D. A. Splomon. *Microsoft Windows Internals*. Microsoft Press, 2003.

[41] SANS. Handler's diary. http://isc.sans.org/diary.php?date=2005-12-31, Jan. 2006. web page fetched May 9, 2006.

[42] Secunia. Novell groupwise webaccess insecure default configuration. http://secunia.com/advisories/11119, Mar. 2004. web page fetched May 9, 2006.

[43] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J. M. Wing. Automated generation and analysis of attack graphs. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 254–265, 2002.

[44] W. R. Stevens. *UNIX Network Programming.* Prentice Hall, 1990.

[45] F. S. I. R. Team. Updated kernel packages fixes netfilter snmp nat memory corruption. http://www.frsirt.com/english/reference/12250, May 2005. web page fetched May 9, 2006.

[46] S. J. Templeton and K. Levitt. A requires/provides model for computer attacks. In *Proceedings of the 2000 workshop on New security paradigms*, pages 31–38. ACM Press, 2000.

[47] Y.-M. Wang, R. Roussev, C. Verbowski, A. Johnson, M.-W. Wu, Y. Huang, and S.-Y. Kuo. Gatekeeper: Monitoring auto-start extensibility points (ASEPs) for spyware management. In *Usenix LISA: 18th Large Installation System Administration Conference*, Nov. 2004.

[48] J. Whaley and M. Lam. Cloning-based context-sensitive pointer alias analyses using binary decision diagrams, 2004.

[49] M. Wojcik, T. Bergeron, T. Wittbold, and R. Roberge. Introduction to OVAL: A new language to determine the presence of software vulnerabilities. http://oval.mitre.org/documents/docs-03/intro/intro.html, Nov. 2003. Web page fetched on October 28, 2004.