PROTECTING PROGRAMS DURING RESOURCE ACCESS

A Dissertation in
Computer Science and Engineering
by
Hayawardh Vijayakumar

© 2014 Hayawardh Vijayakumar

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

May 2014
The dissertation of Hayawardh Vijayakumar was reviewed and approved* by the following:

Trent R. Jaeger  
Professor of Computer Science and Engineering  
Dissertation Advisor, Co-Chair of Committee

Patrick D. McDaniel  
Professor of Computer Science and Engineering  
Co-Chair of Committee

Bhuvan Urgaonkar  
Associate Professor of Computer Science and Engineering

Constantino Lagoa  
Professor of Electrical Engineering

Raj Acharya  
Professor of Computer Science and Engineering  
Head of the Department, Computer Science and Engineering

*Signatures are on file in the Graduate School.
Abstract

With the emergence of targeted malware such as Stuxnet and the continued prevalence of spyware and other types of malicious software, host security has become a critical issue. Attackers break into systems through vulnerabilities in network daemons, malicious insiders, or social engineering, and then attempt to *escalate privileges* to the administrator to gain complete control of the system by exploiting *local vulnerabilities*. Thus far, such local vulnerabilities have received little attention, and it has been taken for granted that any attacker break-in can easily be escalated to full control.

In this dissertation, we identify a class of previously disjoint local vulnerability attack classes that we call *resource access attacks*, and provide a framework to detect and defend against them. Programs have to fetch resources, such as files from the operating system (OS) to function. However, local adversaries such as spyware also share this namespace of resources, and can trick programs into retrieving an unintended resource using a variety of resource access attacks that make up 10-15% of vulnerabilities reported each year. Such attacks are challenging to defend for a few reasons. First, program checks to defend against such attacks cause a performance overhead, so programmers have an incentive to omit checks altogether. Second, there is a disconnect between the parties involved in resource access. On the one hand, due to this overhead, programmers omit checks under the expectation that the deployment’s access control policy will protect a subset of resources from adversaries. On the other hand, access control policies are framed by OS distributors and system administrators, who in turn have little idea about programmer expectations, causing mismatches with programmer expectations. Third, even when programmers check resource access, such checks are difficult to get right due to inherent races in the system call API. Previous work handles a subset of resource access attacks but in ad-hoc ways.

This dissertation takes several steps to address resource access attacks. First, we present a technique for automated evaluation of a program *attack surface* in its system deployment, where checks for resource access are required. Second, we present a technique that uses this attack surface to detect a subset of resource access attacks. We found more than 25 previously-unknown vulnerabilities across a variety of both mature and new programs in the widely-used Fedora and Ubuntu Linux distributions, proving the prevalence of such vulnerabilities. Third, we present the Process Firewall, a system to defend against resource access attacks in an efficient manner without requiring program code change. Fourth, we propose a technique to automatically derive the programmer-expected attack surface of a program, and generate Process Firewall rules to enforce that the only adversary-controlled resource accesses in the deployment are part of the
expected attack surface. The work in this dissertation thus provides a principled starting point to protect programs during resource access, thus reducing the vectors adversaries have to compromise a computer system.
Contents

List of Figures ix
List of Tables xi
Acknowledgments xiii

Chapter 1
Introduction 1
1.1 The Challenge of Defending Resource Access Attacks 2
1.2 Towards a Defense to Resource Access Attacks 3
1.3 Challenges in Defense 4
1.4 Thesis Statement 5

Chapter 2
Background 8
2.1 System Calls and Accessing Resources 8
2.2 Access Control on Resources 10
2.2.1 UNIX Discretionary Access Control 10
2.2.2 Mandatory Access Control 10
2.2.3 Obtaining Privilege 10
2.3 Resource Access Attacks 11
2.3.1 Untrusted Search Path 12
2.3.2 File/IPC Squat 13
2.3.3 Directory Traversal and PHP File Inclusion 14
2.3.4 Link Following and TOCTTOU races 14
2.3.5 Signal Races 15

Chapter 3
Related Work 17
3.1 System Defenses 17
3.2 Program Defenses 18
3.3 Detection of Program Vulnerabilities 19
Chapter 4
Integrity Walls: Finding Attack Surfaces from Mandatory Access Control Policies

4.1 Introduction ................................................. 20
4.2 Problem Definition .......................................... 22
4.3 Design ......................................................... 24
   4.3.1 Building Integrity Walls ............................. 24
   4.3.2 Identifying Attack Surfaces ......................... 26
4.4 Implementation .............................................. 28
   4.4.1 Integrity Wall Construction ......................... 28
   4.4.2 Identifying Attack Surfaces ......................... 29
   4.4.3 Finding Attack Surfaces in Interpreted Code ....... 31
   4.4.4 Enforcing Attack Surfaces ......................... 32
4.5 Evaluation .................................................. 32
   4.5.1 Policy Analysis ........................................ 33
   4.5.2 Runtime Analysis .................................... 34
      4.5.2.1 System TCB ..................................... 34
      4.5.2.2 Apache Webserver ............................. 35
      4.5.2.3 Secure Shell Daemon ......................... 35
      4.5.2.4 Icecat ........................................... 38
4.6 Related Work ............................................... 38
4.7 Conclusion ................................................ 39

Chapter 5
STING: Finding Name Resolution Vulnerabilities in Programs

5.1 Introduction ................................................. 40
5.2 Problem Definition .......................................... 43
   5.2.1 Name Resolution Attacks ............................ 43
   5.2.2 Detecting Name Resolution Attacks ................. 45
      5.2.2.0.1 Static Analysis ................................ 45
      5.2.2.0.2 Dynamic Analysis ............................. 46
      5.2.2.0.3 Symbolic Execution .......................... 46
   5.2.3 Our Solution .......................................... 46
5.3 Testing Model .............................................. 47
5.4 Design ....................................................... 48
   5.4.1 Attack Phase .......................................... 49
   5.4.2 Detect Phase .......................................... 52
5.5 Implementation ............................................. 54
5.6 Evaluation .................................................. 55
   5.6.1 Security Evaluation .................................. 55
      5.6.1.1 Finding Vulnerabilities ....................... 56
      5.6.1.2 Examples ....................................... 58
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.5.2 Name Flow Graph</td>
<td>99</td>
</tr>
<tr>
<td>7.5.3 Detecting Missing Filters</td>
<td>100</td>
</tr>
<tr>
<td>7.5.4 Detecting Presence of Binding Filters</td>
<td>102</td>
</tr>
<tr>
<td>7.5.5 Detecting Presence of Name Filtering</td>
<td>102</td>
</tr>
<tr>
<td>7.6 Enforcing Program Expectations</td>
<td>104</td>
</tr>
<tr>
<td>7.6.1 Protecting Accesses in $P$ at Runtime</td>
<td>104</td>
</tr>
<tr>
<td>7.6.2 Finding Adversary Accessibility $R$</td>
<td>105</td>
</tr>
<tr>
<td>7.6.3 Finding Deployment Attack Surface $S$</td>
<td>106</td>
</tr>
<tr>
<td>7.7 Proving Security of Resource Access</td>
<td>106</td>
</tr>
<tr>
<td>7.8 Implementation</td>
<td>107</td>
</tr>
<tr>
<td>7.8.1 Testing Programs</td>
<td>107</td>
</tr>
<tr>
<td>7.8.2 Enforcing Invariants</td>
<td>108</td>
</tr>
<tr>
<td>7.9 Evaluation</td>
<td>109</td>
</tr>
<tr>
<td>7.9.1 Implicit Programmer Expectations</td>
<td>109</td>
</tr>
<tr>
<td>7.9.2 Case Study: Apache</td>
<td>110</td>
</tr>
<tr>
<td>7.9.3 Process Firewall Enforcement</td>
<td>111</td>
</tr>
<tr>
<td>7.9.4 False Positives</td>
<td>112</td>
</tr>
<tr>
<td>7.9.5 Performance</td>
<td>113</td>
</tr>
<tr>
<td>7.10 Conclusion</td>
<td>113</td>
</tr>
</tbody>
</table>

Chapter 8

Conclusion 114

8.1 Future Directions 115

Bibliography 116
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Showing how resource access involves multiple parties.</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>Showing mismatch between program expectations and system deployment.</td>
<td>3</td>
</tr>
<tr>
<td>2.1</td>
<td>Showing how programs access resources through system calls, and how</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>adversaries can affect resources retrieved by programs. Shown shaded are</td>
<td></td>
</tr>
<tr>
<td></td>
<td>what the OS only knows – the access control policy, and the resource</td>
<td></td>
</tr>
<tr>
<td></td>
<td>hierarchy in the namespace. Red indicates adversary control.</td>
<td></td>
</tr>
<tr>
<td>2.2</td>
<td>Describing resource attributes.</td>
<td>9</td>
</tr>
<tr>
<td>2.3</td>
<td>Showing program code defending against various resource access attacks.</td>
<td>13</td>
</tr>
<tr>
<td>4.1</td>
<td>An example of a webserver process showing four input entry points A to D.</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>Objects and processes are shown with their MAC labels. The shaded entry</td>
<td></td>
</tr>
<tr>
<td></td>
<td>points define the program’s actual attack surface (i.e., access</td>
<td></td>
</tr>
<tr>
<td></td>
<td>adversary-controlled data), although filtering protects only entry point</td>
<td></td>
</tr>
<tr>
<td></td>
<td>D. Entry point B is only activated in certain configurations, but must</td>
<td></td>
</tr>
<tr>
<td></td>
<td>nonetheless be identified as part of the attack surface.</td>
<td></td>
</tr>
<tr>
<td>4.2</td>
<td>When the program makes a syscall, the SELinux function <em>avc_has_perm</em> is</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td>hooked through LSM hooks. If the calling function reads an object with a</td>
<td></td>
</tr>
<tr>
<td></td>
<td>label outside the wall for that process, then a log entry including the</td>
<td></td>
</tr>
<tr>
<td></td>
<td>entry point into the user program is generated to a relayfs file. A</td>
<td></td>
</tr>
<tr>
<td></td>
<td>userspace logging daemon collects this data into a log file, and a</td>
<td></td>
</tr>
<tr>
<td></td>
<td>summarizer analyzes this file to output the attack surface. The integrity</td>
<td></td>
</tr>
<tr>
<td></td>
<td>walls for each subject label are pushed into the kernel via a debugfs file.</td>
<td></td>
</tr>
<tr>
<td>4.3</td>
<td>A log entry from recording of untrusted operations. Two entry points for</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>Apache, with its location information.</td>
<td></td>
</tr>
<tr>
<td>5.1</td>
<td>Improper binding and improper resource attacks. A and V are adversary and</td>
<td>43</td>
</tr>
<tr>
<td></td>
<td>victim processes respectively.</td>
<td></td>
</tr>
<tr>
<td>5.2</td>
<td>Launching an attack.</td>
<td>48</td>
</tr>
<tr>
<td>5.3</td>
<td>Detecting a vulnerability.</td>
<td>48</td>
</tr>
<tr>
<td>5.4</td>
<td>STING is implemented in the Linux kernel and hooks on to the system call</td>
<td>54</td>
</tr>
<tr>
<td></td>
<td>beginning and LSM hooks. It has a modular implementation. We show here in</td>
<td></td>
</tr>
<tr>
<td></td>
<td>particular the interaction between user and kernel space.</td>
<td></td>
</tr>
<tr>
<td>5.5</td>
<td>Code from the GNU mail program in <em>mailutils</em> illustrating a squat</td>
<td>58</td>
</tr>
<tr>
<td></td>
<td>vulnerability that STING found.</td>
<td></td>
</tr>
</tbody>
</table>
5.6 Code from an X11 startup script in Ubuntu 11.10 that illustrates a race condition that STING found. .................................................. 59

6.1 Program code to defend against resource access attacks. .......................... 69
6.2 Process Firewall Design. Shaded steps are added by the Process Firewall. .... 72
6.3 The Process Firewall implementation: Shaded elements are specially developed for Process Firewall rule installation and processing. Unshaded elements in the Process Firewall proper are standard firewall components. ..................... 78
6.4 Comparing performance of link following checks in program or as Process Firewall rules as a function of path length n. ................................. 83
6.5 Comparing performance of Apache’s SymLinksIfOwnerMatch option with Process Firewall rules as a function of path length (n) and number of clients (c). We measured requests per second for Apache 2.2.22 on a Dell Optiplex 845 serving a static web page averaged over 30 runs for each parameter combination. ................. 84

7.1 Motivating example of resource access attacks using a typical processing cycle of a web server. ............................................................... 92
7.2 Demonstrating a mismatch between the expected and deployed attack surfaces. 95
7.3 Overview of the design of our system. .................................................. 97
7.4 Name flow graph for the example in Figure 7.1. ....................................... 99
7.5 Example about determining membership in P ........................................... 100
7.6 Determining whether a resource access in a resource flow graph should be in \( \mathcal{P} \). .................................................. 101
7.7 Inferring \( \mathcal{P} \) from knowledge of filtering ........................................... 101
7.8 Implementation of JIGSAW’s testing framework. ....................................... 108
7.9 Process Firewall rule templates. .......................................................... 109
7.10 Resource Flow Graph for Apache. Nodes that have the icon of adversaries beside them are those where we found adversarial control of resource access in our deployment. .................................................. 110
List of Tables

2.1 Resource access attack classes. Each class is shown with what the adversary controls to carry out the attack, its CWE class, and CVE entries reported per attack type. 12

4.1 Wall statistics for the system TCB types and Apache. Subject types correspond to processes, and object types correspond to OS objects (e.g., files) and processes. 33

4.2 Attack surface for the system TCB. The first column is the TCB type we consider, the second the total number of entry points for all programs running under that type, and the third the number of violating entry points that cross the integrity wall. Next, we list the specific binary with its overt violating entry points (Section 4.4.2) and the object type accessed that causes the entry point to be violating. We also identify vulnerabilities caused due to insufficient filtering at the overt entry points (we could not find any for the covert entry points). When multiple vulnerabilities are available for an entry point, the chronologically earliest is listed. Highlighted rows are discussed further in text. 36

4.3 Entry points in Bash scripts. 37

4.4 Apache entry points receiving low-integrity data 37

4.5 sshd entry points that may receive low-integrity data. Entry points marked with * are in the master part of the privilege-separated daemon. 37

5.1 Classes of name resolution attacks. 42

5.2 Table showing calls that signify acceptance, and therefore detection, for various attacks. write-like system calls are any calls that modifies the resource’s data or metadata (e.g., chmod). 53

5.3 Server programs installed on Ubuntu and Fedora. 55

5.4 Table showing the total number of distinct entrypoints invoking system calls performing namespace resolutions, number accessible to adversaries under an adversary model, and number of interfaces for which STING detected vulnerabilities. 56

5.5 Number and types of vulnerabilities we found. Race is the number of TOCTTOU vulnerabilities, where a check is made but the use is improper. A single entrypoint in Table 5.6 may be vulnerable to more than one kind of attack. 57

5.6 Number of vulnerable entrypoints we found in various programs, and the privilege escalation that the bugs would provide. 57

5.7 Micro-overheads for system calls showing median over 10000 system call runs. 61
5.8 Macro-benchmarks showing compilation and Apache-bench throughput and latency overhead. The standard deviation was less than 3% of the mean. .......................... 61

6.1 Resource access attack classes. For each attack class, we show its Common Weakness Enumeration (CWE [1]) number, and the number of reported vulnerabilities in the Common Vulnerability Exposure (CVE [2]) database. .......................... 66

6.2 Resource access attacks are caused by adversaries forcing victim processes to access unsafe resources instead of safe resources. Also shown is necessary process context to determine when these attacks apply (see Section 6.4). .......................... 68

6.3 Process Firewall rule language with an example. .......................... 79

6.4 The exploits tested against the Process Firewall. .......................... 81

6.5 Process Firewall rules discussed in text ($R1 - R11$) and templates used to generate rules ($T1, T2$). .................................................. 81

6.6 Microbenchmarks using lmbench. All results are in $\mu$s with percentage overhead in brackets. Standard deviation for all results was less than 1% for all our measurements. 95% confidence intervals for the non-fork results was at maximum 0.003, whereas for the fork-related results, the maximum was 0.3. Each column except the last incorporates optimizations of the previous column. DISABLED is the Process Firewall totally disabled, BASE with only the default allow rule, FULL our full rule base without any optimizations, CONCACHE with context caching optimization, LAZYCON with lazy context evaluation, and EPTSPC with entrypoint-specific rule chains. .................................................. 85

6.7 Benchmark overhead means are collected over 30 runs. Web1-1000 indicates ApacheBench latency on a LAMP system serving random database entries with 1 and 1000 concurrent clients respectively (L is latency and T is throughput). PF Base has default allow rules, and PF Full uses our full rule set. .......................... 85

6.8 Classification of entrypoints against the number entrypoint invocations (one invocation is one system call). .......................... 86

7.1 Adversaries control resource access to direct victims to adversary accessible resources when the victim expected an adversary inaccessible resource and vice-versa. Also shown are the common weakness enumeration [1] classes. .......................... 93

7.2 Statistics of program-wide resource accesses. Dev Tests show whether we used developer test suites or created our own. Impl. Exp. is the percentage of resource accesses ($|P|/|V|$) that are implicitly expected to be adversary-inaccessible. The last two columns show the number of instantiations of Invariant 7.1 and Invariant 7.2 in Section 7.3 for resource accesses in the program. *- We augmented the Apache test suite with additional tests. .......................... 110
Acknowledgments

Many acknowledgement sections begin with a pithy saying. This does not.

I arrived at Penn State more than six years ago. I had arrived for “higher” studies, but I had little idea of what I wanted to do. I took three courses in my first semester – one of which was the computer security course taught by my advisor-to-be, Dr. Trent Jaeger. By chance, he happened to be looking for students at that time, and took me on as one. It turned out to be one of the most fortunate quirks of fate ever in my life. I could not have completed my PhD if it were not for Dr. Jaeger. While this is tautologically true for every student and advisor, there are a variety of additional reasons for this in my case. Dr. Jaeger put up with my often erratic schedule and odd working hours. He still believed in me after I went through my first few years without any significant publications, at a point when even I had lost almost all hope. Whenever I hit a wall, I could always count on him for very insightful advice to progress forward. One of the most important lessons I learnt from him is the choosing of an important problem and tackling it in focussed ways, instead of jumping from one hot publishable problem to another. I am proud that my thesis has made a small dent in the circle of knowledge in systems security, and this was possible only due to him. For all these reasons, he has my deepest gratitude. I should also mention his ever-so-kind wife Dana who cooked us several delicious dinners!

I was also fortunate to have a great thesis committee. If I learnt how to work technically from Dr. Jaeger, from Dr. Patrick McDaniel I learnt the importance of confidence and presenting said work in a way the audience can relate to. From Dr. Bhuvan Urgaonkar, I learnt the importance of thinking insightfully – not just solving a problem in a limited way, but developing fundamental insights. His operating systems course was easily the best course I took while at Penn State. The other member on my committee, Dr. Constantino Lagoa, offered suggestions that significantly helped the quality of presentation of my thesis. I thank you all.

The PhD would have been very difficult, if not impossible, without a great set of co-students. I was fortunate to coincide with several brilliant students. Kevin Butler knew something, sometimes a lot, about literally everything. I was once discussing cricket, and was amazed when he came up with a statistic that even I did not know. Will Enck offered an insightful sounding board for ideas. Divya Muthukumaran’s positive energy effused onto everyone, and she always knew a variety of activities to do in State College. Devin Pohly was a human manual, and was my go-to man for any questions UNIX. I also learnt a lot from Damien Octeau’s methodical work ethic, Tom Moyer’s organisation, Nirupama Talele’s cheerfulness, and lively discussions with Srikar Tati. Two junior
students, Yuqiong Sun and Xinyang Ge, have promising careers ahead. I worked with Xinyang on a few projects towards the end of my PhD, and he was ever ready to offer suggestions and implement ideas. I am thankful for all your company. I must reserve special mention for three people. First, Josh Schiffman, a senior student of my advisor, served as a mentor. His clear and sharp insight while thinking about problems or solutions taught me a lot. He helped me significantly hone my writing skills. He also offered valuable advice at difficult times during my PhD. Second, Sandra Rueda. Sandra taught me an invaluable lesson in life – that being happy is unconditional. She also provided constant support when things looked down. Her extremely caring nature and outlook towards life gave me new perspectives. I am grateful for the fortune of having such a friend. Finally, Stephen McLaughlin, who has had the greatest (positive!) impact on me in my time at Penn State. Steve is at once a model student who has achieved great heights, and one who understood that whatever the academic work done, the real use is only judged by the impact on the masses. At times, when the pressure of the PhD became overwhelming, such thoughts reminded me of the bigger picture. I shall treasure this learning for the rest of my life. Steve and I spent many late nights slaving over our projects, and had innumerable conversations that made me much more intelligent and cognisant of a variety of topics.

Next, my family. My parents thought of their children as extensions of themselves. They lived for their children and sacrificed everything for them. In spite of great financial difficulties, they always made a priority of their children’s education. Words cannot begin to describe how fortunate I am to have my parents, nor shall I attempt to. Suffice it to say that in the hypothetical situation that I were born a million times, I would be lucky to get such parents even once. My brother (along with my father) is one of the most intelligent people I know. Though younger to me, he always seems to know what to do in any situation, and has supported me at various points during my PhD. Only after I came to the USA did I understand how much I took for granted from him. I also want to thank my grandparents. I especially learnt a lot of life lessons from my late grandfather V. Ramaratnam, who rose to great heights in spite of not having formal education, by virtue of his fierce determination and kindness of heart. I also thank my several aunts and uncles for their support during the PhD, especially my aunt and uncle in Florida who took me under their wing when I first arrived in the USA. Gratitude is also due to my cousins, especially Deepa and Prem, for their support.

There are a few other people I want to thank. First, my roommate Pandi (Dennis Arun). During our struggles with our PhDs, he provided an ear for my daily travails and constantly kept me cheerful. I do not know that I would have finished my PhD without his presence. I thank Srikanth V. R., with whom I had many conversations on the philosophy and meaning of life. I thank Santhosh for his help when I first moved to State College. I acknowledge the inspiration provided by my friends Balakrishnan Ranganathan and Gangadharan S.A., that led me to embark on the PhD in the first place. I also thank my undergraduate advisor, Prof. T. Srinivasan for the same reason. Finally, I had the fortune of having two excellent preceptors early in my life – Smt. Srimati Brahmanandam and the late Shri Balaganesan. They played a significant part in shaping me during childhood. There are scores of other people who have had a significant impact on me whom I have not mentioned here. To all of these people, I am grateful.

Penn State is set in the idyllic university town of State College. There are few better settings
in which to progress through such a long-term commitment as a PhD. I thank State College for providing me a near-perfect home for all these years. It shall always have a special place in my heart.

People are a product of their environment. With this in mind, I reserve my greatest thanks for last. I thank the universe for having brought such wonderful people in my life, and for having shaped my life in this way.

माता पिता गुरु देव

First comes the mother, then the father, then the teacher, and then the universe.
Dedication

To my parents, my brother, and the universe
Chapter 1

Introduction

Concurrent with the growth of applications has been the growth of malware. Malware is typically downloaded from the Internet, through social engineering or by exploiting browser vulnerabilities. Panda Security reported that around 50% of the computers scanned by their tool were infected with malware [3]. Thus, today’s computers run both malware and normal applications side-by-side. Malware is usually injected with limited user privileges and actively tries to obtain full privileges (e.g., of the administrator) by compromising other running privileged application programs through local exploits\(^1\). Once malware obtains full privileges, it can install keyloggers to steal passwords and credit card numbers or perform any of a number of malicious activities.

One opportunity for malware to attack victim programs is when victim programs perform resource access. A program performs a resource access each time it gets a new resource from the operating system (OS). Resources are objects managed by the OS, such as files, directories, interprocess communication (IPC) channels and signals. Programs are not self-contained code blocks that live in isolation. They need resources from the OS to function. Programs need to store and fetch configuration files, log files and temporary files from the OS. They need to fetch IPC channels such as sockets from the OS to communicate with other programs to obtain services. During such resource access, malware attempts to trick victim programs into accessing inappropriate resources\(^2\) for that resource access, thereby compromising the privileged programs. For example, malware can force the victim program to access a secret file through symbolic links when it was expecting a public file, thus potentially leaking secrets. It can create IPC objects and masquerade as a legitimate service provider, thereby supplying malicious input to the victim program to compromise its integrity. We call such attacks, where a process gets an inappropriate resource during resource

\(^1\)Local exploits, as opposed to remote exploits, are performed by malicious programs on the same physical machine as the victim program they exploit.

\(^2\)The definition of inappropriate is defined differently for different classes of resource access attacks.
access, resource access attacks.

1.1 The Challenge of Defending Resource Access Attacks

Resource access attacks are difficult to prevent because they involve multiple disjoint parties (Figure 1.1). First, there are programmers who write code. These programmers assume a certain subset of resources are potentially accessible to adversaries, and expect resource access attacks only while accessing this subset. Second, it is the OS that actually grants the program access to resources by checking its access control policy. OS distributors define access control policies, thereby determining which resources the adversary has access to. However, these OS distributors have little or no idea about the assumptions the programmer has made about adversarial access, so mismatches are possible. Finally, there are administrators who configure the program in the deployment. Configuration specifies the location of various resources such as log files. Thus, the administrator configuration too may not coincide with the programmer’s assumptions in code.

A second problem is that defensive checks hurt program performance. Programmers can conservatively assume that any resource is accessible to adversaries, and thus place defensive checks on every resource access. However, code to check properties of resources retrieved or to check input names is expensive. For example, the Apache webserver documentation \[4\] recommends switching off defensive resource checks during web page file retrieval to improve performance.

A third problem is that such defensive checks are very challenging to get right. As an example, the system call API that programs use for resource access is not atomic. Thus, to circumvent checks for symbolic link attacks, adversaries can present a normal file during victim program checks and change the file to a symbolic link before the victim program uses the file. As another example,
Programs often check resources indirectly by checking the names they use to fetch the resource, because they cannot convey equivalent resource constraints to the OS (e.g., adversary accessibility). However, both locating all sources of names, as well as input filtering of names to check them, is very hard to get right. Finally, many programmers are completely unaware of resource access attacks and fail to add checks altogether.

While no previous work has taken a unified view of resource access attacks, some work has addressed specific attacks, mainly time-of-check-to-time-of-use (TOCTTOU) race conditions. Broadly, there exist previous system-only defenses, defenses proposing program API changes, and work on detecting resource access attacks. System defenses include isolated namespaces [5] that prevent adversarial sharing and systems to solve specific resource access attacks [6, 7, 8, 9, 10, 11]. Sharing and interaction between adversarial and victim processes is often necessary for programs to function, so namespaces cannot be completely isolated. The above system-only defenses for resource access attacks are fundamentally limited because they do not know program intent and adversary accessibility [12], causing both false positives and negatives. Program defenses [13, 14, 15, 16] require programmers to specify constraints on resources they want to access for particular system calls. However, such solutions do not protect the body of existing code and do not completely address resource access attacks. Finally, there are both static [17, 18, 19, 20] and dynamic [21, 22, 23, 24, 25] work for detecting resource access vulnerabilities, so programs can be fixed before deployment. However, these use limited models of adversary access and are thus burdened with false positives and negatives. Further, due to the absence of standardized code checks to defend against resource access attacks, static analysis is mainly limited to detection of race conditions during resource access.

1.2 Towards a Defense to Resource Access Attacks

To motivate our solution to defend resource access attacks, we identify its causes. Figure 1.2 shows a program performing three resource accesses $r_1, r_2, r_3$. Assume the programmer expects adversary
interaction only for resource access $r_2$, and hence places code to defend attacks at that resource access. Let us call the resource access $r_2$ the expected attack surface. Due to performance penalty of defensive code for each resource access, the programmer does not place any code in $r_1, r_3$, and implicitly assumes that these are never accessible to adversaries. However, when the program is deployed in the system, the access control policy allows adversaries access to resources used in both $r_1$ and $r_3$. Let us call the resource accesses $r_1$ and $r_3$ the deployed attack surface. There is a mismatch between the expected and deployed attack surface, allowing the adversary to exploit the program through a resource access attack. Thus, the first cause for resource access attacks is unexpected adversary accessibility to resource access. Second, writing proper checks are difficult. Suppose that the defensive code the programmer places at $r_2$ has a race condition. Again, the adversary can exploit the program.

Traditional access control in OSes cannot prevent resource access attacks because it grants the same permissions for all system calls in a process. In Figure 1.2, the OS does not differentiate between the three different resource accesses made by the program – it gives the same permissions to all of them. An important characteristic of resource access attacks is that a resource that is unsafe for a particular victim system call is safe for some other victim system call. Thus, traditional access control cannot stop these attacks, as it does not consider each system call differently.

Capability systems implement an alternative access control mechanism to those above [26], where the programmer chooses the capabilities to present to the operating system for different system calls. However, the problem with capability systems in general is that they push the problem of access control back onto the programmers, presenting yet another API for them to solve the complex problems above. While we may yet produce an API for programmers to manage capabilities effectively, we propose instead to protect programs from resource access attacks given the current system call API.

**Security Goal.** Our goal in this work is to develop a mechanism to limit individual system calls to safe resources that would prevent resource access attacks. Considering access to each resource a capability, we would ideally compute the distinct set of capabilities for each system call that allows only safe resource access. However, such knowledge requires complete understanding of the program behavior, which is impractical. Instead, we remove capabilities corresponding to resources whose access would lead to a successful resource access attack.

### 1.3 Challenges in Defense

A solution to address resource access attacks must thus address the following challenges.

- First, the mechanism should enable removing capabilities to resources that lead to attacks
for individual resource accesses. To do this, it should support identifying individual resource access system calls within the program.

- Second, it should be possible to compute policy with minimal manual effort. That is, it should be possible to compute capabilities to resources that lead to attacks for individual resource accesses with minimal manual effort. One implication is that it should not require program code changes or additional annotations from programmers. There already exists a large body of code written to existing APIs. Providing new and better system call APIs to address resource access attacks will require programmers to change code, and will not protect existing code. Experience has shown that programmers still make mistakes even if better APIs are provided. For example, the open system call allows simple constraints on resources using the O_NOFOLLOW flag, which disallows following a symbolic link on the resource. In theory, this could be used to stop most symbolic link following resource access attacks, but these are still widespread.

- Third, the mechanism should be efficient. One of the reasons programmers omit checks is because having checks would hurt performance.

1.4 Thesis Statement

To provide protection against resource access attacks, this thesis proposes analyses on individual program resource accesses to determine which resource accesses would lead to attacks if under adversarial control, and enforces this protection efficiently for the program deployment’s calculated adversary accessibility.

The thesis of this work is thus:

\[
\text{Given a program running on an OS distribution, we can prevent every system call invoked by that program in that OS distribution from retrieving any resource that would lead to a resource access attack efficiently without the need for program modifications.}
\]

To prove our thesis, we address the following:

- **Insufficiency of existing program defenses.** In Chapter 4, we discuss the notion of program-specific adversaries and a deployment attack surface for a program, which defines the resource accesses that allow adversary access in the deployment \((r_1 \text{ and } r_2 \text{ in Figure 1.2})\). Examination of the attack surface for an Ubuntu Linux distribution led to several non-obvious attack surfaces, some missed by previous manual analysis, and two attack surfaces that had
previously unknown vulnerabilities. This shows that we cannot expect programmers to predict adversary accessibility for every system.

In Chapter 5, we use testing guided by the deployment’s attack surface to detect resource access vulnerabilities. Thus, we will detect vulnerable resource accesses that have either completely missing ($r_1$ in Figure 1.2) or insufficient ($r_2$ in Figure 1.2) defensive checks. This study was carried out on then-latest Ubuntu 11.10 and Fedora 15 distributions, using which we found 21 previously-unknown resource access vulnerabilities. The experiment demonstrated that there exist a variety of opportunities for malware to easily escalate privileges using resource access attacks in widely-used OSes.

- **An efficient mechanism to protect existing programs against resource access attacks.** In Chapter 6, we develop an enforcement mechanism called the *Process Firewall* to protect programs during resource access. The Process Firewall can introspect into the process to identify individual resource accesses, and apply corresponding constraints ($r_1, r_2, r_3$ in Figure 1.2). By developing a system mechanism, we can express richer resource constraints that programmers want and check these without adversely hurting program performance. In fact, we find that moving resource checks out from programs to our system actually improves performance. Using heuristics and runtime traces, we are able to generate policies automatically for our mechanism that was able to defend several attacks while not blocking legitimate functionality.

- **A technique to generate policy to protect programs with minimal manual effort.** In Chapter 7, we propose a technique to calculate the expected resource access attack surface ($r_2$ in Figure 1.2) by automatically inferring *programmer expectation* during the resource access. At each resource access, a programmer either expects adversarial control of resource access or not. For example, in Figure 1.2, the programmer does not expect $r_1$ to be controlled by an adversary. First, we provide a technique to automatically infer such expectation without requiring additional programmer annotations. Second, we propose two invariants based on expectation and prove that if the invariants are evaluated and enforced correctly, programs are completely protected from resource access attacks. Finally, we instantiate these invariants into Process Firewall rules that can be enforced at runtime. The only manual effort required is defining an appropriate adversary model, which we alleviate by pre-defining commonly used models. Our evaluation showed that rules produced defended against previously-known vulnerabilities for the programs we examined, and also detected and blocked two previously-unknown vulnerabilities and one default misconfiguration in the very mature Apache webserver, while preserving function.
The next chapters, Chapter 2, Chapter 3, provide background and discuss previous research on resource access attacks, respectively.
Chapter 2

Background

In this chapter, we discuss security problems that occur when processes fetch resources from the system.

2.1 System Calls and Accessing Resources

We first describe how processes access resources. Figure 2.1 shows a process $V$ communicating with the operating system (OS) to retrieve resources through system calls. Resources are objects managed by the system such as files, directories, signals, symbolic links, and IPC objects. System calls are requests to the system for service. A subset of system calls request access to resources (e.g., open), and these are our focus in this chapter. Processes request a resource referenced by a name (e.g., /etc/passwd), and the system resolves the name in a namespace (e.g., the filesystem) and delivers the resource if the system’s access control policy allows the process $V$ accesses to all resources involved. A single system call may access more than one resource. For instance, an open(/etc/passwd) requires accessing the root directory /, the directory etc, and the final file passwd. The directories used in resolution are referred to as bindings.

Processes access resources from the system in response to system calls either synchronously or asynchronously. For system calls such as open, the system responds synchronously with the resource. However, for resources such as signals, the response to processes is asynchronous. Processes show their interest in receiving signals by registering handler functions for signals (referenced by signal names), using the sigaction system call, and signals are received when they are actually available.

Resources have a set of attributes that are checked to defend resource access attacks. We shall discuss sample attributes of the most commonly used resource type—files—under POSIX.

\footnote{Henceforth, we shall refer to the operating system as simply the system.}
Figure 2.1: Showing how programs access resources through system calls, and how adversaries can affect resources retrieved by programs. Shown shaded are what the OS only knows – the access control policy, and the resource hierarchy in the namespace. Red indicates adversary control.

specifications [27]. POSIX specifications are used by UNIX-based systems such as Linux and Mac OS X, as also Windows NT. Figure 2.2 shows the \texttt{stat} structure that describes files (obtained through the \texttt{stat} system call). Similar structures exist for other resources. We describe fields that are commonly checked to defend resource access attacks below.

- **st_dev**: This is an identifier of a device containing the file. Devices are most commonly filesystems in secondary storage, but they can also be temporary filesystems or special filesystems.

- **st_ino**: This is an identifier of a resource within a device. Resources are represented by \texttt{inodes} in POSIX terminology; this field is thus the inode number. The combination of the device and inode number uniquely identifies a resource.

- **st_mode**: The mode bits encode two attributes. First, they encode the type of resource – a regular file, a directory, a symbolic link, an IPC object or special files. A symbolic link contains a name of another resource. Second, they indicate the read, write and execute (\texttt{rwx}) permissions on the resource for its owner, a group, and any other user not the owner or in the group.

- **st_nlink**: This represents the number of hard links to this resource. In POSIX filesystems, a single resource can be referenced by multiple names. This field tracks the number of names

```c
struct stat {
    dev_t     st_dev;     /* ID of device containing file */
    ino_t     st_ino;     /* inode number */
    mode_t    st_mode;    /* protection */
    nlink_t   st_nlink;   /* number of hard links */
    uid_t     st_uid;     /* user ID of owner */
    gid_t     st_gid;     /* group ID of owner */
    dev_t     st_rdev;    /* device ID (if special file) */
    off_t     st_size;    /* total size, in bytes */
    blksize_t st_blksize; /* blocksize for file system I/O */
    blkcnt_t  st_blocks;  /* number of 512B blocks allocated */
    time_t    st_atime;   /* time of last access */
    time_t    st_mtime;   /* time of last modification */
    time_t    st_ctime;   /* time of last status change */
};
```
referencing this resource. A resource is not deleted unless all its hard links are deleted.

- **st_uid**: This is the identifier of the owner of the resource. Every user on the system is assigned an identifier, and a file (typically `/etc/passwd`) stores the mapping between the user identifier, user name and password.

- **st_gid**: This is the identifier of the group that has special permissions on the resource (if any).

### 2.2 Access Control on Resources

When a process requests a resource through a system call, the system access control decides whether to allow access or not (Figure 2.1). Access control uses process and resource **labels** to decide whether to allow or deny access.

#### 2.2.1 UNIX Discretionary Access Control

In traditional UNIX access control, the process label is the user identifier (UID) the process is running as. This is the UID of the parent process. The resource label is the **rwx** permission bits in **st_mode**. UNIX access control is *discretionary*, meaning the owner of the resource can allow or deny access by changing the permission bits (resource labels) at her discretion. This is problematic, because a secure protection system requires that labeling state not change [28].

#### 2.2.2 Mandatory Access Control

In mandatory access control (MAC) systems, resource labels cannot change. SELinux [29] is a widely deployed MAC system for Linux. In SELinux, both processes and resources are assigned labels that are distinct from the UID and **st_mode** bits used by DAC. In SELinux terminology, resource labels are called types, and process labels are called domains. Resource labels are common for resources that are given similar access under SELinux. Thus, for example, many files in the directory `/etc` have the label `etc_t`.

#### 2.2.3 Obtaining Privilege

In some cases, untrusted users must be allowed access to high-integrity or high-secrecy files. The classic example is where a user has to modify her own password. This has to be stored in the

---

2The _t at the end of the label stands for type
high-integrity and high-secrecy password file. However, the user cannot be given arbitrary access to the password file.

To address this problem, such functionality is put into privileged programs, which can be invoked by untrusted users. In UNIX DAC, the `st_mode` bit has a flag called `setuid`, which means the program should set its UID to the owner (or group), instead of the user that started it (which is normally the case). SELinux has a similar concept of domain transitions to achieve the same. Since untrusted users can launch `setuid` programs, `setuid` programs have to protect themselves against adversarial environment variables and libraries.

### 2.3 Resource Access Attacks

Security problems occur when adversaries are able to force programs to access inappropriate resources (resources with inappropriate attributes) for system calls. As shown in Figure 2.1, this happens when adversaries can control: (1) the name the program uses to retrieve resources, or (2) the resource itself. As an example, consider a university webserver serving web pages of students. To authenticate users, it uses the university-wide password file `/etc/passwd`, to which it legitimately has access. An adversary can attempt resource access attacks on the webserver in two ways – by controlling the name, or the resource. First, she can request the file `../../etc/passwd`. If the webserver does not properly limit the name or validate the retrieved resource, it could end up accessing the high-secrecy password file when it meant to access a low-secrecy user web page, an inappropriate resource for that particular system call. Second, an adversary who is a student wanting to get access to the university-wide password file, can change her home page into a symbolic link to `/etc/passwd`. If the webserver does not check that the retrieved web page is a symbolic link to a secret file, it can again access the inappropriate resource.

To protect against these threats, programs limit the resources they receive to the proper set for a particular system call. They do this by either: (1) limiting the name (e.g., filename) used to fetch resources, or (2) rejecting inappropriate resources (e.g., reject symbolic links, block signals). However, problems occur when such checks are insufficient (or completely missing).

Table 5.1 lists well-known attack classes that are instances of resource access attacks, with their Common Weaknesses Enumeration [1] (CWE) category, their count in the Common Vulnerabilities and Exposures [2] (CVE) database, and whether the adversary controls resources directly and/or the name. As can be seen from Table 5.1, these attacks have continued to occur unabated over the last few years. Below, we examine causes for each attack. In short, each attack is caused either because checks on names and resources are: (1) assumed to be unnecessary and completely missing, or (2) wrong. Finally, we conclude with why such attacks cannot be solved given status quo.
### Table 2.1: Resource access attack classes. Each class is shown with what the adversary controls to carry out the attack, its CWE class, and CVE entries reported per attack type.

<table>
<thead>
<tr>
<th>Attack</th>
<th>Adversary Control</th>
<th>CWE class</th>
<th>CVE Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Untrusted Search Path</td>
<td>Name, Resource</td>
<td>CWE-426</td>
<td>109</td>
</tr>
<tr>
<td>Untrusted Library Load</td>
<td>Name, Resource</td>
<td>CWE-426</td>
<td>97</td>
</tr>
<tr>
<td>File/IPC squat</td>
<td>Resource</td>
<td>CWE-283</td>
<td>13</td>
</tr>
<tr>
<td>Directory Traversal</td>
<td>Name</td>
<td>CWE-22</td>
<td>1057</td>
</tr>
<tr>
<td>PHP File Inclusion</td>
<td>Name, Resource</td>
<td>CWE-98</td>
<td>1112</td>
</tr>
<tr>
<td>Symbolic/Hard Link Following</td>
<td>Binding</td>
<td>CWE-59</td>
<td>480</td>
</tr>
<tr>
<td>TOCTTOU Races</td>
<td>Binding, Resource</td>
<td>CWE-362</td>
<td>17</td>
</tr>
<tr>
<td>Total Resource Access CVEs</td>
<td>-</td>
<td>-</td>
<td>2894</td>
</tr>
<tr>
<td>Total CVEs</td>
<td>-</td>
<td>-</td>
<td>23337</td>
</tr>
<tr>
<td>% Total CVEs</td>
<td>-</td>
<td>-</td>
<td>12.40</td>
</tr>
</tbody>
</table>

#### 2.3.1 Untrusted Search Path

An untrusted search path occurs when a program searches for critical resources such as configuration files or libraries in adversary-accessible directories, and accepts these adversary-controlled low-integrity files. When programs start up, the library loader searches for and loads shared libraries of code. Next, they often search for configuration files in a variety of directories (i.e., paths). Some may then search for executable files to carry out additional functionality. However, searching for these critical files in insecure paths (e.g., public temporary directories) allows adversary-controlled files to be loaded, thereby completely compromising the program.

Untrusted search paths occur because of missing or insufficient checks on search path names. First, some programs simply assume their search path names to be correct, and do not have any checks at all. Secondly, even if programs do have checks, adversaries can control names in a variety of ways, and programs often miss filtering names from all these sources. We show this below with an example of an untrusted library load. Furthermore, programs do not check that the resource they fetch using the search path is adversary-inaccessible, because they trust the search path to be correct.

Figure 6.1(a) shows simplified code from `ld.so` when it loads a library. An adversary, when launching a `setuid` program, may supply malicious environment variables for `LD_LIBRARY_PATH`, forcing a victim program to search for untrusted libraries. In this example, an untrusted library load resource attack succeeds if the adversary forces the victim to access a low-integrity resource on line 8 where the victim program was expecting a high-integrity resource. To prevent this, `ld.so` unsets such environment variables on lines 3 and 4, builds a search path on line 6, opens the library...
/* fail if file is a symbolic link */
int open_no_symlink(char *fname)
{
    struct stat lbuf, buf;
    int fd = 0;
    lstat(fname, &lbuf);
    if (S_ISLNK(lbuf.st_mode))
        error("File is a symbolic link!");
    fd = open(fname);
    fstat(fd, &buf);
    if ((buf.st_dev != lbuf.st_dev) ||
        (buf.st_ino != lbuf.st_ino))
        error("Race detected!");
    lstat(fname, &lbuf);
    if ((buf.st_dev != lbuf.st_dev) ||
        (buf.st_ino != lbuf.st_ino))
        error("Cryogenic sleep race!");
    return fd;
}

/* serve file while protecting against directory traversal ("../") etc. */
void serve_file(int client_fd)
{
    char filename[512];
    read(client_fd, filename);
    /* Prevent traversal */
    if (filename contains "../") {
        replace("../", "/");
    }
    fd = open(filename);
    read(fd, buf);
    write(client_fd, buf);
}

/* ignore malicious env var */
int load_library()
{
    if ((uid!=euid) || (gid!=egid)) {
      /* setuid binary */
      unsetenv("LD_LIBRARY_PATH");
      unsetenv("LD_PRELOAD");
      }
    path = build_search_path();
    foreach path p {
      fd = open(p/lib);
      if (fd != -1) {
          mmap(fd); break; }
    }
}

/* serve file while protecting against directory traversal ("../") etc. */
void serve_file(int client_fd)
{
    struct stat lbuf, buf;
    int fd = 0;
    lstat(fname, &lbuf);
    if (S_ISLNK(lbuf.st_mode))
        error("File is a symbolic link!");
    fd = open(fname);
    fstat(fd, &buf);
    if ((buf.st_dev != lbuf.st_dev) ||
        (buf.st_ino != lbuf.st_ino))
        error("Race detected!");
    lstat(fname, &lbuf);
    if ((buf.st_dev != lbuf.st_dev) ||
        (buf.st_ino != lbuf.st_ino))
        error("Cryogenic sleep race!");
    return fd;
}

   \(\text{(d) Link Following and TOCTTOU races}\)

(a) Untrusted Search Path
(b) Directory traversal
(d) Link Following and TOCTTOU races
(c) PHP local file inclusion
(e) Signal races

Figure 2.3: Showing program code defending against various resource access attacks.

file on line 8, and maps it into the process address space on line 10. Unfortunately, there are a variety of other ways that adversaries can control names in search paths apart from environment variables – RUNPATHs in binaries, programmer bugs, and even dynamic linker bugs. It is difficult for programmers to anticipate and restrict all sources of adversarial names for library search paths. For example, a bug in the Debian installer (CVE-2006-1564) led to Apache module binaries being installed with insecure RUNPATH settings, which allowed insecure library loading.

2.3.2 File/IPC Squat

In a file/IPC squat, an adversary creates a resource such as a file or an interprocess communication (IPC) socket at a well-known name, and waits for a victim to use the resource. Consider a program that requires to connect to another process to communicate. To achieve this, one program (the server) creates a socket at a well-known name, and the other program (the client) connects to this socket. However, if an adversary creates the server socket instead of the expected server, and the client program (the victim) connects to the adversary, it may leak secrets or get compromised by adversarial input. A squatting attack is different from an untrusted search path, because the

\(^3\text{CVE-2011-0536, CVE-2011-1658}\)
name is correct; only, the resource at that name is wrong. Squatting attacks occur because victim programs assume that adversaries cannot access the resource referenced by the name.

2.3.3 Directory Traversal and PHP File Inclusion

In a Directory Traversal attack, the adversary (a client) controls a name that the victim program (a server) uses to retrieve a resource. For example, a webserver takes the name of a webpage to return to a client. The attack occurs when the adversary supplies names to access files not intended to be served by the server. Typically, such files are outside the directory tree that the server intends to restrict clients to; thus the adversary effectively traverses out of this directory tree.

Directory traversal attacks occur because input names are not properly filtered. Figure 6.1(b) shows code from a webserver that filters HTTP request names to restrict the web pages accessed. In this example, a Directory Traversal (or PHP File Include) resource attack succeeds if the adversary forces the victim to access a high-secrecy (e.g., password) file when the victim was expecting a low-secrecy (e.g., user HTML page) file on line 8. On line 5, this code replaces filenames that contain ../ with /, to prevent names that escape out of the directory root of the webserver (e.g., ../../etc/passwd). However, the adversary can simply supply ../../..../etc/passwd to bypass such input filtering. Although this is a simple example, there are many more ways to bypass input filtering, such as using unexpected input encoding [30].

In PHP file inclusion attacks, the adversary forces a PHP script to include and execute a low-integrity script, thereby allowing arbitrary code execution. PHP scripts often include files based on user input, but similar to directory traversal, they may not filter such names properly. Figure 6.1(d) shows an example of a PHP file inclusion attack. This is a simple script that includes the actual script to execute (line 2) depending on user input (line 1). However, it does not filter the name properly, and thus the adversary can include her own code. A variety of ways exist for adversaries to supply code [31] to later include. Here, the technique used is the script’s environment variables (line 4), the contents of which are populated by the adversary’s HTTP request.

Input filtering of input names has traditionally been a difficult problem to solve, and many programs are written by web developers unaware of such vulnerabilities.

2.3.4 Link Following and TOCTTOU races

In a symbolic or hard-link following attack, a victim program follows an adversary-controlled file which is a symbolic or hard link to an adversary-inaccessible (high-secrecy or high-integrity) file. The adversary thus fools the victim into accessing a file that the adversary does not have direct access to. Thus, the victim program ends up leaking secrets from a high-secrecy file or corrupting a high-integrity file determined by the adversary.
Time-of-check-to-time-of-use (TOCTTOU) races occur when programs perform checks on resources (e.g., use `lstat` check if it is a symbolic link to prevent link following), but the check is not atomic with the use (e.g., `open`). Thus, adversaries can change a resource to a symbolic link after the check, but before the use, rendering the check useless.

Checks for symbolic link attacks are often missing due to assumptions of adversary inaccessibility; however, checks are often wrong even when the programmer realizes there is possibility of adversary access. Figure 6.1(c) shows code from a program trying to defend itself against symbolic link attacks. On line 3, an `lstat` checks if the file is a symbolic link. If not, on line 6, the file is opened to read. However, there is a race condition possible between lines 3 and 6. A time-of-check-to-time-of-use (TOCTTOU) attack \[32, 17\] can be successful if the adversary forces the victim to access a different file on line 6 than its previous “check” on line 3. An adversary scheduled after line 3 but before line 6 could change the file to a symbolic link. To defend against this, an `fstat` is performed on line 7, and the inode and device numbers that uniquely identify a file on a device are checked with `lstat` to make sure the file checked is the one opened on lines 8 and 9. However, even this is insufficient. Olaf Kirch \[33\] showed a “cryogenic sleep” attack, in which an adversary could put a `setuid` process to sleep before line 6 but after line 3, and wait for the inode numbers to recycle, thus passing the checks on lines 8 and 9. To defend this, an additional `lstat` is done on line 11, and the inode and device numbers compared again. So long as the file remains open, the same inode number cannot be recycled. Finally, Chari et al. \[8\] showed that such similar checks must be done for each pathname component, not only the final resource, and proposed a generalized `safe_open` function that allows following of links so long as the link points to an adversary’s own files and not the victim’s files. Unfortunately, `safe_open` is used by very few programs currently (Postfix uses a weaker version) and has some false negatives \[4\].

### 2.3.5 Signal Races

Signal races occur when adversaries are able to deliver signals to victim processes that are modifying global process state, and the signal handler itself also uses this global state. Examples of functions that modify global state are heap memory allocation functions `malloc` and `free`. Signal handlers that use such functions should not be called when program code itself is executing such functions, or programs can crash or even execute arbitrary code \[34\]. The one solution is to block signals during program execution that modifies global state used by signal handlers, if an adversary is able to deliver signals at arbitrary times.

However, programs often do not have code to block such problems because they assume ad-

---

\[4\] In `safe_open`, an adversary A cannot trick a victim V into opening its own files after following A’s links, but can trick V into accessing another victim B’s files through A’s symbolic links.
versaries cannot deliver signals. Figure 6.1(e) shows simplified code from Sendmail v8.11.3. It has a single signal handler registered for multiple signals. On calling the handler, it logs some information using `syslog`, modifies some global variables, and exits. However, `syslog` is not a async-signal safe function [34], as it modifies shared data in non-atomic ways. In this example, a non-reentrant signal handler resource attack succeeds if the adversary delivers a signal to a victim program already handling a signal. If programs are installed `setuid` in the OS, then contrary to programmer assumption, adversaries can deliver signals. Zalewski [34] shows how adversaries can exploit such code by quickly sending a `SIGHUP` and `SIGTERM` in succession, forcing the non-reentrant signal handler to re-enter, thereby crashing the program or even executing malicious code.
Chapter 3

Related Work

In this section, we list previous research on these classes of attacks. While no work has taken a unified view of all these attack classes, they address specific issues for certain attacks. We cover three broad categories of related work that protect programs from resource access attacks. First, we cover system-only defenses, that attempt to protect programs without requiring program cooperation or change. Second, we cover defenses that do require program changes to use new system call APIs the system provides. Finally, a third approach is to detect and fix resource access attacks before deploying programs.

3.1 System Defenses

Traditional Access Control. First, traditional access control in OSes cannot solve resource access attacks. Access control grants permissions to processes as a whole, and does not differentiate different resource requests. Resource access attacks force the victim process to access an inappropriate resource for one particular request, but that resource is valid for another request.

Isolated Namespaces. Isolated namespaces eliminate resource sharing between adversary and victim processes. Per-process namespaces have recently been introduced in Linux [5], which are often used to maintain compatibility, but they have also been applied to create per-process temporary directories to prevent unwanted sharing. However, sharing between victim and adversary processes is often necessary for functionality.

Prevent TOCTTOU race conditions and link following. A lot of work exists on system defenses to solve TOCTTOU race conditions. The basic idea of these systems is to externally enforce atomicity between “check” (e.g., stat) and “use” (e.g., open) system calls. Some mechanisms are implemented in the process or as library extensions [35, 36, 37, 38] and some as kernel extensions [6, 7, 9, 10, 11]. In general, such methods have been found to be limited [39] because user-space defenses
lack insight into the resource properties only known by the system such as adversary accessibility, and system defenses lack information about the the resource constraints for particular program requests \[12\]. Chari et al. \[8\] propose a library function `safe_open` that aims to prevent link following attacks, and thus, TOCTTOU race conditions also. Their idea is that victim processes should not end up with a high-integrity or secrecy resource after having been in a low-integrity directory, in a single name resolution. While their concept is sound, their implementation uses only the DAC adversary model, and even in that case, has both false positives and negatives. Moreover, their solution adds expensive checks; each `open` call performs at least 4 additional system calls for each pathname component (i.e., directory) in the filename.

### 3.2 Program Defenses

**More general resource access APIs.** Researchers have realized that current OS system call APIs for resource access do not allow programs to express resource constraints to the OS for protection. Decentralized information flow control (DIFC) systems have mechanisms that allow programs to express resource constraints in terms of information flow control to the OS, which enforces it for the application. Completely new OSes have been proposed \[40, 41, 42\], as also retrofitting such mechanisms to the Unix system call API \[13\]. However, expressing resource access constraints in terms of information flow is not straightforward, and these systems require programs to be rewritten using new APIs and also partitioned into privileged and unprivileged parts.

**APIs to prevent race conditions.** Several systems \[43, 44, 45, 46, 14\] have attempted to retrofit atomicity in resource access onto the system call API, for programs to express their constraints. For example, TxOS presents an API for programs to begin and end transactions, which are executed atomically in the OS, enabling prevention of TOCTTOU races and signal races. However, these again require program changes, and do not address resource access attacks in general.

**Direct access to resource through capabilities.** One way to address adversary control of resources is to have a direct reference to the proper resources. Capability systems \[26\] restrict processes to resources to which they have a reference, preventing adversaries from redirecting resource retrieval. Capability systems can also prevent confused deputy attacks by requiring clients to supply capabilities, rather than names, for the resources that they wish to access \[47\]. Pure capability systems \[48, 49, 50, 51\] do not require namespaces at all. Mazieres and Kaashoek propose to solve some namespace resolution attacks by requiring processes to use capability-like objects (file descriptors and credentials) on system calls, where possible \[16\]. Eliminating namespaces is difficult however, because of their convenience. For example, the Capsicum capability system obtains its
capabilities from the system on initiation [15], which can be subverted by adversaries.

### 3.3 Detection of Program Vulnerabilities

A third defense is to detect resource access vulnerabilities in programs and fix them before programs are deployed. There are static and dynamic detection techniques, particularly for TOCTTOU attacks. Static analyses [17, 18, 19, 20, 52] look at the program in isolation, and therefore produce a large number of possible vulnerabilities. However, not all of these are accessible to adversaries. Previous runtime analyses [8, 21, 22, 23, 24, 25] have limited adversary models, and in addition, even if they find adversary-accessible points, they cannot tell if the program defends itself against these attacks, which requires an active adversary that launches attacks. Detection is discussed in more detail in Chapter 5.
Chapter 4

Integrity Walls: Finding Attack Surfaces from Mandatory Access Control Policies

4.1 Introduction

Protecting host system integrity in the face of determined adversaries remains a major problem. Despite advances in program development and access control, attackers continue to compromise systems forcing security practitioners to regularly react to such breaches. With the emergence of more sophisticated malware, such as Stuxnet \[53\], malware has begun to target program entry points that are left undefended, thus exacerbating the problem.

While security practitioners may eventually learn which entry points must be defended over a software’s lifetime, new software and configuration options are frequently introduced, opening additional vulnerabilities to adversaries. The application developers’ problem is to identify the program entry points accessible to adversaries and provide necessary defenses at these entry points before the adversaries use these to compromise the program. Unfortunately, this is a race that developers often lose. While some program vulnerable entry points are well-known (mostly network), the complexity of host systems makes it difficult to prevent local exploits should attackers gain control of any unprivileged processing. For example, the OpenSSH daemon was reengineered to defend two entry points in the privileged part through which several vulnerabilities were exploited \[54\], but a third entry point also existed that was vulnerable to any user processes \[55\]. The question we explore in this chapter is whether the program entry points accessible to adversaries can be found proactively, so defenses at these entry points can also be developed proactively.
Prior efforts to better understand how adversaries can access programs focus either on system security policies or program entry points, but each provide a limited view. With the widespread introduction of mandatory access control (MAC) enforcement in commercial operating systems (OSes) [56, 57, 58], it is possible to determine the subjects in the MAC policy that may be influenced by adversary-controlled data [59, 60, 61, 62]. Also, methods have been developed to compute attack graphs [63, 64, 65], which generate a sequence of adversary actions that may result in host compromise. However, these methods treat programs as black boxes, where any program entry point may be able to access either adversary-controlled data or benign data. As these accesses are not connected to the program entry points that use them, it is difficult to know where exactly in the program or even the number of points in the program that access adversary-controlled data.

From the program’s perspective, researchers have argued for defenses at a program’s attack surface [66], which is defined by the entry points of the program accessible to adversaries because they may access adversary-controlled data. Unfortunately, programs often have a large number of library calls signifying potential entry points, and it is difficult to know which of these are accessible to adversaries using the program alone. Some experiments have estimated attack surfaces using the value of the resources behind entry points [67, 68]. However, if the goal is simply to take control of a process, any entry point may suffice. While researchers have previously identified that both the program and the system security policy may impact the attack surface definition [66], methods to compute the accessibility of entry points have not been developed.

In this chapter, we compute the attack surface entry points for programs relative to the system’s access control policy, thus overcoming the above limitations of focusing only on either one, and enabling accurate location these entry points. First, we propose an algorithm that uses the system’s access control policy to automatically distinguish adversary-controlled data from trusted data based on the permissions of each program’s adversaries. This constructs what we call a program’s integrity wall. We use the system’s MAC (as opposed to DAC) policy for this purpose because it is immutable, thus preventing the permissions of adversaries from changing dynamically. To determine adversary access using MAC policies, past work leveraged program packages to define what is trusted by programs [62, 69]. However, the subjects associated with packages are not all necessarily trusted equally. For example, the Apache package includes user-defined CGI scripts, and clearly these cannot be trusted by the Apache webserver. Instead, we propose a novel approach for computing per-program adversaries based on the ability to modify the program’s executable content.

Second, we construct a runtime analysis to collect the program entry points that access objects outside its integrity wall. Fundamental to the runtime analysis are techniques to find the program

---

1 Adversary-controlled data lies outside the program’s wall, and trusted data inside the wall.
entry points (instructions in the program’s binary), that receive adversary-controlled inputs. Our
techniques support both binary code and several interpreted languages (e.g., Bash, PHP, Python)
to enable system-wide computation of program attack surfaces. Where available, we use developer
test suites for application programs; these often test multiple program configurations as well, using
which we were able to associate certain entry points with configuration options that enabled them.

We evaluate a prototype runtime analysis tool on Ubuntu Linux LTS 10.04.2, using the distribution’s
SE Linux MAC policy to build integrity walls and the application packages’ test suites to
guide the runtime analysis to collect attack surfaces. The tool found that this distribution’s trusted
computing base (TCB) processes have 2138 entry points, but only 81 attack surface entry points
that an adversary could potentially exploit. While examining the system TCB attack surface, we
found a previously unknown vulnerability in one entry point in a script that has been present
in Ubuntu for several years. Detailed analyses of Apache and OpenSSH found an entry point in
OpenSSH missed by a previous manual analysis, and demonstrates the ability of our tool to asso-
ciate entry points with configuration options and find subtle, easily overlooked entry points. Also,
analysis of a recent program, the Icecat web browser, revealed a previously unknown untrusted
search path vulnerability, demonstrating the value in applying this analysis proactively on new
programs.

In summary, we make the following contributions:

• We propose an algorithm to construct an “integrity wall” for applications based on MAC
  policy and a runtime technique to precisely identify attack surface entry points in programs
  (including interpreted scripts) using the constructed wall,

• We present results of the attack surface for the system TCB for Ubuntu 10.04.2 and some of
  its applications, which helped uncover two previously unknown bugs, one present for several
  years in Ubuntu, showing the value of locating attack surfaces before an adversary does.

4.2 Problem Definition

The aim of this chapter is to identify program entry points that access adversary-controlled objects.
If an adversary can modify an object that is accessed by a program entry point that expects only
safe objects, the running program can often easily be compromised.

Consider some of the entry points in a typical webserver program shown in Figure 4.1. During
development, the application developers have realized that entry point $D$ receives adversary input
via the network, making $D$ part of the program’s attack surface. As a result, the developers
have added defenses to filter input at $D$. The program is then deployed in a system under a
particular access control policy and configuration that allows the accesses shown. Entry point $A$
reads a configuration file, and under any reasonable MAC policy the adversary cannot access that file. Thus, A is not part of the attack surface. Suppose the administrator has enabled the UserDir configuration directive, allowing users to define their own HTML files (e.g., ~/public_html). Then, entry point B receives adversary-controlled input (user-defined web pages), but the developers have overlooked this entry point because it is opened only under certain configurations and, moreover, not an obvious threat. Finally, entry point C reads in module library files (e.g., ModCGI) to serve a request. While this entry point is supposed to read in files labeled lib_t from /usr/lib, it has an untrusted search path bug that first searches for files in the current working directory. Hence, C exercises permissions that it is not meant to, and reads the user’s public_html directory for libraries. An adversary can easily take control of the web server and gain its privileges if she plants a malicious module library in that directory. Thus, the adversary has found two entry points B and C into the program not anticipated by the developers.

In practice, we have seen much the same pattern. After the Apache webserver was launched in 1998, vulnerabilities were found at entry points that access log files, CGI script output, user-defined HTML files, and user-defined configuration files over a period of six years. We believe that locating the attack surface proactively enables: (1) verification of where input filtering is necessary to protect the program, such as B and D, that have to handle adversary input, and (2) identification of entry points that should not be part of the attack surface, such as C, so the program or policy can be fixed. Our evaluation (Section 6.6) found two previously unknown vulnerabilities, one for

\footnote{CVEs 1999-1206, 2001-1556, 2002-1850, 2004-0940, 2004-2343 respectively}
each of the above cases.

While classical security principles stress the importance of recognizing where programs may receive adversary input (e.g., Clark-Wilson [70] requires entry points to upgrade low-integrity data), we lack systematic techniques to identify these program attack surfaces entry points. Recent work has focused on how programmers can express their attack surfaces to systems for enforcement [71, 13, 41] or for further testing [66, 67]. However, this work assumes developers already have a complete understanding of their program’s attack surfaces, which experience and our results show to be incorrect. Our results demonstrate that both mature and new programs may have undefended attack surface entry points, and many entry points are accessible to adversaries in subtle ways.

**Assumptions.** Our work calculates attack surface entry points in programs and not the attack surface of the OS kernel itself. Thus, we assume the OS kernel to be free from vulnerabilities that a local attacker can exploit. Further, we assume that the reference monitor enforcing access control in the OS enforces a MAC policy, and satisfies complete mediation and is tamperproof [72]. This implies that the only way for local adversaries to attack programs is through rules specified in the OS MAC policy.

### 4.3 Design

Calculating the attack surface has two steps. First, for a particular subject (e.g., `httpd.t`), we need to define its adversaries (e.g., `user.t`), and locate OS objects under adversarial control (e.g., `httpd_user_content_t`). We do this using the system’s MAC policy. Next, we need to identify the program entry points (e.g., entry points `B; C; D`) that access these adversary-controlled objects. Statically analyzing the program cannot tell which permissions are exercised and which OS objects accessed at each entry point, and thus we use a runtime analysis to locate such entry points. In this section, we detail solutions to these two steps.

#### 4.3.1 Building Integrity Walls

A program may receive many inputs. However, not every input into a program is necessarily under the control of adversaries. A program depends on (i.e., trusts) some inputs (e.g., `etc.t` and `lib.t` in Figure 4.1), whereas it needs to filter (i.e., protect itself from) other inputs (e.g., `httpd_user_content.t` in Figure 4.1). Our insight is that the system’s MAC policy enables differentiation between those OS objects that a subject depends on and those OS objects that it needs to filter. This is simply because a properly designed MAC policy limits the modification of OS objects that a particular subject `s` depends on only to subjects that are trusted by `s`, and any other object is untrusted and needs to be filtered on input. Thus, if we identify the set of subjects trusted by `s`,
we can then derive the trusted and untrusted objects for $s$ from the MAC policy.

**Integrity Wall Approach.** The observations that we use to calculate the set of trusted subjects are outlined below. First, a process fundamentally depends on the integrity of its executable program file. Thus, a subject in the MAC policy has to trust other subjects that have the permission to modify its executable program file$^3$, called *executable writers*. While we could expand this definition to include all code used by a process, such as libraries, we find that the set of labels for approved libraries are ambiguous and that these are covered by the other cases below.

Second, a process depends on the integrity of its underlying system. If the kernel can be compromised, then this process can be trivially compromised. Thus, all subjects depend on the subject labels with permission to modify any kernel objects, called *kernel subjects*. Naturally, each subject also depends on the executable writers of the kernel subjects as well. This combination forms the system’s *trusted computing base* (TCB).

Third, several applications consist of multiple distinct processes, some of which are trusted and some not. For example, `htpasswd` is a helper program for Apache that maintains the password file `.htpasswd`. Intuitively, Apache depends on this program to maintain the password file properly. On the other hand, Apache should filter inputs from user-defined CGI scripts. We state that a subject label $s$ depends upon a *helper subject*, if: (1) the two subject labels are part of the same application (e.g., package) and (2) the helper subject’s executable writers are in the application or trusted by $s$. Identifying that two subject labels are part of the same application is often easy because MAC policies are now written per application (e.g., there is an SELinux policy module for Apache).

**Integrity Wall Algorithm.** The problem is thus to compute for each subject a partition of the set of MAC policy labels $P$ based on whether the subject depends on the label or not, based on the three criteria above, forming that subject’s integrity wall. An integrity wall for a subject $s$ is a partition of the set of labels$^4$ in the system policy $P$ into sets $I_s$ and $O_s$, such that $s$ depends on labels in $I_s$ ("inside the wall"), and filters inputs from labels in $O_s$ ("outside the wall").

The integrity wall derivation computes $I_s = P - O_s$ from MAC policy containing relations $x \text{Write} y$ and $x \text{Writex} y$, which mean subjects of label $x$ can write objects of label $y$ and subjects of label $x$ can write executable file objects of subject $y$, respectively.

1. The *kernel subjects* $K \subseteq P$ of a system are:
   $K = \{s1 \mid \exists o \in \text{Kernel}(P), \text{where} (s1, o) \in \text{Write}\}$

$^3$Typically, MAC policies are designed by assigning permissions to each executable independently, which means there is often a one-to-one mapping between subject labels and executable files.

$^4$The set of object labels includes the set of subjects labels in the policy, but not vice versa. Also, we use the terms subject and object for subject label and object label from this point forward when unambiguous.
2. The trusted computing base $T \subseteq P$ of a system is:
\[
T^0 = K; \\
T^i = T^{i-1} \cup \{s2 \mid \exists s1 \in T^{i-1}, (s2, s1) \in \text{Writex}\}; \\
T = \bigcup_{i \in \mathbb{N}} T^i
\]

3. The executable writers $E_s \subseteq P$ for a subject $s$ are:
\[
E_s^0 = s; \\
E_s^i = E_s^{i-1} \cup \{s2 \mid \exists s1 \in E_s^{i-1}, (s2, s1) \in \text{Writex}\}; \\
E_s = \bigcup_{i \in \mathbb{N}} E_s^i
\]

4. The helper subjects $H_s \subseteq \text{App}(s)$ for a subject $s$ are:
\[
H_s = \{s1 \mid (s1 \in (\text{App}(s) - \{s\})) \land (E_s \subseteq (\text{App}(s) \cup E_s))\}
\]

5. The trusted subjects $T_s \subseteq P$ for a subject $s$ are:
\[
T_s = T \cup E_s \cup H_s
\]

6. The trusted objects $I_s \subseteq P$ for a subject $s$ are:
\[
I_s = T_s \cup \{o \mid \exists s1 \in (P - T_s), (s1, o) \in \text{Write}\}
\]

First, we compute the kernel subjects (i.e., subjects with Write access to $\text{Kernel}(P)$ objects) and TCB for the system at large. The TCB is derived from a transitive closure of the writers of the kernel subjects’ executables (Writex). Then, for each subject we compute its executable writers (again, using transitive closure) and its helper subjects. Helper subjects must be part of the same application ($\text{App}(s)$) as the target subject $s$ and can only be modified by a subject outside the application that is trusted by $s$. Thus, $\text{htpasswd}$ is an Apache helper, but user scripts are not, as their executable is written to by an untrusted subject (the user). Finally, we collect the trusted objects for the subject: the set of objects that are only modified by the trusted subjects. A problem is that some objects are written only by trusted subjects, but are known to contain adversary-controlled data, such as log files. We assume these objects to be untrusted. More such cases are discussed in the evaluation.

This method computes the object labels inside the integrity wall for a subject label, and all other objects labels are outside the integrity wall for that subject. Access to objects outside the wall will be the focus in building each program’s attack surface.

### 4.3.2 Identifying Attack Surfaces

Using an integrity wall for a subject, we can find the attack surfaces of all programs that run under that subject. As noted, it is impractical to identify these entry points statically, because any system
call is authorized to access any object to which the program’s subject is authorized. Therefore, we propose a runtime analysis to locate entry points. Runtime analysis provides a lower-bound for the number of entry points in an attack surface, but nonetheless, we have found many non-trivial attack surfaces with recent vulnerabilities and we identified new vulnerabilities (Section 6.6).

The most important design decision is to define what an entry point is. To find the program entry points, we obtain the process’s execution stack at the time of the system call. Consider a program performing a system call that receives input, through a stack of function calls $F_1, F_2, \ldots, F_n$, where $F_i$ calls $F_{i+1}$. The entry point into the program occurs at the greatest index $i$ where $F_i$ is not trusted to filter all input. That is, we may trust $\text{libc}$ to protect itself from untrusted input, making the caller of $\text{libc}$ the entry point. This is often, but not always, the program executable instruction that invoked the library call.

Developing a runtime analysis for identifying program attack surfaces must meet the following requirements.

1. All security-sensitive operations from all processes must be mediated,
2. The subject and object labels of each operation are available, and
3. The context of the process execution (e.g., the instruction pointer and process stack) is available.

First, both user-level and kernel-level mechanisms have been designed to mediate system calls, but the use of kernel-level mediation is preferred in this case because: (1) multiple security-sensitive operations and objects may be accessed in one system call, and it requires significant parsing effort in user-space to find them all accurately and (2) all processes can be mediated in a single location for low overhead. In several modern operating systems, reference monitors have been implemented to mediate all security-sensitive operations [56, 58, 57], which we extend to detect accesses of objects outside the integrity wall.

Second, we need to know the subject and object labels of the operation to determine if this operation is outside the integrity wall for that subject. The label information is obtained from the reference monitor module enforcing system security (e.g., SELinux [29] and AppArmor [73] for Linux). We use this information to determine whether the subject is accessing an object outside its wall based on the trusted objects $I_s$ or its set complement $O_s$.

Third, when an untrusted object is accessed, we find the process’ user stack at the time of the call to find the entry point. The main challenge is to search up the stack to find the first stack frame that is not trusted to filter all inputs. Each frame must be mapped to its code file to determine whether it is trusted for all inputs. We use the virtual memory mappings to determine the code
file, and we maintain a table of those that are fully trusted. The specific mechanism is described in the implementation.

Finally, we log entry points to user-space, so they can be collected and analyzed. A log entry record consists of the subject and object labels of the operation and the entry point in the process' user stack at the time of the operation.

4.4 Implementation

In this section, we describe how we implemented our design on Ubuntu 10.04.2 LTS Desktop Edition running a Linux 2.6.35 kernel, with SELinux as the MAC enforcement mechanism. We first describe our implementation to construct the integrity wall for subjects, and then our modification of the Linux kernel to log the entry points at runtime. We also explain how we extended our system to deal with interpreters. Our modifications added 1189 lines of code to the Linux 2.6.35 kernel: 588 lines of code for interpreter processing and the rest to fetch the stack backtrace, detect if the operation is untrusted, and log fresh entries to userspace.

4.4.1 Integrity Wall Construction

We implement the design described in Section 4.3.1. We implement the algorithms for each step in XSB/Prolog. In total, the algorithms required 101 Prolog statements, 77 of these were for parsing the SELinux policy, and the rest for the wall generation. The main input is the system’s SELinux MAC policy, which consists of a set of policy modules for individual Linux applications deployed on the system\(^5\). We describe implementation of the algorithms in terms of TCB computation and the subject label’s integrity wall computation.

To construct the TCB of the system, we manually identify a set of 13 kernel objects (\texttt{Kernel(P)})\(^5\), write access to which could directly compromise the kernel (e.g., \texttt{/dev/kmem}). Using Steps 1 and 2 of Section 4.3.1, the SELinux policy and the kernel objects are used to identify the set of subjects that can write to these kernel objects and to perform a transitive closure of the writers of the binaries of the kernel subjects. The SELinux policy identifies all write operations (\texttt{Write}) and the objects that may be executables for a subject for computing \texttt{WriteX}. SELinux defines the object types\(^6\) for initiating a subject using \texttt{type-transition} rules. The object type in such rules corresponds to the label of the corresponding executable file.

Using the SELinux MAC policy, we compute the integrity wall for SELinux subject types in the Ubuntu distribution, using Steps 3-6 in Section 4.3.1. To identify helper subjects, we need to

\(^5\)Many, but not all Linux applications have SELinux modules. Those programs without their own modules run under a generic label, such as \texttt{user_t} for generic user programs.

\(^6\)In SELinux, labels are called \texttt{types}. 
identify the subjects that are part of the same application (App(s)). SELinux offers policy modules for applications, and we consider all subjects defined in an application policy module as being part of the same application. All TCB subjects use the same integrity wall.

As a special case, we force all log files to be outside the integrity wall of all subjects. Log file types are easily identified in the SELinux policy (e.g., var_log_t).

### 4.4.2 Identifying Attack Surfaces

Once we have the integrity wall, we use it to locate operations crossing the wall using runtime analysis. Figure 4.2 details our implementation in Linux, which leverages the Linux Security Modules (LSM) framework [56]. We use the SELinux LSM running in the Linux kernel to find untrusted operations. This satisfies the three requirements for identifying untrusted operations (Section 4.3.2). First, it mediates all security-sensitive operations using the LSM interface. Second, we instrument the SELinux access decision function, avc_has_perm, which authorizes a subject type to perform an operation on an object of a particular object type. Finally, the kernel always has details about the currently executing process.

Our implementation enables: (1) uploading of integrity walls; (2) identifying operations outside the wall for a process; (3) finding the process entry point; and (4) logging the operation. We examine implementation of these below.
To upload the integrity walls into the kernel, we export a `debugfs` file to communicate the integrity wall for each subject type to the kernel.

To identify operations accessing objects outside a subject’s wall, we look for operations that input data (i.e., read-like operations). We use the permission map from Apol [74], which identifies whether an operation is read-like, write-like, or both. Interesting to note is that the permission map classifies operations from most covert (least actual input, such as file locking) to most overt (most actual input, such as file reading). We ignore covert operations, excepting for directory searches. We find these valuable because they indicate the presence of an attack surface if the directory is untrusted, even if the file itself is not present. For example, when a program searches for libraries in the current working directory (untrusted search path), the library file itself is not present in benign conditions, but the search indicates an attack surface.

Once we identify an operation that reads from outside the wall, we find the entry point into the process. We first obtain the user stack trace of the process, and then identify the exact entry point. The user stack trace is available by a simple unrolling of the linked list of base pointers on the userspace stack. Such functionality is already available using the `ftrace` framework in the Linux kernel, which we use.

To find the entry point, we search for the stack frame that belongs to a code object that cannot protect itself from all input. In Linux, the `vma_struct` is associated with the name of its code object file (`current->comm`). Thus, for an instruction pointer, we retrieve its code object file through the `vma_struct` to identify entry points. Due to address-space randomization, the exact IP may vary across different runs of the same process. Thus, we use the offset from the base of the binary, which is a constant. Then, this information can be used offline to find the exact line of code in the program if versions of the binaries are available with debug information (many of these are readily available in Ubuntu repositories).

To export the data to userspace, we use `relayfs`. A userspace daemon reads the kernel output and dumps the output to a file. The daemon registers itself with the kernel so it itself will not be traced. For each untrusted operation, we log the following: (1) process name; (2) process ID; (3) the entry point IP into the process as an offset from the base of the binary; (4) the SELinux context of the subject; (5) the SELinux context of the object; (6) operations requested; and (7) filename, if object is a file.

Once the log is available in userspace, it is parsed to output a list indexed by process name (Figure 4.3). For each process, each entry point indexed by IP is listed, with the operation(s),

---

7An SELinux context includes a type and other information, including a user identity and role. We are mainly interested in the type.
Figure 4.3: A log entry from recording of untrusted operations. Two entry points for Apache, with its location information.

types of data read through that entry point, and the number of times that entry point is invoked. If debug information is available for the process, we also print the C source line of code that the entry point is associated with.

4.4.3 Finding Attack Surfaces in Interpreted Code

For interpreted programs, a normal backtrace of the user stack will supply an entry point into the interpreter, and not the script that it is executing. Several shell scripts are run during normal operation of the system, some of which have fallen victim to adversaries, so we also have to accurately identify the attack surfaces of interpreted programs.

We built a kernel-based mechanism for extracting entry points from programs in a variety of interpreted languages (PHP, Python, Bash). This mechanism takes advantage of the common architecture of interpreters. First, these interpreters execute their programs by running each language instruction in a function that we call the fundamental loop function. Each interpreter also maintains a global current interpreter object that represents the state execution of the program, much like a process control block describes processes. Second, when an error occurs, the fundamental loop functions each call print backtrace function, that extracts the stack frames of the currently executing program from the current interpreter object.

To collect entry points for interpreted code, we made interpreter state visible to our kernel mechanism. This involved creating kernel modules that are aware of each interpreter: (1) obtaining access to each’s current interpreter object from their ELF binary symbol tables and (2) using each’s print backtrace functions to find entry points. First, the ELF binary loader already contains mechanisms for accessing the symbol table during program loading that we used to gain access to the desired references. Second, we integrate the backtrace code from the interpreter into the kernel module to find entry points. This task is complicated because we need to use this code to access user addresses from kernel space. To do this safely, we use macros to handle page faults that
may result from user space access (copy_from_user) and remove code that causes side-effects (any writes to user space). Ultimately, very little code needed to be transferred from user space: Bash required 59 lines of code, most of it to handle hash tables in which it stores its variables, whereas PHP required just 11 lines of code. Ultimately, 588 lines of code were added to the kernel for the three interpreters, but 391 lines are for defining data structure headers.

4.4.4 Enforcing Attack Surfaces

We note that the same infrastructure that logs attack surface entry points can also enforce them. In other words, any access crossing the integrity wall would be blocked unless made through one of the authorized entry points for that program and between appropriate types. When any previously unknown entry point crossing the integrity wall is found, its details can be reported to the OS distributor much like crash reports are sent currently, who can decide if the entry point is valid. Note that our enforcing mode can block entry points exercising improper permissions such as untrusted search paths (even those having previously unknown bugs), while access control cannot (as the process might legitimately have those permissions at another entry point). To make our tool performant for online logging and enforcement, we made some enhancements. The integrity walls for subjects are stored as a hash table so looking up whether an object is inside or outside the wall is fast. Also, we only log operations if they have not been logged already.

4.5 Evaluation

In this section, we present the results of our analysis of attack surfaces for the system TCB of an out-of-the-box install of Ubuntu Desktop LTS 10.04.2, with the SELinux policy from the repositories. Our aim in this evaluation is to demonstrate the effectiveness of our approach in computing the attack surfaces for all the system programs in a widely-used Linux distribution in relation to its default SELinux policy. In addition, we performed a detailed study of application programs, including Apache httpd, sshd and the Icete browser, the GNU version of Firefox. While the Apache and OpenSSH are mature programs, we show that our approach can identify attack surface entry points that are easily overlooked. Icete is a relatively new program, so its analysis demonstrates how our approach may aid in the proactive defenses of immature programs.

We found the following results. For the system TCB, we found that: (1) our analysis was able to obtain an attack surface of 81 entry points, including in scripts and some subtle entry points, 35 of which have had previous vulnerabilities, and (2) the attack surface of these programs is a small percentage of their total number of entry points. Examining the system TCB attack surface, we found a vulnerability in one entry point in a script that has been present in Ubuntu
Table 4.1: Wall statistics for the system TCB types and Apache. Subject types correspond to processes, and object types correspond to OS objects (e.g., files) and processes.

<table>
<thead>
<tr>
<th></th>
<th>Types Inside Wall</th>
<th>Types Outside Wall</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Subjects</td>
<td>Objects</td>
</tr>
<tr>
<td>System TCB</td>
<td>111</td>
<td>679</td>
</tr>
<tr>
<td>Apache (httpd)</td>
<td>118</td>
<td>700</td>
</tr>
</tbody>
</table>

for several years. For Apache and sshd, we were able to associate attack-surface entry points with the configuration option that enabled them by correlation with the configuration used by their test suites. In sshd, we found an entry point in the privilege-separated part that was missed by earlier manual identification [71]. In Icecat, we found an entry point that was part of the attack surface due to a bug.

4.5.1 Policy Analysis

This section presents the results of the wall generation algorithm described in Section 4.3.1, on Ubuntu 10.04.2’s SELinux policy. This policy used 65 application policy modules, and had 1058 types (subject and object) in total.

**System TCB.** We first need to locate the TCB that is common to all applications. We build the TCB as described in Section 4.4.1. In total, we had 111 subject types in the TCB.

**Wall Generation Results.** Table 4.1 shows the number of subject types inside and outside the wall for the system TCB subject types and the Apache subject types and the resulting number of high and low integrity object types. We note that only seven new subject types are added to the integrity wall for Apache over the system TCB subject types, which it must already trust. Interestingly, the number of high integrity object types given this wall outnumbers low integrity types by more than 4:1.

We confirmed that for the system TCB programs we examined, the integrity wall derived from the policy corresponded to our intuitive notion of dependence and filtering, (i.e.,) configuration files and library files were within the wall, and user-controlled input outside; we present more details of the wall when we discuss individual applications.

**Violating permissions.** We found that of 115,611 rules in our SELinux policy, 34.4% of these rules (39,848) crossed across the system TCB integrity wall, allowing input from object types outside the system TCB to subjects in the system TCB. The attack surface will consist of entry points in TCB programs that exercise a permission that crosses the wall. These cannot be found from the MAC policy; we need information from the program.
4.5.2 Runtime Analysis

4.5.2.1 System TCB

Evaluation of the system’s TCB demonstrates that: (1) the number of attack surface entry points is a small percentage of the total number of entry points, (2) some attack surface entry points are subtle, and (3) even for mature programs in the system TCB, it is beneficial to locate the attack surface, as demonstrated by a bug we found in an entry point in a mature script that sets up the X server. We gathered the attack surface over several days on a system in normal use, involving bootup, typical application usage, and shutdown.

We found that only 13.8% (295 of 2138) of the total entry points are part of the attack surface (Table 4.2). For example, we found many entry points accessing trusted objects such as etc_t; these entry points would not be part of the attack surface. Thus, simply listing all possible entry points in TCB programs as part of the attack surface would be a significant overapproximation, and not very useful for analysis. Of the 295 attack surface entry points across various programs in the TCB, that received untrusted input, 81 are overt operations (Section 4.4.2); 35 of these have had input filtering problems, many recently discovered for the first time. Five Bash scripts add a total of 8 entry points to the attack surface (Table 4.3). In addition, we found a previously unknown vulnerability at an entry point in a script that sets up the X server that has been around for several years, which we discuss below. This leads us to believe that identification and examination of such entry points prior to deployment is key to preventing exploits.

Runtime analysis is inherently incomplete. To examine completeness, we ran our kernel module in an enforcing mode (Section 4.4.4), where any access crossing the system TCB integrity wall was blocked unless made through one of the entry points in Table 4.2 and between appropriate types. We did not note any new accesses, and since we have a conservative adversary model (including unprivileged users), we believe our set of entry points to be complete for a default Ubuntu 10.04.2 Desktop distribution in relation to its SELinux policy.

We located various subtle entry points that are part of the attack surface. We illustrate this using the example of logrotate. logrotate has an entry point that reads from user_home_t, and the source code for this entry point called a library function that gave little hint as to why this was happening. The reading is actually done inside a library function in libpopt attempting to read its configuration file. As another example, we found entry points calling libc glob(). This function performs the system call getdents returning untrusted directory filenames. A recent untrusted filename attack on logrotate (CVE-2011-1155), was found at this entry point. Neither of the above entry points are as a result of simply calling read() in the source code, and can be easily missed by manual code inspection.
We examined some of the entry points identified, to see if we could locate any obvious problems. The script corresponding to entry point 2 in Table 4.3 is responsible for setting up the /tmp/.X11-unix directory, in which the X server creates a Unix-domain socket that clients can connect to. This flow is into initrc_t from tmp_t (Table 4.2). However, we found that it is vulnerable to a time-of-check-to-time-of-use (TOCTTOU) vulnerability. Looking at the script makes it fairly clear that the developer did not expect a threat at this entry point. This script has existed as part of Ubuntu distributions since at least 2006, and is an example of how locating the attack surface made the problem obvious. We believe that a more thorough testing of the entry points uncovered may expose further vulnerabilities; however, that is outside the scope of this chapter.

### 4.5.2.2 Apache Webserver

We use our tool to evaluate a typical webserver deployment, the Apache webserver (version 2.2.14) with mod_perl. For the wall generation, of particular interest are object types in the SELinux policy module for Apache that were not included in the application TCB, four httpd_user types and httpd_log_t. For the runtime analysis for Apache, we ran the Apache::Test perl module, which contains test cases generated by the Apache developers. We found 30 entry points for the Apache webserver, of which 5 received untrusted operations. Details are in Table 4.4.

We located several entry points accessible to adversaries. Network attacks being well understood, we list implications of the entry points accessing local untrusted data (1, 3 and 5 in Table 4.4). httpd_user_htaccess_t denotes the user-defined configuration file .htaccess. Previous problems with this entry point are Bugtraq IDs 8911, 11182, 15177. httpd_user_content_t are user-defined web pages that Apache serves. A vulnerability due to incorrect parsing of the HTML files is BID 11471. Entry point 5 is where Apache forks a child to execute a user-defined CGI script - the exec operation reads an untrusted executable is untrusted (BID 8275), and could easily be missed by manual analysis.

As mentioned before, our tool was able to associate some entry points with the configuration option that controlled the entry point; different application configurations may expose different attack surfaces. This knowledge is helpful to administrators, who can view the effect of their configuration on the attack surface.

### 4.5.2.3 Secure Shell Daemon

We also performed a study on the SSH daemon, sshd (v. 5.1p1). In total, there were 78 entry points, of which 27 required filtering. 14 of these which correspond to overt input are listed in Table 4.5. Entry points 12, 13 and 14 were opened by non-default configuration options. Of key interest, is that OpenSSH has been re-engineered to separate the privileged operations from
<table>
<thead>
<tr>
<th>TCB Type</th>
<th>Total Entry</th>
<th>Viol. Entry</th>
<th>Program</th>
<th>Overt Violating Entry</th>
<th>Object Type Accessed</th>
<th>Bug ID / Notes</th>
<th>Prev. unknown</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>apmtd</td>
<td>3</td>
<td>3</td>
<td>acp</td>
<td>1 Unix socket</td>
<td>system_dbus_daemon</td>
<td>CVE-2009-0579</td>
<td></td>
<td></td>
</tr>
<tr>
<td>avahi</td>
<td>38</td>
<td>14</td>
<td>avahi-daemon</td>
<td>1 * 2 Unix socket</td>
<td>avahi-daemon</td>
<td>CVE-2007-3372</td>
<td></td>
<td></td>
</tr>
<tr>
<td>consolekit</td>
<td>37</td>
<td>3</td>
<td>console-kit-daemon</td>
<td>1 file</td>
<td>tty_device</td>
<td>CVE-2010-4664</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cupsd</td>
<td>56</td>
<td>10</td>
<td>cupsd</td>
<td>1 TCP socket</td>
<td>cupsd</td>
<td>CVE-2009-0548</td>
<td></td>
<td></td>
</tr>
<tr>
<td>devkit_disk</td>
<td>72</td>
<td>6</td>
<td>udisks-daemon</td>
<td>1 * 4 Unix socket</td>
<td>system_dbus_daemon</td>
<td>CVE-2010-0746</td>
<td></td>
<td></td>
</tr>
<tr>
<td>devkit_power</td>
<td>97</td>
<td>7</td>
<td>upowerd</td>
<td>1 * 2 Unix socket</td>
<td>devkit_power</td>
<td>CVE-2008-4039</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dhcpc</td>
<td>15</td>
<td>2</td>
<td>dhclient3</td>
<td>1 raw socket read</td>
<td>system_dbus_daemon</td>
<td>CVE-2009-0692</td>
<td></td>
<td></td>
</tr>
<tr>
<td>getty</td>
<td>18</td>
<td>3</td>
<td>getty</td>
<td>1 file read</td>
<td>intrayarun2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>halld</td>
<td>188</td>
<td>28</td>
<td>halld</td>
<td>1 Unix socket</td>
<td>system_dbus_daemon</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mitrc</td>
<td>479</td>
<td>23</td>
<td>sh</td>
<td>1 file read</td>
<td>intrayarun2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NetworkManager</td>
<td>76</td>
<td>45</td>
<td>NetworkManager</td>
<td>1 netlink socket</td>
<td>system_dbus_daemon</td>
<td>CVE-2009-0578</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ntps</td>
<td>24</td>
<td>4</td>
<td>ntptime</td>
<td>1 unix socket</td>
<td>system_dbus_daemon</td>
<td>CVE-2001-0944</td>
<td></td>
<td></td>
</tr>
<tr>
<td>restorecond</td>
<td>17</td>
<td>9</td>
<td>restorecond</td>
<td>1 * 3 file read</td>
<td>generic - all types</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rtkit_daemon</td>
<td>20</td>
<td>9</td>
<td>rtkit-daemon</td>
<td>1 file read</td>
<td>user_home</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sshd</td>
<td>78</td>
<td>11</td>
<td>sshd</td>
<td>1 dir search</td>
<td>user_home</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>systemd</td>
<td>29</td>
<td>1</td>
<td>systemd</td>
<td>1 netlink socket</td>
<td>system_dbs_daemon</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>systemd</td>
<td>63</td>
<td>15</td>
<td>dbus-daemon</td>
<td>1 * 3 Unix socket</td>
<td>system_dbs_daemon</td>
<td>CVE-2008-3834</td>
<td></td>
<td></td>
</tr>
<tr>
<td>udev</td>
<td>217</td>
<td>25</td>
<td>udev</td>
<td>1 * 2 netlink socket</td>
<td>system_dbs_daemon</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x比利</td>
<td>56</td>
<td>16</td>
<td>gdm-binary</td>
<td>1 file read</td>
<td>x比利</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xserver</td>
<td>43</td>
<td>18</td>
<td>Xorg</td>
<td>1 * 3 file read</td>
<td>x比利</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>total</td>
<td>2138</td>
<td>295</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2: Attack surface for the system TCB. The first column is the TCB type we consider, the second the total number of entry points for all programs running under that type, and the third the number of violating entry points that cross the integrity wall. Next, we list the specific binary with its overt violating entry points (Section 4.4.2) and the object type accessed that causes the entry point to be violating. We also identify vulnerabilities caused due to insufficient filtering at the overt entry points (we could not find any for the covert entry points). When multiple vulnerabilities are available for an entry point, the chronologically earliest is listed. Highlighted rows are discussed further in text.
Table 4.3: Entry points in Bash scripts.

<table>
<thead>
<tr>
<th>ID</th>
<th>Source Script:Line Number</th>
<th>Target Type</th>
<th>Source Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>/lib/udev/console-setup-tty:76</td>
<td>tty_device_t</td>
<td>udev_t</td>
<td>read user .htaccess file</td>
</tr>
<tr>
<td>2</td>
<td>/etc/rcS.d/S70x11-common:33,47</td>
<td>tmp_t</td>
<td>initrc_t</td>
<td>execute CGI user script</td>
</tr>
<tr>
<td>3</td>
<td>/usr/lib/pm-utils/functions:30</td>
<td>initrc_var_run_t</td>
<td>initrc_t</td>
<td>execute CGI user script</td>
</tr>
<tr>
<td>4</td>
<td>/etc/NetM/dispatcher.d/01ifupdown:27,29</td>
<td>NetworkManager_t</td>
<td>NetworkManager_t</td>
<td>execute CGI user script</td>
</tr>
<tr>
<td>5</td>
<td>/etc/init/mounted-tmp.conf:44,45</td>
<td>init_t</td>
<td>init_t</td>
<td>execute CGI user script</td>
</tr>
</tbody>
</table>

Table 4.4: Apache entry points receiving low-integrity data

<table>
<thead>
<tr>
<th>ID</th>
<th>Source File:Line Number</th>
<th>Object Type Accessed</th>
<th>Config Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>monitor_wrap.c:123</td>
<td>sshd_t</td>
<td>UsePrivilegeSeparation</td>
<td>Master-slave Unix socket read</td>
</tr>
<tr>
<td>2</td>
<td>msg.c:72</td>
<td>sshd_t</td>
<td>-</td>
<td>Unix socket read</td>
</tr>
<tr>
<td>3</td>
<td>msg.c:84</td>
<td>sshd_t</td>
<td>-</td>
<td>Unix socket read</td>
</tr>
<tr>
<td>4</td>
<td>sshd.c:442</td>
<td>sshd_t</td>
<td>-</td>
<td>TCP socket read</td>
</tr>
<tr>
<td>5</td>
<td>dispatch.c:92</td>
<td>sshd_t</td>
<td>-</td>
<td>TCP socket read</td>
</tr>
<tr>
<td>6</td>
<td>packet.c:1005</td>
<td>sshd_t</td>
<td>-</td>
<td>TCP socket read</td>
</tr>
<tr>
<td>7</td>
<td>misc.c:627</td>
<td>user_home_ssh_t</td>
<td>AuthorizedKeysFile</td>
<td>~/.ssh/.authorized_keys file read</td>
</tr>
<tr>
<td>8</td>
<td>channels.c:1396</td>
<td>ptmx_t</td>
<td>-</td>
<td>pseudo-terminal read</td>
</tr>
<tr>
<td>10</td>
<td>serverloop.c:380</td>
<td>sshd_t</td>
<td>-</td>
<td>file file read</td>
</tr>
<tr>
<td>11</td>
<td>loginrec.c:1423</td>
<td>initrc_var_run_t</td>
<td>PermitUserEnvironment</td>
<td>read utmp</td>
</tr>
<tr>
<td>12</td>
<td>session.c:1001</td>
<td>user_home_ssh_t</td>
<td>IgnoreUserKnownHosts</td>
<td>~/.ssh/environment file read</td>
</tr>
<tr>
<td>13</td>
<td>lastfile.c:222</td>
<td>user_home_ssh_t</td>
<td>IgnoreRhosts</td>
<td>~/.ssh/known_hosts file read</td>
</tr>
<tr>
<td>14</td>
<td>auth-rhosts.c:82</td>
<td>user_home_ssh_t</td>
<td>IgnoreRhosts</td>
<td>~/.ssh/.rhosts file read</td>
</tr>
</tbody>
</table>

Table 4.5: sshd entry points that may receive low-integrity data. Entry points marked with * are in the master part of the privilege-separated daemon.
4.5.2.4 Icecat

We also performed a study on the GNU version of Firefox, Icecat. The objective of this study was to look at a relatively less-known project, to see if we could find any problems using our tool. We envision this to be a typical use-case of our tool. In total, we found 18 entry points for Icecat, of which 4 accessed untrusted data. On closer examination of the attack surface, we found an entry point that searched the directory `user_home_dir_t`, whose code was in the dynamic loader/linker library `ld.so`. We suspected an untrusted library search path, and confirmed that this indeed was the case. This could easily be exploited by an adversary-controlled library that the user downloads to her home directory. The developers accepted our patch [75].

4.6 Related Work

Taint tracking has been used to track the flow of untrusted input to a program, and find places where it may affect the integrity of the program. Tracking can be done for whole systems [76, 77] or for specific processes [78, 79]. However, these systems expect manual specifications of taint entry points. For example, [78] considers data “originating from or arithmetically derived from untrusted sources such as the network as tainted”. However, it is not clear that all entry points that receive low-integrity data are locatable manually. Our tool provides this origin input to taint tracking systems.

Manadhata et al. [67] calculate an attack surface metric for programs based on methods, channels and data. They prepare a list of input and output library calls from `libc` that are used to determine the methods. Although this is useful for a first approximation, it does not distinguish between entry points receiving high-integrity input and those receiving low-integrity input. In our analysis, only a small percentage (13.8%) of the entry points were found to receive data of low-integrity. Hence, a simple listing of all such library methods may not give a true picture of work required to secure an application. Further, a library may be called through several layers of libraries, and the context of a lower-layer library call may not be relevant through several layers. We identify the point in the application that receives low-integrity input, which is more helpful to application developers than a low-level library function that may be called in different contexts.

Several practical integrity models [80, 62, 71, 13] are related to our work. UMIP [80] and PPI [62] identify trusted subjects that need to maintain their integrity on receiving low-integrity input. Though their goals differ from ours, they also build integrity walls. UMIP builds integrity walls system-wide based on the DAC policy, whereas PPI uses package dependencies for the same. However, they identify trusted processes as a whole and do not identify entry points within a process, which we have seen to be necessary. Further, they only consider system-wide integrity walls, and
not per-application. Flume [13] allows entry point-level control, but leaves the specification of the policy up to the user, who has to decide which entry points to allow to receive untrusted input. Such policy could benefit from knowledge of the attack surface. Shankar et. al [71] identify that we need to verify input filtering for entry points that receive low-integrity input. However, they identify entry points manually, and missed an entry point in `sshd` that we identified using our automated approach.

Bouncer is a system that uses knowledge of vulnerabilities to generate filters automatically [81]. It symbolically executes the vulnerable program to build a filter that covers the particular attack and generalizes the filter to cover other unauthorized inputs without preventing legitimate function. EXE automatically generates inputs that will crash a program [82]. Both of these systems would benefit from knowledge of the attack surface of a program. In the latter case, this will focus use on legitimate entry points for consideration. The inputs that cause failure may then be used to generate filters via Bouncer.

We could also have leveraged system call interposition [83, 84, 85, 86, 87] to monitor objects accessed by a program, instead of doing it in the kernel. However, as noted in Section 4.3.2, we have to maintain a list of system calls that causes inputs, know the sets of objects accessed by each of these calls, and fetch the security contexts of these objects from the kernel – all duplicating information readily available in the kernel. Also, system call interposition has high overhead and is challenging to do system-wide.

4.7 Conclusion

In this chapter, we introduced an approach to identify attack surfaces in programs with respect to an integrity wall constructed from the system’s security policy. We implemented a system in the Linux kernel that enabled precise identification of attack surface entry points, even in interpreter scripts. Our results indicate that accurate location of attack surfaces requires considering a program in relation to the system’s access control policy. For the system TCB in an Ubuntu 10.04.2 Desktop system, we obtained an attack surface of 81 entry points, some subtle; 35 of these have had past vulnerabilities, many recently. Our attack surface indicated an entry point in `sshd` that was missed by earlier manual analysis, and an entry point in the GNU Icecat browser that was due to an untrusted search path bug. Further, our attack surface helped us find a bug in an entry point of the system TCB of Ubuntu that has been around for several years. We envision that our tool will be used on new programs to identify attack surfaces before an adversary does and prepare defenses, moving us away from the current penetrate-and-patch paradigm.
Chapter 5

STING: Finding Name Resolution Vulnerabilities in Programs

In this chapter, we present STING, our testing framework that detects name resolution vulnerabilities, a subset of resource access vulnerabilities caused by direct adversarial control of the resource (as opposed to indirect control through a name).

5.1 Introduction

The association between names and resources is fundamental to computer science. Using names frees computer programmers from working with physical references to resources, allowing the system to store resources in the way that it sees fit, and enables easy sharing of resources, where different programs may use the different names for the same object. When a program needs access to a resource, it presents a name to a name server, which uses a mechanism called name resolution to obtain the corresponding resource.

While name resolution simplifies programming in many ways, its use has also resulted in several types of vulnerabilities that have proven difficult to eliminate. Adversaries may control inputs to the name resolution process, such as namespace bindings, which they can use to redirect victims to resources of the adversaries’ choosing. Programmers often fail to prevent such attacks because they fail to validate names correctly, follow adversary-supplied namespace bindings, or lack insight into which resources are accessible to adversaries. Table 5.1 lists some of the key classes of these vulnerabilities.

While a variety of system defenses for these attacks have been proposed, particularly for name resolution attacks based on race conditions [35, 36, 37, 88, 6, 7, 8, 9, 10, 11], researchers have found that such defenses are fundamentally limited by a lack of knowledge about the program [12]. Thus,
the programmers’ challenge is to find such vulnerabilities before adversaries do. However, finding such vulnerabilities is difficult because the vectors for name resolution attacks are outside the program. Table 5.1 shows that adversaries may control namespace bindings to redirect victims to privileged resources of their choice, using what we call *improper binding attacks* or redirect victims to resources under the adversaries’ control, using what we call *improper resource attacks*. Further, both kinds of attacks may leverage the non-atomicity of various system calls to create races, such as the time-of-check-to-time-of-use (TOCTTOU) attacks [32, 17], which makes them even more difficult for victims to detect.

Researchers have explored the application of dynamic and static analysis to detect namespace resolution attacks. Dynamic analyses [21, 22, 23, 24, 25] log observed system calls to detect possible problems, such as check-use pairs that may be used in TOCTTOU attacks. However, the existence of problems does not necessarily mean that the program is vulnerable. Many of the check-use pairs found were not exploitable. Static analyses use syntactic analyses [17, 18] and/or semantic models of programs to check for security errors [19, 20], sometimes focusing on race conditions [52]. These static analyses do not model the system environment, however, so they often produce many false positives. In addition, several of these analyses result in false negatives as they rely on simpler models of program behavior (e.g., finite state machines), limited alias analysis, and/or manual annotations.

The key insight is that such name resolution attacks are possible only when an adversary has write access to a directory shared with the victim. Using this write access, adversaries can plant files with names used by victims or create bindings to redirect the victim to files of the adversaries’ choice. Chari et al. [8] demonstrated that when victims use such bindings and files planted by adversaries attacks are possible, so they built a system mechanism to authorize the bindings used in name resolution. However, we find that only a small percentage of name resolutions are really accessible to adversaries and most of those are defended by programs. Further, the solution proposed by Chari et al. is prone to false positives, as any pure system solution is, because it lacks information about the programs’ expected behaviors [12]. Instead, we propose to test programs for name resolution vulnerabilities by having *the system assume the role of an adversary*, performing modifications that an adversary is capable of, at runtime. Using the access control policy and a list of adversarial subjects, the system can determine whether an adversary has write access to a directory to be used in a name resolution. If so, the system prepares an attack as that adversary would and detect whether the program was exploited or immune to the attack (e.g., did the program follow the symbolic link created?). This is akin to directed black-box testing [89], where a program is injected with a dictionary of commonly known attacker inputs.

In this chapter, we design and implement the STING test engine, which finds name resolution
<table>
<thead>
<tr>
<th>Attack</th>
<th>CWE ID</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Improper Binding Attacks</strong></td>
<td></td>
</tr>
<tr>
<td>UNIX Smlink Following</td>
<td>CWE-61</td>
</tr>
<tr>
<td>UNIX Hard Link Following</td>
<td>CWE-62</td>
</tr>
<tr>
<td><strong>Improper Resource Attacks</strong></td>
<td></td>
</tr>
<tr>
<td>Resource Squatting</td>
<td>CWE-283</td>
</tr>
<tr>
<td>Untrusted Search Path</td>
<td>CWE-426</td>
</tr>
<tr>
<td><strong>Attacks Caused by Either Bindings or Resources</strong></td>
<td>CWE-362</td>
</tr>
<tr>
<td>TOCTTOU Race Condition</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.1: Classes of name resolution attacks.

vulnerabilities in programs by performing a dynamic analysis of name resolution processing to produce directed test cases whenever an attack may be possible. STING is an extension to a Linux Security Module [56] that implements the additional methods described above to provide comprehensive, system-wide testing for name resolution vulnerabilities. Using STING, we found 21 previously-unknown name resolution vulnerabilities in 19 different programs, ranging from startup scripts to mature programs, such as cups, to relatively new programs, such as x2go. We detail several bugs to demonstrate the subtle cases that can be found using STING. Tests were done on Ubuntu and Fedora systems, where interestingly some bugs only appeared on one of the two systems because of differences in the access control policies that implied different adversary access.

This research makes the following novel contributions:

- We find that name resolution attacks are always possible whenever a victim resolves a name using a directory where its adversaries have permission to create files and/or links, as defined in Section 5.3. If a victim uses such a directory in resolving a name, an adversary may redirect them to a resource of the adversary’s choosing, compromising victims that use such resources unwittingly.

- We develop a method for generating directed test cases automatically that uses a dynamic analysis to detect when an adversary could redirect a name resolution in Section 5.4.1.

- We develop a method for system-wide test case processing that detects where victims are vulnerable to name resolution attacks, restores program state to continue testing, and manages the testing coverage in Section 5.4.2.

- We implement a prototype system STING for Linux 3.2.0 kernel, and run STING on the current versions of Linux distributions, discovering 21 previously-unknown name resolution vulnerabilities in 13 different programs. Perhaps even more importantly, STING finds that 90% of adversary-accessible name resolutions are defended by programs correctly, eliminating many false positives.
We envision that STING could be integrated into system distribution testing to find programs that do not effectively defend themselves from name resolution attacks given that distribution’s access control policy before releasing that distribution to the community of users.

5.2 Problem Definition

Processes frequently require system level resources like files, libraries, and sockets. Since the system’s management of these objects is unknown to the process, names are used as convenient references to the desired resource. A name resolution server is responsible for converting the requested resource name to the desired object via a namespace binding. Typical namespaces in Unix-based systems include the filesystem and System V IPC namespaces (semaphores, shared memory, message queues, etc.). Some namespaces may even support many-to-one mappings (e.g., multiple pathnames may be linked to the same file inode).

Unfortunately, various name resolution attacks are possible when an attacker is able to affect this indirection between the desired resource and its name. In this section, we broadly outline two classes of name resolution attacks and give several instances of them. We then discuss how previous efforts attempt to defend against these attacks and their limitations. Finally, we present our solution, STING, that overcomes many of these shortcomings.

5.2.1 Name Resolution Attacks

Malicious parties can control the name resolution process by modifying the namespace’s binding to trick victim processes into accessing unintended resources. We find that these attacks can be categorized into two classes. The first, improper binding attacks, are when attackers introduce
bindings to resources outside of the attacker’s control. This can give adversary indirect access to the resource through the victim. Such attacks are instances of the confused deputy [47]. The second class, improper resource attacks, is when an attacker creates an unexpected binding to a resource the adversary controls.

Instances of these attacks depend on the namespace. For example, the filesystem namespace is often exploited through malicious path bindings like symbolic links and the creation of files with frequently used names. Consider a mail reader program running as root attempting to check mail from /var/mail/root. Users in the mail group are permitted to place files in this directory for the program to read and send. Figure 5.1 demonstrates how name resolution attacks from both categories could be performed on this program.

- **Symbolic link following:** The adversary wishes to exfiltrate a protected file (/etc/passwd) that it cannot normally access. Since users in group mail are permitted to create (and delete) bindings (files) in /var/mail, the adversary inserts a symbolic link /var/mail/root in the namespace that is bound to the desired file. If a victim mail program running as root does not check for this link, it might inadvertently leak the protected file. A similar attack can be launched through hard links. This is an instance of an improper binding attack, where adversaries use control of bindings to redirect victim programs with privileges to access or modify resources the adversaries cannot directly.

- **Squatting:** Even if the mail program defends itself against link following attacks, the adversary could simply squat a file on /var/mail. If the mail program accepts this file, the adversary could spoof the contents of mail read by root. This is an example of an improper resource attack, where the adversary uses control of bindings to create a resource under her control when the victim does not expect to interact with the adversary.

- **Untrusted search path:** Programs frequently rely on files like system libraries or configuration files, but the names they supply to access these files may be wrong. One frequent cause is the program supplying a name relative to its working directory, which causes a problem if the working directory is adversary controlled. Adversaries can then simply bind arbitrary resources at these filenames, possibly gaining control of the victim’s program. This is another instance of an improper resource attack, where the adversary supplies an improper resource to the victim.

While the attacks an adversary can carry out are well known, the ways in which programs defend themselves are often ad hoc and complex [90]. Even the most diligent programs may fail to catch all the ways in which an adversary might manipulate these namespaces. Moreover, defenses to these
attacks can often be circumvented through *time-of-check-to-time-of-use (TOCTTOU)* attacks. To do this, the adversary waits until the mail program checks that `/var/mail/root` is a regular file prior to opening it and then switches the file to a link before the `open` call is made. Given the variety of possible name resolution attacks and the complex code needed to defend against them, it should come as little surprise that vulnerabilities of this type continue to be uncovered. Such attacks contribute 5-10% of CVE entries each year.

5.2.2 Detecting Name Resolution Attacks

Researchers have explored a variety of dynamic and static analyses to detect instances of name resolution attacks, particularly TOCTTOU attacks. However, all such analyses are limited in some ways when applied to the problem of detecting name resolution attacks.

5.2.2.0.1 Static Analysis  Static analyses of TOCTTOU attacks vary from syntactic analyses specific to *check-use* pairs [17, 18], to building various models of programs to check for security errors [19, 20, 91], to race conditions in general [52]. However, static analyses are disconnected from essential environmental information, such as the system’s access control policy to determine whether an adversary can even launch an attack. For example, a program may legitimately accesses files in `/proc` without checking for name resolution attacks; however, the same cannot be done in `/tmp`. Thus, these analyses yield a significant number of false positives. Further, static techniques are limited to TOCTTOU attacks, due to the absence of standardized program checks against name resolution attacks in general.

5.2.2.0.2 Dynamic Analysis  Dynamic analyses [21, 22, 23, 24, 25] typically take a system-wide view, logging observed system calls from processes to detect possible problems, such as check-use pairs. Dynamic analyses can also detect specific vulnerabilities, either at runtime [25] or after the fact [24]. Compared to static analyses, dynamic analyses can take into account the system’s environment, but suffer the disadvantage of being unaware of the internal code of the program. In addition, the quality of dynamic analysis is strongly dependent on the test cases produced. Because name resolution attacks require an active adversary, the problem is to produce adversary actions in a comprehensive manner. Using benign system traces may identify some vulnerabilities, such as those built-in to the normal system configuration [90], but will miss many other feasible attacks. Finally, any dynamic analysis must distinguish program actions that are safe from those that are vulnerable effectively. We have found that programs successfully defend themselves from a large percentage of the attempted name resolution attacks (only 12.5% were vulnerable), so test case processing must find cases where program defenses are actually missing or fail. Since previous
dynamic analyses lack insight into the program, several false positives have resulted.

5.2.2.0.3 Symbolic Execution Researchers have recently had success finding the conditions under which a program is vulnerable using symbolic execution [92, 93, 94, 82, 81, 95, 96, 97, 98, 99, 100]. Symbolic execution has been used to produce test cases for programs to look for bugs [101, 102], to generate filters automatically [81, 93], and to generate test cases to leverage vulnerabilities in programs [103] automatically. In these symbolic execution analyses, the program is analyzed to find constraints on the input values that lead to a program instruction of interest (i.e., where an error occurs). Then, the symbolic execution engine solves for those constraints to produce a concrete test case that when executed would follow the same path. Finding name resolution attacks using symbolic execution may be difficult because the conditions for attack are determined mainly by the operating environment rather than the program. While symbolic execution often requires a model of the environment in which to examine the program, the environment needs to be the focus of analysis for finding name resolution attacks.

5.2.3 Our Solution

As a result, we use a dynamic analysis to find name resolution vulnerabilities, but propose four key enhancements to overcome the limitations of prior analyses of all types.

First, each name resolution system call is evaluated at runtime to find the bindings used in resolution and to determine whether an adversary is capable of applying one or more of the attack types listed in Table 5.1. If so, a test case resource is automatically produced at an adversary-directed location in the namespace and provided to the victim. As a result, test cases are only applied where adversaries have the access necessary to perform such attacks.

Second, we track the victim’s use of the test case resource to determine whether it accepts the resource as legitimate. If the victim uses the resource (e.g., reads from it), we log the program entrypoint$^1$ that obtained the resource as vulnerable. While it is not always possible to exploit such a flaw to compromise the program, this approach greatly narrows the number of false positives while still locating several previously-unknown true vulnerabilities. We also log the test cases run by program entrypoint to avoid repeating the same attack.

Third, another problem with dynamic analysis is ensuring that the analysis runs comprehensively even though programs may fail or take countermeasures when attacks are detected. We take steps to keep programs running regardless of whether they fall victim to the attack or not. Our test case resources use the same data as the expected resource to enable vulnerable programs to

---

$^1$A program entrypoint is a program instruction that invoked the name resolution system call, typically indirectly via a library (e.g., libc).
continue, and we automatically revert namespaces after completion of a test and restart programs that terminate when an attempted attack on them is detected.

5.3 Testing Model

In this section, we define an adversarial model that we use to generate test cases that can be used to identify program vulnerabilities.

Our goal is to discover vulnerabilities that will compromise the integrity of a process. Classically, an integrity violation is said to occur when a lower integrity process provides input to a higher integrity process or object [104, 70]. For the name resolution attacks described in the last section (see Table 5.1), integrity violations are created in two ways: (1) improper binding attacks, where adversaries may redirect name resolutions to resources that are normally not modifiable by adversaries, enabling adversaries to modify higher integrity objects, and (2) improper resource attacks, where adversaries may redirect name resolutions to resources that are under the adversaries’ control, enabling adversaries to deliver lower integrity objects to processes. In this section, we define how such attacks are run and detected to identify the requirements for the dynamic analysis.

A nameserver performs namespace resolution by using a sequence of namespace bindings, \( b_{ij} = (r_i, n_j, r_k) \), to retrieve resource \( r_k \) from resource \( r_i \) given a name \( n_j \). In a file system, \( r_i \) is a directory, \( n_j \) is an element of the name supplied to the nameserver for resolution, and \( r_k \) is another directory or a file. Attacks are possible when adversaries of a victim program have access to modify binding \( b_{ij} \) to \( (r_i, n_j, r'_{k}) \) or create such a binding if it does not exist, enabling them to redirect the victim’s process to a resource \( r'_{k} \) instead of \( r_k \). Since bindings cannot be modified like files, adversaries generally require the delete permission to remove the old binding and the create permission to create the desired binding to perform such modification. Two types of name resolution attacks are possible when adversaries have such permission (e.g., write permission to directories in UNIX systems).

**Improper binding attacks** use the permission to modify a binding to create a link (symbolic or hard) to an existing resource that is inaccessible to the adversary, as in symbolic and hard link attacks described above. That is, the improper binding may lead to privilege escalation for the adversary by redirecting the victim process to use an existing resource on behalf of that adversary.

**Improper resource attacks** use the permission to modify a binding to create a new resource controlled by the adversary. That is, the adversary tries to trick the victim into using the improper resource to enable the adversary to provide malicious input to the victim, such as in resource squatting and untrusted search path attacks described above.

**STING** discovers name resolution vulnerabilities by identifying scenarios where an attack is
possible and generating test cases to validate the vulnerability. Whenever a *name resolution system call* is requested by the victim (i.e., a system call that converts a name to a resource reference, such as `open`), Sting finds the bindings that would be used in the namespace resolution process to determine whether an adversary of the process has access to modify one or more of these bindings. If so, Sting generates an *attack test case* by producing a test case resource, which emulates either an existing, privileged resource or a new adversary-controlled resource, and adjusting the bindings as necessary to resolve to that resource. A reference to this test case resource is returned to the victim.

**Vulnerability in the Victim.** We define a victim to be vulnerable if the victim runs an *accept system call* using a reference to the test case resource.

A victim accepts a test case resource if it runs an *accept system call*, a system call that uses the returned reference to the test case resource to access resource data (e.g., `read` or `write`). If a victim is tricked into reading or writing a resource inaccessible to the adversary, the adversary can modify the resource illicitly\(^2\). If a victim is tricked into reading or writing a resource that is controlled by the adversary, then the adversary can control the victim’s processing.

### 5.4 Design

The design of Sting is shown in Figures 5.2 and 5.3. Sting is divided into two phases. The *attack phase* of Sting is invoked at the start of a name resolution system call. Sting resolves the name itself to obtain the bindings that would be used in normal resolution, and then determines whether an attack is possible using the program’s adversary model. When an attack is possible, Sting chooses an attack from the list in Table 5.1 that has not already been tried and produces a test case resource and associated bindings to launch the attack. The *detect phase* of Sting is invoked on accept system calls. This phase detects if a process “accepted” Sting’s changes, indicating a

---

\(^2\)A read operation on a test case resource is indicative of integrity problems if the resource is opened with read-write access.
vulnerability, and records the vulnerability information in the previously added entry in the attack history. Sting reverts the namespace to avoid side-effects. These two phases are detailed below.

Sting is designed to test systems, not just individual programs, so Sting will generate test cases for any program in the system that has an adversary model should the opportunity arise. To control the environment under which a program is run, Sting intercepts execve system calls. For example, programs that may be run by unprivileged users (e.g., setuid programs) are started in an adversary’s home directory by this interception code. Other than initialization, the attack and detect phases are the same for all processes.

5.4.1 Attack Phase

Shadow resolution. Whenever a name resolution system call is performed, Sting needs identify whether an attack is possible against that system call. The first step is to collect the bindings that would normally be used in the resolution. We cannot use the existing name resolution mechanism, however, since that has side-effects that may impact the process and also does not gather the desired bindings for evaluation. Instead, we perform a shadow resolution that only collects the bindings.

There are two challenges with shadow resolution. First, we have to ensure that all name resolution operations performed by the system are captured in the shadow resolution. This task can be tricky because some name resolution is performed indirectly. For example, exec resolves the interpreter that executes the program in the “shebang” line in addition to the program whose name is an argument to the system call. To capture all the name resolution code, we use Cscope to find all the system calls that invoke a fundamental function of name resolution, do_path_lookup. Using this we find 62 system calls that do name resolution for the Linux filesystem. The three System V IPC system calls that do name resolution were identified manually.

Second, we need to modify the name resolution code to collect the bindings used without causing side-effects in the system. Fortunately, the name resolution code in Linux does not cause side-effects itself. The system call code that uses name resolution creates the side-effects. Thus, we simply invoke the name resolution functions directly when the system call is received. Some effort must be taken to format the call to the name resolution code at the start of the system call, but fortunately the necessary information is available (name, flags, etc.).

Find vulnerable bindings. To carry out an attack, Sting has to determine whether any adversary of the program has the necessary permissions to the bindings involved in the resolution. To answer this question, we need to identify the program’s adversaries and evaluate the permissions these adversaries have to bindings efficiently. We note that the specific permissions necessary to launch an attack are specified in Section 5.3.

3http://cscope.sourceforge.net/
We do not want the dynamic analysis to depend on a single adversary model for the system, but instead permit the use of program-specific adversary models. The adversaries of a process are determined by the process’s subject (i.e., in the access control policy) and optional program-specific sets of subjects and/or objects that are adversaries or adversary-controlled, respectively. From this information, a comprehensive set of adversary subjects are computed. Using a discretionary access control (DAC) policy, an adversary is any subject other than the victim and the trusted superuser root. Chari et al. used the DAC policy in their dynamic analysis [90], which worked adequately for root processes but incurred some false positives for processes run under other subjects. For systems that enforce a mandatory access control (MAC) policy, methods have been devised to compute the adversaries of individual subjects [105, 62]. We note that MAC adversaries may potentially be running processes under the same DAC user, so they are typically finer-grained.

Finding the permissions of a process’s adversaries at runtime must be done efficiently. If a process has several adversaries, the naive technique of querying the access control policy for each adversary in turn is unacceptable. To solve this, we observe we can precompute the adversaries of particular process as in a capability-list, where each process has a list of tuples consisting of an object (or label in a MAC policy), a list of adversaries with create permission to that object (or label), and the list of adversaries with delete permission to that object (or label). We store these in a hash table for quick lookup at runtime.

Identify possible attacks. Once we identify a binding that is accessible to an adversary, we need to choose an attack from which to produce a test case. For improper binding attacks, an attack needs to modify a binding from an existing resource to the target resource using a symbolic or hard link. Such attacks are only possible in the Linux filesystem namespace, where a single file (inode) may have multiple names.

Improper resource attacks are applicable across all namespaces. We consider two instances of improper resource attacks (see Table 5.1). For resource squatting, attacks are only meaningful if the adversary can supply a resource with a lower integrity than the victim intended to access. To determine the victim’s intent, we simply check if a non-adversarial subject has permissions to supply the resource the adversary is attacking. This occurs in directories shared by parties at more than one integrity level. If so, we assume that the victim intended to access the higher integrity file (i.e., one that could be created by a non-adversarial subject), and attempt a squatting attack which succeeds if the victim later accepts the test case resource. MOPS [19] uses a similar but narrower heuristic to identify intent and detect ownership stealing attacks, which are another case of resource squatting attacks.

---

4We discount root superuser permissions while checking non-adversarial subjects, as otherwise root will be a non-adversary in any directory.
Launch an attack. Launching an attack involves making modifications to the namespace to generate realistic attack scenarios. Different attacks modify the namespace in different ways. For improper binding attacks, we create a new test case resource (e.g., file) that represents a privileged resource, and change the adversary-modifiable bindings to point to it (e.g., symbolic link). For improper resource attacks, we replace the existing resource (if present) with a new test case resource and binding.

Modification of the filesystem namespace in particular presents challenges of backing up existing files, rollback and multiple views for different subjects. First, we have to change the file system to create the test cases, such as deleting existing files. Second, once the test case finishes, we need to rollback the namespace to its original state. While we can back up files (costing the overhead of copy), other resources such as UNIX domain sockets are hard or impossible to rollback once destroyed. Another requirement is that the attack should only be visible to the appropriate victim subjects having the attacker as an adversary. Thus, direct modification of the existing filesystem is undesirable.

To solve the above problems, we take inspiration from the related problem of filesystem unions. Union filesystems unite two or more filesystems (called branches) together \[106, 107\]. A common use-case is in Live CD images, where the base filesystem mounted from a CDROM is read-only, and a read-write branch (possibly temporary) is mounted on top to allow changes. When a file on the lower branch is modified, it is “copied up” to the upper branch and thereafter hides the corresponding lower branch file. “Whiteouts” are created on the upper branch for deleted files.

To support Sting, our general strategy is thus to have a throw-away “upper branch” mounted on top of the underlying filesystem “lower branch”. Sting creates resources only on the upper branch. As Sting does not deal with data, files are created empty. Next, Sting directs operations to upper branches if the resource exists on the upper branch and was created by an adversary to the currently running process. This enables different processes to have different views of the filesystem namespace.

Once a test case resource is created, we taint it using extended attributes to identify when it is used in an accept system call\(^5\), signaling a vulnerability. We also record rollback information about the resources created in a rollback table.

These changes to the bindings have to be done as the adversary. The most straightforward option is to have a separate “adversary process” that is scheduled in to perform needed changes. This was the first option we explored; however, it introduced significant performance degradation due to scheduling. Instead, we perform the change using the currently running victim process itself.

\(^5\)Some filesystems do not support extended attributes. Since we use tmpfs as the upper branch, we extended it to add such support for our testing. For other namespaces such as IPCs, we store the taint status in a field in the security data structure defined by SELinux.
We change the credentials of the victim process to that of the adversary, carry out the change, and then revert to the victim’s original credentials. We do this without leaving behind side effects on the victim process’ state. For example, if we create a namespace binding using `open`, we close the opened file descriptor.

There are some cases where the system needs to revert a test case resource back to a “benign” version. “Check” system calls [21] (e.g., `stat`, `lstat`) resolve the name to verify properties of the resource, so attacks should present a benign resource to prevent the victim from detecting the attack. We simply redirect such accesses to the lower branch.

In addition to adversarial modification of the namespace, STING also changes process attributes relevant to name resolution. In particular, it changes the working directory a process is launched in to an adversary-controlled directory.

We want to prevent STING from launching the same attack multiple times. Trying the same attack on the same system call prevents forward exploration of the attack space and further program code from being exercised. A unique attack associates an attack type with a particular system call entrypoint in the program. Thus, when we launch an attack, we check an attack history that stores which attack types have been attempted already and their result (see Figures 5.2 and 5.3). We do not attempt multiple binding changes for an attack type. We have not found any programs that perform different checks for name resolution attacks based on the names used. Tracking such history requires unique identification of system call entrypoints in the program, which we discuss in Recording below.

### 5.4.2 Detect Phase

**Detect vulnerability.** Detecting whether a victim is vulnerable to an attack is relatively straightforward – we simply have to determine if the program “accepted” the test case resource. Definition of acceptance for different attacks are presented in Table 5.2. On the other hand, we conclude that the program defends itself properly from an attack if it: (1) exits without accepting the test case resource or (2) retries the operation at the same program entrypoint. When detection determines that a victim is vulnerable or invulnerable, it fills this information in the attack history entry created during the attack phase, and optionally logs the fact.

STING detects successful attacks by identifying use of a test case resource. Each test case resource is marked when returned to the victim. To detect when a victim uses a test case resource, we must have access to the inode, so such checking is integrated with the access control mechanism (e.g., Linux Security Module). Once a test case resource is found, we need to determine if it is being accepted by retrieving the system call invoked. As an access control check may apply to multiple system calls, we have to retrieve the identity of the system call from the state of the calling process.
<table>
<thead>
<tr>
<th>Attack</th>
<th>Accept</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbolic link</td>
<td>write-like, read, readlink</td>
</tr>
<tr>
<td>Hard link</td>
<td>write-like, read</td>
</tr>
<tr>
<td>File squat</td>
<td>write-like, read</td>
</tr>
<tr>
<td>UNIX-domain socket squat</td>
<td>connect</td>
</tr>
<tr>
<td>System V IPC squat</td>
<td>msgsnd, semop, shmat</td>
</tr>
</tbody>
</table>

Table 5.2: Table showing calls that signify acceptance, and therefore detection, for various attacks. 
write-like system calls are any calls that modifies the resource’s data or metadata (e.g., fchmod).

Vulnerabilities found have their attack history record logged into user space.

We note that the process of detecting a vulnerability is the same for all attack types, including those based on race conditions. STING automatically switches resources between check and use as discussed above, so we only need to detect when an untrusted resource is accepted. 

fstat is not an accept system call, so the “use” of the test case resource in that system call does not indicate a vulnerability. Thus, if the program should somehow detect an attack using fstat, preventing further use of the test case resource, then STING will not record a vulnerability.

Update attack history. Once a particular attack has been tried on a system call, trying it again in future invocations of the program is redundant and may prevent further code from being exercised. Avoiding this problem requires storing attacks tested for system calls in the attack history. The challenge is unique identification of the system call entrypoint, which uniquely identifies the instruction from which the program made the system call. To find this instruction, we perform a backtrace of the user stack to find the first frame within the program that is not in the libraries.
We also extend our system to support interpreters by porting interpreter debugging code into the kernel that locates and parses interpreter data structures to the current line number in the script, for the Bash, PHP and Python interpreters. Only between 11 and 59 lines of code were necessary for each interpreter. We use the current line number in the script as the entrypoint for interpreted programs.

Namespace recovery. Finally, we make changes so that STING can work online despite changing the namespace state. While it appears that such changes could cause processes to crash, we have not found this to be the case. Unlike data fuzzing, we find changes in namespace state do not cause programs to arbitrarily crash, as we preserve data and only change resource metadata. When an attack succeeds, the only change needed is to redirect the access to the corresponding resource in the lower branch of the unioned filesystem that contains the actual data (if one exists), and delete the resource in the upper branch. On the other hand, if the attack fails, STING again deletes the resource in the upper branch. Programs that protect themselves proceed in two ways. First, the program might retry the same system call again. Interestingly, we find this happens in
a few programs (Section 5.6.2). In this case, STING will not launch an attack at that entrypoint again, and the program again continues. Second, the program might exit. If so, STING records that the attack failed at that entrypoint and restarts the program with its original arguments (recorded via execve interception). For many programs that exit, restarting them from the beginning does not affect system correctness. Thus, we find our tool can perform online without complex logic. We are currently exploring how to integrate process checkpointing and rollback [108] to carry out recovery more gracefully for the exit cases.

5.5 Implementation

![Diagram of STING implementation](image)

Figure 5.4: STING is implemented in the Linux kernel and hooks on to the system call beginning and LSM hooks. It has a modular implementation. We show here in particular the interaction between user and kernel space.

Figure 5.4 shows the overall implementation of STING. STING is implemented in the Linux 3.2.0 kernel. It hooks into LSM through the SELinux function avc_has_perm for its detect phase, and into the beginning of system calls for its attack phase. STING has an extensible architecture, with modules registering themselves, and rules specifying the order and conditions under which modules are called.

The modules implement functionality corresponding to the steps shown in the design (Figures 5.2 and 5.3). The entrypoint module uses the process’ eh_frame ELF section to perform the user stack unwinding. eh_frame replaces the DWARF debug_frame section, and is enabled by default on Ubuntu 11.10 and Fedora 15 systems. Stack traces in interpreters yield an entrypoint inside the interpreter, and not the currently executing script. We extended our entrypoint module to identify line numbers inside scripts for the Bash, PHP and Python interpreters [109].

The data for the MAC adversary model is pushed into the kernel through a special file, and code to use each of these to decide adversary access is in the test attack conditions module. After launching an attack, the modified resources are tainted through extended attributes for later detection when a victim program uses the resource. We had to extend the tmpfs filesystem to handle
extended attributes. The recording module can print vulnerabilities as they are detected, and also log the vulnerabilities and search history into userspace files through relayfs. A startup script loads the attack search history into the kernel during bootup, so the same attacks are not tried again.

We prototyped a modified version of the UnionFS filesystem [106] for STING. We mount a tmpfs as the upper branch, and the root filesystem as the lower branch. The main change involved redirecting to the upper or lower branches depending on a subject’s adversaries, and disabling irrelevant UnionFS features such as copy-up.

5.6 Evaluation

We first evaluate STING’s ability to finding bugs, as well as broader security issues in Section 6.6. We then analyze the suitability of STING as an online testing tool in Section 5.6.2

5.6.1 Security Evaluation

The aim of the security evaluation is to show that:

- STING can detect real vulnerabilities with a high percentage of them being exploitable in both newer programs, and older, more mature programs, and

- Normal runtime and static analysis would result in a large number of false positives, and normal runtime would also miss some attacks.

We tested STING on the latest available versions of two popular distributions - Ubuntu 11.10 and Fedora 16. In both cases, we installed the default base Desktop distribution, and augmented them with various common server programs (Table 5.3). Note that STING requires no additional special setup; it simply reacts to normal name resolution requests at runtime. We collected data on both systems over a few days of normal usage.

<table>
<thead>
<tr>
<th>Server Programs Installed</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>BIND DNS Server</td>
<td>Apache Web Server</td>
</tr>
<tr>
<td>MySQL Database</td>
<td>PHP</td>
</tr>
<tr>
<td>Postfix Mail Server</td>
<td>Postgresql Database</td>
</tr>
<tr>
<td>Samba File Server</td>
<td>Tomcat Java Server</td>
</tr>
</tbody>
</table>

Table 5.3: Server programs installed on Ubuntu and Fedora.
### Table 5.4: Table showing the total number of distinct entrypoints invoking system calls performing namespace resolutions, number accessible to adversaries under an adversary model, and number of interfaces for which STING detected vulnerabilities.

<table>
<thead>
<tr>
<th>Adversary model</th>
<th>Total Resolutions</th>
<th>Adversary Access</th>
<th>Vulnerable</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAC - Ubuntu</td>
<td>2345</td>
<td>134 (5.7%)</td>
<td>21 (0.9%)</td>
</tr>
<tr>
<td>DAC - Fedora</td>
<td>1654</td>
<td>66 (4%)</td>
<td>5 (0.3%)</td>
</tr>
</tbody>
</table>

#### 5.6.1.1 Finding Vulnerabilities

Using a DAC attacker model, in total, STING found 26 distinct vulnerable resolutions across 20 distinct programs (including scripts). Of the 26 vulnerable resolutions, 5 correspond to problems already known but unfixed. 17 of these vulnerabilities are latent \[90\], meaning a normal local user would have to gain privileges of some other user and can then attempt further attacks. For example, one bug we found required the privileges of the user postgres to carry out a further attack on root. This can be achieved, for example, by remote network attackers compromising the PostgreSQL daemon. For all vulnerabilities found, we manually verified the source code that a name resolution vulnerability existed. Several bugs were reported, of which 2 were deemed not exploitable (although a name resolution vulnerability existed) (Section 5.6.1.3).

Table 5.4 shows the total number of distinct name resolutions received by STING that were attackable. This data shows challenges facing static and normal runtime analysis. Only 4-5.7% of the total name resolutions are accessible to the adversary under the DAC adversary model. Therefore, static analysis that looks at the program alone will have a large number of false positives, because programs do not have to protect themselves from name resolutions inaccessible to the adversary. Second, normal runtime analysis cannot differentiate between when programs are vulnerable and when they protect themselves appropriately. We found only 7.5-15.6% of the name resolutions accessible to the adversary are actually vulnerable to different name resolution attacks. Further, 6 of these vulnerabilities would simply not have been uncovered during normal runtime; they are untrusted search paths that require programs to be launched in insecure directories.

Table 5.5 shows the total number of vulnerabilities by type. A single entrypoint may be vulnerable to more than one type of attack. We note that STING was able to find vulnerabilities of all types, including 7 that required race conditions.

Table 5.6 shows the various programs across which vulnerabilities were found. Interestingly, we note that 6 of the 24 vulnerable name resolutions in Ubuntu were found in Ubuntu-specific scripts. For example, CVE-2011-4406 and CVE-2011-3151 were assigned to two bugs in Ubuntu-specific scripts that STING found. Further, the programs containing vulnerabilities range from mature (e.g., cupsd) to new (e.g., x2go). We thus believe that STING can help in detecting vulnerabilities before an adversary, if run on test environments before they are deployed.
Table 5.5: Number and types of vulnerabilities we found. Race is the number of TOCTTOU vulnerabilities, where a check is made but the use is improper. A single entrypoint in Table 5.6 may be vulnerable to more than one kind of attack.

<table>
<thead>
<tr>
<th>Type of vulnerability</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symlink following</td>
<td>22</td>
</tr>
<tr>
<td>Hardlink following</td>
<td>14</td>
</tr>
<tr>
<td>File squatting</td>
<td>10</td>
</tr>
<tr>
<td>Untrusted search</td>
<td>6</td>
</tr>
<tr>
<td>Race conditions</td>
<td>7</td>
</tr>
</tbody>
</table>

Table 5.6: Number of vulnerable entrypoints we found in various programs, and the privilege escalation that the bugs would provide.

<table>
<thead>
<tr>
<th>Program</th>
<th>Vuln. Entry</th>
<th>Priv. Escalation DAC: uid-&gt;uid</th>
<th>Distribution</th>
<th>Previously known</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbus-daemon</td>
<td>2</td>
<td>messagebus-&gt;root</td>
<td>Ubuntu</td>
<td>Unknown</td>
</tr>
<tr>
<td>landscape</td>
<td>4</td>
<td>landscape-&gt;root</td>
<td>Ubuntu</td>
<td>Unknown</td>
</tr>
<tr>
<td>Startup scripts (3)</td>
<td>4</td>
<td>various-&gt;root</td>
<td>Ubuntu</td>
<td>Unknown</td>
</tr>
<tr>
<td>mysql</td>
<td>2</td>
<td>mysql-&gt;root</td>
<td>Ubuntu</td>
<td>1 Known</td>
</tr>
<tr>
<td>mysql_upgrade</td>
<td>1</td>
<td>mysql-&gt;root</td>
<td>Ubuntu</td>
<td>Unknown</td>
</tr>
<tr>
<td>tomcat script</td>
<td>2</td>
<td>tomcat6-&gt;root</td>
<td>Ubuntu</td>
<td>Known</td>
</tr>
<tr>
<td>lightdm</td>
<td>1</td>
<td>*-&gt;root</td>
<td>Ubuntu</td>
<td>Unknown</td>
</tr>
<tr>
<td>bluetooth-applet</td>
<td>1</td>
<td>*-&gt;user</td>
<td>Ubuntu</td>
<td>Unknown</td>
</tr>
<tr>
<td>java (openjdk)</td>
<td>1</td>
<td>*-&gt;user</td>
<td>Both</td>
<td>Known</td>
</tr>
<tr>
<td>zeitgeist-daemon</td>
<td>1</td>
<td>*-&gt;user</td>
<td>Both</td>
<td>Unknown</td>
</tr>
<tr>
<td>mountall</td>
<td>1</td>
<td>*-&gt;root</td>
<td>Ubuntu</td>
<td>Unknown</td>
</tr>
<tr>
<td>mailutils</td>
<td>1</td>
<td>mail-&gt;root</td>
<td>Ubuntu</td>
<td>Unknown</td>
</tr>
<tr>
<td>bsd-mailx</td>
<td>1</td>
<td>mail-&gt;root</td>
<td>Fedora</td>
<td>Unknown</td>
</tr>
<tr>
<td>cupsd</td>
<td>1</td>
<td>cups-&gt;root</td>
<td>Fedora</td>
<td>Known</td>
</tr>
<tr>
<td>abrt-server</td>
<td>1</td>
<td>abrt-&gt;root</td>
<td>Fedora</td>
<td>Unknown</td>
</tr>
<tr>
<td>yum</td>
<td>1</td>
<td>sync-&gt;root</td>
<td>Fedora</td>
<td>Unknown</td>
</tr>
<tr>
<td>x2gostartagent</td>
<td>1</td>
<td>*-&gt;user</td>
<td>Extra</td>
<td>Unknown</td>
</tr>
<tr>
<td><strong>19 Programs</strong></td>
<td><strong>26</strong></td>
<td></td>
<td></td>
<td><strong>21 Unknown</strong></td>
</tr>
</tbody>
</table>

MAC adversary model. We carried out similar experiments for a MAC adversary model on Fedora 16’s default SELinux policy. We assume an adversary limited only by the MAC labels, and allow the adversary permissions to run as the same DAC user. This is one of the aims of SELinux – even if a network daemon running as root gets compromised, it should still not compromise the whole system arbitrarily. However, we found that the SELinux policy allowed subjects we consider untrusted (such as the network-facing daemon sendmail_t) create permissions to critical labels such as etc_t. Thus STING immediately started reporting vulnerable name resolutions whenever any program accessed /etc. Thus, either the SELinux policy has to be made stricter, the adversary model must be weakened for mutual trust among all these programs, or all programs have to defend
Figure 5.5: Code from the GNU mail program in mailutils illustrating a squat vulnerability that Sting found.

themselves from name resolution attacks in /etc (which is probably impractical). This problem is consistent with the findings that /etc requires exceptional trust in the SELinux policy reported elsewhere [69].

5.6.1.2 Examples

In this section, we present particular examples highlighting Sting’s usefulness, and also broader lessons.

Mail Programs. GNU mail is the default mail client on Ubuntu 11.10, in which Sting found a vulnerability. This example shows the difficulty of proper checking in programs, and why detection tools with low false positives are necessary – programmers can easily get such checks wrong, and there are no standardized ways to write code to defend against various name resolution attacks.

The code shows the program preparing to read the file /var/mail/root. In summary, this program creates an empty file when the file doesn’t already exist (lines 4-10), using flags (O_EXCL) to ensure that a fresh file is created. The program performs several checks to verify the safety of the file opened, guarding against race conditions and link traversal (both symbolic and hard links) (11-29). Unfortunately, the program fails to protect itself against a squatting attack if the file already exists, as it does not check st_uid or st_gid; any user in group mail can control the contents of root’s inbox. Interestingly, it protects itself against squatting attacks on line 6.

```
01 */ filename = /var/mail/root */
02 */ First, check if file already exists */
03 fd = open (filename, O_CREAT|O_EXCL);
04 if (fd == -1) {
05    /* Create the file */
06    fd = open(filename, O_CREAT|O_EXCL);
07    if (fd < 0) {
08        return errno;
09    }
10 }
11  /* We now have a file. Make sure we did not open a symlink. */
12 struct stat filebuf, filebuf;
13 if (fstat (fd, &filebuf) == -1)
14   return errno;
15 if (fstat (filename, &filebuf) == -1)
16   return errno;
17 /* Now check if file and fd reference the same file, file only has one link, file is plain file. */
18 if (!(filebuf.st_dev == filebuf.st_dev
19    || filebuf.st_ino == filebuf.st_ino
20    || filebuf.st_nlink == 1
21    || (filebuf.st_mode & S_IFMT) == S_IFREG))
22   error ("Is must be a plain file with one link", filename);
23   close (fd);
24   return EINVAL;
25 }
26 if (fd < 0) {
27   close (fd);
28   return EINVAL;
29 }
30  /* If we get here, all checks passed.
31   Start using the file */
32 read(fd, ...)
```
X11 script STING found a race condition exploitable by a symbolic link attack on the script that creates /tmp/.X11-unix in Ubuntu 11.10. The code snippet is shown in Figure 5.6. The aim of the script is to create a root-owned directory /tmp/.X11-unix. Lines 6-8 check if such a file already exists that is not a directory, and if so, moves it away so a directory can be created. In Line 9, the programmer creates the directory, and assumes it will succeed, because the previous code had just moved any file that might exist. However, because /tmp is a shared directory, an adversary scheduled in between the moving of the file and the mkdir might again create a file in /tmp/.X11-unix, thus breaking the programmer’s expectation. If the file is a link pointing to, for example, /etc/shadow, the chmod on Line 11 will make it world-readable. STING was able to detect this race condition by changing the resource into a symbolic link after the move and before the creation on line 9, as it acts just before the system call on line 9. This script has existed for many years, showing how it is easy to overlook such conditions. This also shows how STING can synchronously produce any race condition an adversary can, because it is in the system. This script was independently fixed by Ubuntu in its latest release. The discussion page for the bug [110] shows how such checks are challenging to get right even for experienced programmers. Consequently, manually scanning source code can also easily miss such vulnerabilities.

mountall This program has an untrusted search path that is not executed in normal runtime but was discovered by STING. This Ubuntu-specific utility simply mounts all filesystems in /etc/fstab. When launched in an untrusted directory, it issues mount commands that search for files such as none and fusectl in the current working directory. If these are symbolic links, then the contents of these files are read through readlink, and put in /etc/mtab. Thus, the attacker can influence /etc/mtab entries and potentially confuse utilities that depend on this file, such as umounters. This is an example of how very specific conditions are required to detect the attack – the program needs to be launched in an adversarial directory, and the name searched for needs to be a symbolic link. Normal runtime would not give any hint of such attacks.

postgresql init script. This vulnerability highlights the challenge facing developers and OS
distributors. This script runs as root, and is vulnerable to symbolic and hard link attacks on accessing files in /etc/postgresql. That directory is owned by the user postgres, which could be compromised by remote attacks on PostgreSQL, who can then use this vulnerability to gain root privileges. The problem is that the developers who wrote the script did not expect the directory /etc/postgresql to be owned by a non-root user. However, the access control policy did not reflect this assumption. STING is useful in finding such discrepancies in access control policies as it can run with attacker models based on different policies.

5.6.1.3 False Positives

Two issues in STING cause false positives.

Random Name. The programs yum, abrt-server, zeitgeist-daemon in Table 5.6 were vulnerable to name resolution attacks, but defended themselves by creating files with random names. Library calls such as mktemp are used to create such names. STING cannot currently differentiate between “random” and “non-random” names. Exploiting such vulnerabilities requires the adversary to guess the filename, which may be challenging given proper randomness. In any case, such bugs can be fixed by adding the O_EXCL flag to open when creating files.

Program Internals. STING does not know the internal workings of a program. Thus, it cannot know if use of a resource from a vulnerable name resolution can affect the program or not, and simply marks it for further perusal. A vulnerable name resolution involving a write-like accept operation can always be exploited. However, whether those involving read can be exploited depends on the internals of the program. Eight of the 26 vulnerable name resolutions in Table 5.6 are due to read. While this has led to some false positives (two additional vulnerable name resolutions involving read not in Table 5.6 were deemed to not affect program functioning), STING narrows the programmers’ effort significantly. Nonetheless, more knowledge regarding program internals would improve the accuracy even further.

5.6.2 Performance Evaluation

We measured the performance of STING to assess its suitability as an online testing tool. While the performance of STING is of not of primary importance because it is meant to be run on test environments in non-production systems before deployment, it must nevertheless be responsive to online testing. We measured performance using micro- and macro-benchmarks. While STING does cause noticeable overhead, it did not impede our testing. All tests were done on a Dell Optiplex 980 machine with 8GB of RAM.

Micro-performance (Table 5.7) was measured by the time taken for individual system calls under varying conditions. For an attack launch, system call overhead is caused by the time to
<table>
<thead>
<tr>
<th>Case</th>
<th>Time (µs)</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attack Phase: open system call</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Base</td>
<td>14.57</td>
<td>–</td>
</tr>
<tr>
<td>+ Find Vulnerable Bindings</td>
<td>31.44</td>
<td>2.15×</td>
</tr>
<tr>
<td>+ Obtain entrypoint and check attack history</td>
<td>211.20</td>
<td>12.33×</td>
</tr>
<tr>
<td>+ Launch attack</td>
<td>365.87</td>
<td>25.1×</td>
</tr>
<tr>
<td>Detect Phase: read system call</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Base</td>
<td>8.73</td>
<td>–</td>
</tr>
<tr>
<td>+ Detect vulnerability</td>
<td>9.18</td>
<td>1.05×</td>
</tr>
<tr>
<td>+ Namespace recovery</td>
<td>63.08</td>
<td>7.22×</td>
</tr>
</tbody>
</table>

Table 5.7: Micro-overheads for system calls showing median over 10000 system call runs.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Base</th>
<th>STING</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache 2.2.20 compile</td>
<td>151.65s</td>
<td>163.79s</td>
<td>8%</td>
</tr>
<tr>
<td>Apachebench: Throughput</td>
<td>231.77Kbps</td>
<td>221.89Kbps</td>
<td>4.33%</td>
</tr>
<tr>
<td>Latency</td>
<td>1.943ms</td>
<td>2.088ms</td>
<td>7.46%</td>
</tr>
</tbody>
</table>

Table 5.8: Macro-benchmarks showing compilation and Apache-bench throughput and latency overhead. The standard deviation was less than 3% of the mean.

(1) detect adversary accessibility, (2) get and compare process entrypoint against attack history, and (3) launch the attack. The main overhead is due to obtaining the entrypoint to check the attack history and carrying out the attack. However, obtaining the entrypoint is required only if the name resolution is adversary accessible (around 4-5.7% in Table 5.4), and an attack is launched only once for a particular entrypoint, thereby alleviating their impact on overall performance. For the detection phase, we have (1) detect vulnerable access, and (2) rollback namespace. Namespace recovery through rollback is expensive, but occurs only once per attack launched.

Macro-benchmarks (Table 6.7) showed up to 8% overhead. Apache compilation involved a lot of name resolutions and temporary file creation. During our system tests, the system remained responsive enough to carry out normal tasks, such as browsing the Internet using Firefox and checking e-mail. We are investigating further opportunities to improve performance.

**Program retries and restarts.** We came across thirteen programs that retried a name resolution system call on failure due to a STING attack test case. The most common case was temporary file creation—programs retry until they successfully create a temporary file with a random name. Programs that retry integrate well with STING, which maintains its attack history and does not retry the same attacks on the same entrypoints. On the other hand, a few programs exited on
encountering an attack by Sting. We currently simply restart such programs (Section 5.4.2).
For example, dbus-daemon exited during boot due to a Sting test case and had to be restarted
by Sting to continue normal boot. However, programs may lose state across restarts. We are
investigating integrating process checkpoint and rollback mechanisms [108].

5.7 Related Work

Related work that deals with detecting name resolution attacks was presented in Section 5.2.2. Here, we discuss dynamic techniques to detect other types of program bugs, and revisit some
dynamic techniques that detect name resolution attacks.

Black-box testing. Fuzz testing, an instance of black-box testing, runs a program under
various input data test cases to see if the program crashes or exhibits undesired behavior. Part-
cular program entrypoints (usually network or file input) are fed totally random input with the
hope of locating input filtering vulnerabilities such as buffer overflows [111, 112, 113]. Black-box
fuzzing generally does not scale because it is not directed. To alleviate this, techniques use some
knowledge of the semantics of data expected at program entrypoints. SNOOZE [114] is a tool to
generate fuzzing scenarios for network protocols using which bugs in programs were discovered.
TaintScope [115] is a directed fuzzing tool that generates fuzzed input to pass checksum code in
programs. Web application vulnerability scanners [89] supply very specific strings to detect XSS
and SQL injection attacks.

We find a parallel can be drawn between our approach and directed black-box testing, where
semantics of input data is known. While such techniques change the data presented to a program
to exercise program paths with possible vulnerabilities, we change the resource, or the metadata
presented to the application for the same purpose. Thus, Sting can be viewed as an instance of
black-box testing that changes the namespace state to evaluate program responses.

Taint Tracking. Taint techniques track flow of tainted data inside programs. They can be
used to find bugs given specifications [116], or can detect secrecy leaks in programs [117]. Flax [118]
uses a taint-enhanced fuzzing approach to detect input filtering vulnerabilities in web applications.
However, taint tracking by itself does not actively change any data presented to applications, and
thus has different applications.

Dynamic Name Resolution Attacks Detection. As mentioned in Section 5.2.2, most
dynamic analysis are specific to detecting TOCTTOU attacks. Chari et al. [90] present an approach
to defend against improper binding attacks; however, they cannot detect them until they actively
occur in the system. We use active adversaries to generate test cases and to exercise potentially
vulnerable paths in programs to detect vulnerabilities that would not occur in normal runtime
traces. Further, none of the approaches deal with improper resource attacks, of which we detect several.

5.8 Discussion

Other System State Attacks. More program vulnerabilities may be detected by modifying system state. For example, non-reentrant signal handlers can be detected by delivering signals to a process already handling a signal. Similarly, return values of system calls can be changed to cause conditions suitable for attack (e.g., call to drop privileges fails). While STING could be easily extended to perform these attacks, we believe that these cases are more easily handled through static analysis, as standard techniques are available (e.g., lists of non-reentrant system calls, unchecked return value) through tools such as Fortify [119]. For the reasons we have seen, no such standard techniques are available for name resolution attacks.

Solutions. One of the more effective ways we have seen programs defending against improper binding attacks is by dropping privileges. For example, the privileged part of sshd drops privileges to the user whenever accessing user files such as .ssh/authorized_keys. Thus, even if code is vulnerable to improper binding attacks, the user cannot escalate privileges.

User directory. Administrators running as root should take extreme care when in user-owned directories, as there are several opportunities for privilege escalation. For example, we found during our testing that if the Python interpreter is started in a user-owned directory, Python searches for modules in that directory. If a user has malicious libraries, then they will be loaded instead. More race conditions are also exploitable as a user can delete even root-owned files in her directory.

Integration with Black-box Testing. We believe that STING can also integrate with other data fuzzing platforms [112]. Such tools need special environments (e.g., attaching to running processes with debuggers) to carry out their tests on running programs. Instead, we can take input from these platforms and use STING to feed such input into running processes. Since STING also takes into account the access control policy, opportunities to supply adversarial data can readily be located.

Deployment. We envision that STING would be deployed during Alpha and Beta testing of distribution releases. We plan to package STING for distributions, so users can easily install it through the distribution’s package managers. STING will test various programs as users are running them, and program vulnerabilities found can be fixed before the final release. Being a runtime analysis tool, STING can possibly find more vulnerabilities as it improves its runtime coverage. Even if a small percentage of users install the tool, we expect a significant increase in the runtime coverage, because different users configure and run programs in different ways.
5.9 Conclusion

In this chapter, we introduced STING, a tool that detects name resolution vulnerabilities in programs by dynamically modifying system state. We examine the deficiencies of current static and normal runtime analysis for detecting name resolution vulnerabilities. We classify name resolution attacks into improper binding and improper resource attacks. STING checks for opportunities to carry out these attacks on victim programs based on an adversary model, and if an adversary exists, launches attacks as an adversary would by modifying namespace state visible to the victim. STING later detects if the victim protected itself from the attack, or it was vulnerable. To allow online operation, we propose mechanisms for rolling back the namespace state. We tested STING on Fedora 15 and Ubuntu 11.10 distributions, finding 21 previously unknown vulnerabilities across various programs. We believe STING shall be useful in detecting name resolution vulnerabilities in programs before attackers. We plan to release and package STING for distributions for this purpose.
Chapter 6

Process Firewalls: Protecting Processes During Resource Access

6.1 Introduction

Programmers write their programs with expectations about the system resources they will obtain from particular system calls. For example, when writing a web server, programmers may expect to restrict the files served to a particular subtree in the filesystem (e.g., DocumentRoot in Apache), to use only system-approved shared libraries, and to create new files only the web server can access, even in shared directories (e.g., /tmp). However, adversaries have found several methods to direct victims to resources chosen by the adversary instead. We call these types of attacks resource access attacks. As a group, resource access attacks are less common than those based on memory errors (e.g., buffer overflows) or web programming errors (e.g., cross-site scripting), but they account for a steady stream of around 10% of reported vulnerabilities in the CVE [2] database (Table 6.1).

As an example, consider a typical web server that serves web content to authenticated users. Modern access control limits processes to least privilege [120] permissions, based on the functional requirements of the running program, but the web server needs to access the files containing its web content and password files for authenticating users. However, the program instructions that request these two sets of files are distinct, so the web server should not access the password file when it expects to access web content or vice versa. Adversaries can take advantage of flexibility in namespace resolution, ambiguity in environment variables, and plain old programmer errors to redirect the web server to violate such requirements. Access control cannot prevent such attacks because it treats all of the web server’s system calls equally.

On the one hand, it was thought that these resource access attacks could be prevented by better programmer practice and extended system-call APIs, but many vulnerabilities remain, even
Table 6.1: Resource access attack classes. For each attack class, we show its Common Weakness Enumeration (CWE \[1\]) number, and the number of reported vulnerabilities in the Common Vulnerability Exposure (CVE \[2\]) database.

<table>
<thead>
<tr>
<th>Attack Class</th>
<th>CWE class</th>
<th>CVE Count</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>&lt;2007</td>
</tr>
<tr>
<td>Untrusted Search Path</td>
<td>CWE-426</td>
<td>199</td>
</tr>
<tr>
<td>Untrusted Library Load</td>
<td>CWE-426</td>
<td>97</td>
</tr>
<tr>
<td>File/IPC squat</td>
<td>CWE-283</td>
<td>13</td>
</tr>
<tr>
<td>Directory Traversal</td>
<td>CWE-22</td>
<td>1057</td>
</tr>
<tr>
<td>PHP File Inclusion</td>
<td>CWE-98</td>
<td>1112</td>
</tr>
<tr>
<td>Link Following</td>
<td>CWE-59</td>
<td>480</td>
</tr>
<tr>
<td>TOCTTOU Races</td>
<td>CWE-362</td>
<td>17</td>
</tr>
<tr>
<td>Signal Races</td>
<td>CWE-479</td>
<td>9</td>
</tr>
</tbody>
</table>

| % Total CVEs | -        | 12.40%    | 9.41%     |

In several cases when the programmers use the extended APIs. Extended system-call APIs allow programmers to check the properties of files retrieved by pathnames, but these checks do not block all adversary threats. For example, Chari et al. \[8\] show that checks must be applied to each pathname component to prevent link following attacks, but \texttt{lstat} only checks whether the last component is a link. Several other resource access attacks depend on system knowledge for which no API exists. For example, a bug in the Debian installer (BID 17288) led to Apache module binaries being installed with insecure RUNPATH settings, which allowed insecure library loading.

On the other hand, researchers have also shown that system-only defenses to resource access attacks \[17, 35, 37, 38\] are limited \[121, 37\] or incur false positives because they lack an understanding of the process’s internal state \[12\]. In recent work, researchers propose using capabilities \[122\] to control the access to resources per system call using information flow control \[13, 41\]. However, the effectiveness of such defenses again depends on programmers using capabilities correctly, and it is often difficult or impossible to express defenses in practice using information flow (e.g., Time-of-Check-to-Time-of-Use TOCTTOU races and signal races).

This paper presents the Process Firewall, a kernel security mechanism designed to block resource access attacks system-wide. The Process Firewall examines the processes’ internal state to enforce \textit{attack-specific invariants} on each system call. By enforcing invariants on each system call, the Process Firewall provides attack-specific mediation that is analogous to a network firewall \[123\] for the system-call interface. Recall that prior to the introduction of network firewalls, host processes were trusted to protect themselves from network attackers. However, misconfigurations and programmer errors led to the need for a layer of defense to control hosts’ access to network resources. The network firewall does not guarantee the security of hosts because some risky network accesses may be allowed for functional reasons, but the network firewall can block attacks by limiting network access to approved processes and controlling how connections are constructed. The Process Firewall provides similar defenses for the system call interface, limiting the resources available to
particular system calls based on the process’s internal state. Using this approach, the Process Firewall is a single mechanism that unifies defenses for a variety of previously unrelated resource access attacks.

Because of differing goals and representations, the Process Firewall is *complementary to system-call interposition mechanisms* for sandboxing [83, 84, 87] and host intrusion detection [124, 125, 126], as well as access control in general. These mechanisms aim to *confine* malicious processes by mediating any operations that they request. These mechanisms cannot utilize the process’s internal state, such as its stack memory, because a malicious process could forge such state to circumvent confinement. In contrast, the Process Firewall only aims to *protect* benign processes from accessing resources that are not appropriate for their current state. Thus, the Process Firewall is able to use internal process state, in addition to detailed information about system resources already available to it, to enforce attack-specific invariants. If a process is actually malicious, it can only invalidate its own protection. Thus, the Process Firewall is not an alternative to system-call interposition mechanisms for confinement, but instead protects processes by blocking access to system resources that fail attack-specific invariants.

Designing the Process Firewall presents several challenges. First, we need a precise way to express the attack-specific invariants for resource access attacks. We find that a small number of resource and process attributes are sufficient to implement strong defenses from such attacks. Second, the low latency of the system-call interface demands methods to check invariants efficiently. We carefully design the Process Firewall’s invariant checking mechanism to avoid unnecessary processing time in extracting resource and/or process information and to only process invariants for attacks that are possible for each system call. Third, the Process Firewall must be practical for use on system-wide defenses. The Process Firewall supports both binary and interpreted programs. Fourth, the Process Firewall must be easy to use, requiring no program modification or additional user configuration. We describe a variety of methods for OS distributors to produce practical invariants in which they can manage or eliminate false positives.

In our experiments, we show that several types of attacks can be prevented, including two previously-unknown vulnerabilities. For example, we comprehensively defend the Joomla! content management system from PHP File Include attacks, and prevent a variety of Untrusted Library Load attacks. We find that the Process Firewall incurs less than 4% overhead on a variety of macronbenchmarks. The performance overhead on individual system-call processing varies from <3% for system calls not dealing with resource access to <11% for system calls that do. Surprisingly, the Process Firewall can actually *improve* program performance and security simultaneously, if program checks are replaced by equivalent Process Firewall rules, as demonstrated by Apache, which was able to handle 3-8% more requests.
<table>
<thead>
<tr>
<th>Safe Resource</th>
<th>Unsafe Resource</th>
<th>Attack Class</th>
<th>Process Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adversary Inaccessible (High Integrity, High Secrecy)</td>
<td>Adversary Accessible (Low Integrity, Low Secrecy)</td>
<td>Untrusted Search File/IPC Squat Untrusted Library PHP File Inclusion</td>
<td>Entrypoint</td>
</tr>
<tr>
<td>Adversary Accessible (Low Integrity, Low Secrecy)</td>
<td>Adversary Inaccessible (High Integrity, High Secrecy)</td>
<td>Link Following Directory Traversal</td>
<td>Entrypoint</td>
</tr>
<tr>
<td>Same as prev. “check”/“use”</td>
<td>Diff. from prev. “check”/“use”</td>
<td>TOCTTOU Races</td>
<td>Entrypoint + System-Call Trace</td>
</tr>
<tr>
<td>No signal (Blocked)</td>
<td>Adversary delivers signal</td>
<td>Non-reentrant Signal Handlers</td>
<td>System-Call Trace + In Signal Handler</td>
</tr>
</tbody>
</table>

Table 6.2: Resource access attacks are caused by adversaries forcing victim processes to access unsafe resources instead of safe resources. Also shown is necessary process context to determine when these attacks apply (see Section 6.4).

The result of this work is the Process Firewall system implemented for the Linux kernel, which includes: (1) a firewall rule language for expressing attack-specific invariants using attributes of processes and system resources; (2) mostly-automated methods for installing optimized Process Firewall rule bases; and (3) rule processing mechanisms specialized for system calls. Experiments show that the Process Firewall can be deployed with no user input or program changes to prevent a variety of attacks system-wide with a low likelihood of false positives.

### 6.2 Problem Definition

Processes require many system resources in order to function effectively. Resources may be delivered to processes either synchronously (e.g., files in response to `open`) or asynchronously (e.g., signals in response to `sigaction`). In either case, adversaries have several means for exploiting flaws when processes retrieve resources. Table 6.2 shows some resource access attacks, which occur when adversaries direct victim processes to resources chosen by the adversary. Columns 1 and 2 in Table 6.2 show the properties of the safe resource expected by the victim and the properties of unsafe resources to which adversaries direct victims for these resource access attacks.

In general, there are two ways by which adversaries can realize such attacks. First, an adversary can control how resources are retrieved directly. For example, when a victim program wants to read from a file in `/tmp`, the adversary who has write access to `/tmp` can create a symbolic link to `/etc/passwd`. If the victim does not detect that the target of the symbolic link is a high-secrecy file, the victim may end up accessing and possibly leaking the high-secrecy password file, when it meant to access a low-secrecy temporary file. Second, the adversary can indirectly control the resource accessed by tricking the victim process into requesting a resource specified by the adversary. For example, in a Directory Traversal attack [127], an adversary may request the file
Figure 6.1: Program code to defend against resource access attacks.

```c
/* fail if file is a symbolic link */
int open_no_symlink(char *fname)
{
    struct stat lbuf, buf;
    int fd = 0;
    lstat(fname, &lbuf);
    if (S_ISLNK(lbuf.st_mode))
        error("File is a symbolic link!");
    fd = open(fname);
    fstat(fd, &buf);
    if ((buf.st_dev != lbuf.st_dev) ||
        (buf.st_ino != lbuf.st_ino))
        error("Race detected!");
    lstat(fname, &lbuf);
    if ((buf.st_dev != lbuf.st_dev) ||
        (buf.st_ino != lbuf.st_ino))
        error("Cryogenic sleep race!");
    return fd;
}
```

```c
/* ignore malicious env var */
int load_library()
{
    if ((uid != euid) || (gid != egid)) {
        /* SUID binary */
        unsetenv("LD_LIBRARY_PATH");
        unsetenv("LD_PRELOAD");
    }
    path = build_search_path();
    foreach path p {
        fd = open(p/lib);
        if (fd != -1) {
            mmap(fd);
            break;  
         } 
    }
}(a) Symbolic link check
(b) Library search path check
```

An important characteristic of resource access attacks is that a resource that is unsafe for a particular victim context is safe for some other victim context. In the above example, the webserver can access `/etc/passwd` legitimately when it wants to authenticate clients. However, it should not do so when serving a user web page. As a result, traditional access control is insufficient to prevent resource access attacks, because it assigns permissions to processes as a whole, not distinguishing between what is safe or unsafe for different victim context.

### 6.2.1 Challenges in Preventing Resource Access Attacks

Figure 6.1 shows sample program code that aims to prevent resource access attacks to highlight the challenges. To prevent these attacks, victim programs either: (1) retrieve and check resource properties (first example) or (2) restrict the name used to retrieve the resource (second example).

**Link Traversal and TOCTTOU Races.** Figure 6.1(a) shows code from a program trying to defend itself against symbolic link attacks. On line 3, an `lstat` checks if the file is a symbolic link. If not, on line 6, the file is opened to read using the `open` system call. However, there is a race condition possible between lines 3 and 6. A TOCTTOU attack [32, 17] can be successful if the adversary forces the victim to “use” a different file on line 6 than its previous “check” on line 3. Row 3, Columns 1 and 2 in Table 6.2 show that an unsafe resource for a successful TOCTTOU attack is different from the previous corresponding “check” or “use” call, whereas the safe resource that protects against the attack is the same as the corresponding call. An adversary scheduled by the OS to run after line 3 but before line 6 could change the file to a symbolic link. To defend against this, an `fstat` is performed on line 7, and the inode and device numbers that uniquely identify

`.../.../etc/passwd` from a webserver. If the webserver does not properly filter the untrusted input or validate the retrieved resource, it could again end up accessing the high-secrecy password file when it meant to access a low-secrecy user web page.
a file on a device are compared to the `lstat` to make sure the file checked is the one opened on lines 8 and 9. However, even this is insufficient. Olaf Kirch [33] showed a “cryogenic sleep” attack, in which an adversary could put a `setuid` process to sleep before line 6 but after line 3, and wait for the inode numbers to recycle, thus passing the checks on lines 8 and 9. To defend this, an additional `lstat` is done on line 11, and the inode and device numbers compared again. So long as the file remains open, the same inode number cannot be recycled. Finally, Chari et al. [8] showed that such similar checks must be done for each pathname component, not only the final resource, and proposed a generalized `safe_open` function that allows following of links so long as the link points to an adversary’s own files and not the victim’s files. Unfortunately, `safe_open` is used by few programs currently (Postfix uses a weaker version) and has false negatives¹.

**Untrusted Search Path.** Figure 6.1(b) shows simplified code from `ld.so` when it loads a library. An adversary, when launching a `setuid` program, may supply malicious path values for the `LD_LIBRARY_PATH` environment variables, forcing a victim program to use untrusted libraries. In this example, an untrusted library load resource attack succeeds if the adversary forces the victim to access a low-integrity resource on line 8 where the victim program was expecting a high-integrity resource (as shown in Columns 1 and 2 of row 1 in Table 6.2). To prevent this, `ld.so` unsets such environment variables on lines 3 and 4, builds a search path on line 6, opens the library file on line 8, and maps it into the process address space on line 10. Unfortunately, there are a variety of other ways that adversaries can control names in search paths apart from environment variables – RUNPATHs in binaries, programmer bugs, and even dynamic linker bugs (e.g., CVE-2011-0536, CVE-2011-1658, Payer et al. [128]). It is difficult for programmers to anticipate and restrict all sources of adversarial names for library search paths. For example, a bug in the Debian installer (CVE-2006-1564) led to Apache module binaries being installed with insecure RUNPATH settings, which allowed insecure library loading.

### 6.2.2 Limitations of Prior Defenses

Prior defenses can be divided into two broad categories: (1) system-only defenses and (2) program defenses. First, system-only defenses require no program modifications and are deployed either as libraries or in-kernel defenses. System-only defenses have been proposed for TOCTTOU [17, 35, 37, 38, 129] and link following [8] attacks. However, system-only defenses in general are fundamentally limited because they do not take into account *process context*. Process context captures the process’s intent and therefore the set of valid resources for the process’s particular system call. For example, Cai et al. [12] proved that without this process context, all system-only TOCTTOU defenses are prone to false positives or negatives.

¹In `safe_open`, an adversary A can trick V into accessing another victim B’s files through A’s symbolic links
On the other hand, while program code defenses can limit resource access per system call depending on process context, programs lack sufficient visibility into the system to defend against resource access attacks. Since resources are managed outside the program in OS namespaces, programs need to query the OS to ensure that they access the correct resource. However, there are several difficulties in doing so. First, such program checks are complicated under the current system-call API. As one example, the system-call API that programs use for resource access is not atomic, leading to TOCTTOU races. There is no known race-free method to perform an access-open check in the current system call API [12]. As another example, Chari et al. [8] show that to defend link following attacks, programmers should perform at least four additional system calls per path component for each resource access. Second, as a consequence of requiring additional system calls, program defenses are also inefficient. For example, the Apache webserver documentation [4] recommends switching off resource access checks during web page file retrieval to improve performance. Thirdly, program checks are incomplete, because adversary accessibility to resources is not sufficiently exposed to programs by the system-call API. The first two rows of Table 6.2 show that adversary accessibility is necessary to identify unsafe resources for some attacks. Currently, programs can query adversary accessibility for only UNIX discretionary access control (DAC) policies (e.g., using the access system call), but many UNIX systems now also enforce mandatory access control (MAC) policies (e.g., SELinux [29] and AppArmor [73]) that allow different adversary accessibility. Finally, many programmers are unaware of resource access attacks and fail to add checks altogether. All these factors have led to resource access attacks making up around 10% of CVE entries (Table 6.1).

Program defenses such as privilege separation [131] and namespace isolation (using chroot), capability systems [122, 15], and information flow systems [13, 41] enable customized permission enforcement per system call. However, such solutions are manually intensive because programmers must tailor their programs to block resource access attacks. More importantly, these solutions are not portable because different deployments may have different adversary accessibility (determined by the access control policy), therefore having the impractical requirement of rewriting programs for each distinct system deployment to provide effective protection. In addition, some of these defenses do not address temporal properties of processes, necessary to block TOCTTOU and signal races. Lastly, privilege separation that uses the current system-call API inherits challenges of inefficiency [132] and complexity.

In this work, we want to develop a mechanism that protects each system call from resource

---

2A resource is adversary accessible if the OS access control policy grants an adversary of the current process permissions to the resource. In UNIX discretionary access control (DAC), an adversary is a user with a different UID (except root). A similar approach can be applied to calculate adversaries for other security models, such as SELinux mandatory access control (MAC) [130]. Write permissions to the resource lead to integrity attacks and read permissions to secrecy attacks.
access attacks. Noting the incompleteness, complexity, and inefficiency of program code checks, our solution has the following goals: (1) must be capable of preventing instances of the resource access attacks in Table 6.2; (2) must require no programmer or user effort to be deployed; (3) must be possible to configure useful policies that do not introduce false positives; and (4) must be more efficient than program-based defenses. That is, the Process Firewall must be an efficient and easy-to-use mechanism that prevents many resource access attacks without blocking valid function.

6.3 Solution Overview

Our solution is based on the insight that past defenses have been incomplete in stopping resource access attacks because they did not simultaneously consider both the program request’s particular requirements and system knowledge about adversary access to protect processes during resource access. While system-only defenses do not consider the context of a program’s particular request, program defenses do not have the system knowledge of adversary accessibility to resources. In this work, we propose a system-based solution, the Process Firewall, to protect programs against resource access attacks by augmenting the system’s knowledge about adversary access with process context.

Our solution augments the OS kernel’s authorization mechanism to account for the current process context and accessed resource context, such as adversary accessibility, as shown in Figure 6.2. First, during a setup phase (Step 0 in Figure 6.2), invariant rules (simply, invariants) are defined for attack types in Table 6.2 to form an invariant database. These invariants describe the preconditions, in terms of the process and resource contexts, under which a particular type of resource access attack is possible. While any process and resource context may be applied to an invariant in general, Table 6.2 shows that only a few of these (as explained in Section 6.4.2) are sufficient to block many resource access attacks. During runtime, when a process makes a system call, the OS authorization mechanism, a reference monitor [72], decides whether a subject responsible for the system call (e.g., the process) can perform the requested operations on the specified
object (e.g., the resource) (Step 1 in Figure 6.2) using its access control policy. If the authorization mechanism allows a process access to the requested resource, the Process Firewall is invoked to block the resource access if it would result in any resource access attack (Step 2 in Figure 6.2). Step 3 of Figure 6.2 fetches invariants from the invariant database. Resource and process context to evaluate invariants is fetched using context modules (Step 4 in Figure 6.2). If the invariant precondition matches, the Process Firewall implements the specified action. In general, invariants are deny rules, where the default action is to allow the resource access (i.e., because no resource access attack was found to be possible in that process context for the target resource).

As an example, consider a setuid root binary vulnerable to an untrusted library search path attack (Figure 6.1(b)). Suppose the dynamic linker ld.so makes a system call on line 8 to load an adversary-accessible library. Since root processes are authorized to access any file, the authorization mechanism allows the process access to the library. The Process Firewall is then invoked (Step 2 in Figure 6.2). The Process Firewall fetches invariants from the invariant database (Step 3 in Figure 6.2). Assume that one invariant blocks the open system call in line 8 of ld.so from accessing adversary-accessible resources. To evaluate this invariant, context modules are invoked (Step 4 in Figure 6.2) to retrieve required process context (e.g., the process’s user stack and the mapping of the file ld.so in the process) and resource context (e.g., that the resource is adversary-accessible) using the applicable context modules for the invariant’s preconditions. In this case, the Process Firewall finds that the invariant precondition matches and thus blocks the resource access (Step 5 in Figure 6.2).

We identify a key difference between our goals and those of access control and host intrusion detection systems (IDSes) [124, 125, 126] that enable the deployment of such defenses in the kernel. Preventing resource access attacks only requires that we protect the process from unsafe resources. For both access control and host IDS systems, the goal is to confine a potentially malicious process, so they cannot act on any process context for fear of it being spoofed by the adversary. For example, host IDSes model the expected behavior of programs and compare the externally-visible behavior (e.g., process’s system calls) to these models to detect intrusions. Unfortunately, they cannot trust any of the process’s internal state because malicious processes can mimic a legitimate program [133]. In contrast, a malicious process that mimics another program to our system only affects its own protection, and access control still confines its operation. Section 6.4.4 describes how our mechanism protects itself from malicious or misbehaving processes during memory introspection.

6.4 Design

In this section, we describe the steps shown in Figure 6.2.
6.4.1 Defining Attack-Specific Invariants

To start (Step 0), we need to define what an attack-specific invariant is. These invariants define: (1) a resource context, which describes the properties of a resource that may indicate a resource access attack is in progress, (2) a process context, which describes the properties of a process that would be vulnerable to the corresponding resource access attack implied by the resource context, and (3) an authorization decision to take when context matches (allow or block access). In Step 0 of Figure 6.2, known attacks, of the types summarized in Table 6.2, are translated into a database of attack-specific invariants. Invariants for attack types are either manually defined once across all deployments, or automatically generated for the specific deployment (Section 6.6.3). The challenge in this section is to define the format of the invariants necessary to prevent all these types of attacks. We find very few types of contextual information are necessary to express the necessary invariants for our current resource access attack classes: two for resource contexts and three for process contexts (Table 6.2).

Below, we define a Process Firewall attack-specific invariant as a function. Since Process Firewall attack-specific invariants augment access control, we will start with access control. An access control function takes as input subject label, an object label, and an operation, and returns whether access is allowed or denied.

\[
\text{authorize}(\text{subject}, \text{object}, \text{op}) \rightarrow Y|N
\]

In contrast, a Process Firewall attack-specific invariant for preventing resource-access attacks augments the conventional authorization function by making decisions also depend on process and resource context:

\[
\text{pf.invariant}(\text{subject, entrypoint, syscall.trace}, \text{object, resource.id, adversary.access}, \text{op}) \rightarrow Y|N
\]

Attacks are caused when an unsafe resource is returned instead of the safe resource for a particular process context. Column 2 in Table 6.2 shows that the required resource context to identify unsafe and safe resources are the resource identifier and adversary accessibility. Column 4 in Table 6.2 shows that the required process context is the program entrypoint and/or prior system calls executed by the process. Thus, to detect whether an invariant applies we may need to identify the program entrypoint and/or some part of the system call trace. Hence, in this case, a subject, entrypoint, and syscall trace identify the process context for detecting resource access attacks. An entrypoint is the program counter of a function call instruction on the process’s call stack. For examples in Section 6.2.1, it can be thought of as identifying the line number of the function call in the program.
We use `pf.invariant` to block unsafe resources rather than allowing safe resources. This follows from our design decision to prevent false positives, at the cost of possibly allowing false negatives. By definition of the attack-specific invariants, any unsafe resource enables an exploit; therefore, there can be no false positives.

### 6.4.2 Checking Invariants

In Step 3, the Process Firewall’s main function is retrieving process and resource contexts, and checking them against the invariants to verify that the resource request is not an attack. A naïve design simply fetches all process and resource contexts and then matches them against each invariant. Since context retrieval incurs overhead, we want to retrieve them only when necessary. Firstly, to prevent unnecessary context collection, we lazily retrieve context values. Secondly, to preserve and reuse gathered context as long as it is valid, we support module-specific caching.

Lazy context retrieval gathers context only when it is needed by an invariant. The Process Firewall associates each context field with a bit in a `context bit mask` that shows which context field values have already been collected. To enable modular context retrieval, we designed `context modules`. Each context module retrieves one context field value. When evaluating an invariant, the rule matching mechanism checks if all necessary context field bits are set. If not, it triggers the associated context module.

Computed context field values themselves are then cached for reuse. Once we have obtained and stored a context value, this value may apply across other rules and even across multiple invocations of the Process Firewall. For example, the process call stack used to find program entrypoints is valid throughout a single system call, but multiple resource requests may be made (e.g., in pathname resolution). Context modules must support an API to check whether to invalidate their context at the beginning or end of rule processing.

### 6.4.3 Finding Applicable Invariants

As the size of our invariant database grows, sequential evaluation becomes impractical. Several systems, such as Windows access control lists and network firewalls sequentially scan rules until a match is found, leading to long authorization times for large rule sets. To combat this problem, network firewalls provide facilities to organize rules into `chains`, enabling only applicable rules to be run. The problem is such chains are usually manually configured in network firewalls.

To solve the problem of sequential rule traversal and manual configuration, we automatically create chains for entrypoints. Because nearly all our invariants are associated with a specific entrypoint, we organize our invariants into `entrypoint-specific chains` and traverse the chain specific to an entrypoint. Thus, the Process Firewall determines the entrypoint associated with this resource
request and traverses only that chain. Rules that do not involve entrypoints are matched before jumping to entrypoint-specific chains.

This simple traversal arrangement is possible because we have only deny rules (Section 6.4.1) followed by a default allow rule. If we had both deny and allow rules, then the order of traversal of rules would be important, and rule organization also would also become more complicated as in network firewalls.

6.4.4 Retrieving Entrypoint Context

In Step 4, we run the context modules necessary for the current invariant being checked, as described in Section 6.4.2. The only context values that need to be retrieved from the processes’ internal states are the entrypoint contexts. The Process Firewall is designed to protect processes from resource access attacks system-wide, so we need to be able to retrieve entrypoint contexts from multiple types of programs being run on the system. In this section, we explore how to safely retrieve entrypoint contexts for different types of programs.

Our entrypoint context modules should handle both binary and interpreted programs. For binaries, the challenge is to reason correctly about compiler optimizations, such as tail-call elimination, and compile-time options, such as those that remove frame-pointer information. In these cases, we can still retrieve the call stack if debug or exception handler information is available (e.g., Ubuntu by default compiles all programs with exception handler information). In case such information is unavailable, we fall back to producing a stack trace using function prologue information (e.g., as used by GDB). For interpreted programs\(^3\), we adapt the backtrace code from the interpreter to run in the kernel. This is only a small amount of code, ranging from 11 lines for PHP to 59 lines for Bash. We have not found any programs in the Ubuntu 10.04 desktop distribution or LAMP stack for which the call stack is not retrieved correctly by these methods.

Since context modules obtain data from potentially malicious processes, they need to perform careful input sanitization to avoid arbitrary kernel compromise through invalid pointer dereferences or denial-of-service (DoS) attacks such as unwinding infinite call stacks. Our entrypoint context prevents invalid pointer dereferences by using the kernel’s \texttt{copy\_from\_user} function. Further, to prevent DoS attacks, it sets an upper limit on the number of stack frames. These techniques ensure the Process Firewall aborts evaluation of malformed context without itself exiting or functioning incorrectly. As a result, an applicable rule may mistakenly not match for a malicious process, but this only affects the malicious process’s protection.

\(^3\text{We support the Bash, PHP, and Python languages thusfar.}\)
6.5 Implementation

We have implemented versions of the Process Firewall for Linux kernel versions 2.6.35 and 3.2.0. We find that the Process Firewall performs a service similar to a network firewall. The Process Firewall controls access to specific resources (e.g., by resource identifier) or groups of resources (e.g., by label) by specific ports (e.g., entrypoints) in a stateful way (e.g., system-call traces). As a result, we construct the Process Firewall by adapting the `iptables` firewall mechanism. The main benefit we derive from the `iptables` architecture is the extensibility of the rule language and modules to new attacks, just as `iptables` is extensible to new protocols. Thus, we can extend the system-call API modularly to arbitrary process and resource contexts and new attacks without affecting core kernel code.

Adding code to the kernel increases the TCB and also raises questions of maintainability. The Process Firewall core consists of 1102 LOC (501 LOC for rule traversal and matching, 378 LOC for validating and setting up rules pushed in from userspace into chains, and 223 LOC for initialization and module registration) with 2451 LOC for modules (dominated by the entrypoint context module with 1735 LOC). Bugs in interaction with userspace (validating rules and fetching entrypoints) may cause kernel compromise. However, the core code (including rule validation) is mostly borrowed from the mature `iptables`, and the entrypoint context module (modified from a proposed kernel patch [134]) carefully handles interactions with potentially malicious userspace as described in Section 6.4.4. Thus, we believe the increase in TCB is acceptable. Finally, since Process Firewall code is divorced from the mainline kernel code, it is easily maintainable. Porting from the Linux 2.6.35 to Linux 3.2.0 kernel required minimal effort.

6.5.1 Process Firewall Rule Processing Loop

Figure 6.3 shows an overview of the Process Firewall system, where the components added for the Process Firewall are shaded. When a user-level process submits a system call request, the kernel’s existing authorization system (e.g., SELinux [29] over the Linux Security Modules interface [56]) uses the host’s access control policy to authorize the set of security-sensitive operations resulting from each system call. If an operation is authorized, the Process Firewall is then invoked through a `PF hook` to further determine whether the resource is appropriate for that particular process context based on the specified invariants. These invariants are stated in terms of firewall-style rules stored in the `PF Rule Base`. We use LSM for the Process Firewall rather than build on system-call interposition techniques [87, 83] as LSM has no race conditions [135] and has been analyzed for complete mediation to resources [136, 137].

Once invoked, the Process Firewall starts processing the first rule in the rule base. A rule
matches a “packet” if all its classifiers (Table 6.3) match values in the “packet.” A rule can contain several classifiers, and user-defined classifiers can be added through extensible match modules, similar to how iptables extensibly handles network protocols.

In a network firewall, the packet to match is readily available. The Process Firewall constructs its “packet” by fetching information required by match modules from the process and resource through context modules. Context collected is recorded using a bitmask. If a rule matches, target modules are invoked, which either produce a decision to be returned to the authorization system, ask to continue processing the next rule, or jump to a new chain of rules. This is again similar to iptables, where different target modules (called jumps) can accept or drop packets. If a rule is not matched, processing continues on the next rule.

Those wishing to write new match and target modules must write both a userspace part that handles rule-language extensions, and a kernel handler function, similar to iptables. For the Process Firewall, module writers also have to implement the necessary context modules to obtain required context for their match and target modules.

One critical issue is that iptables is not re-entrant – it saves the stack of chain traversals of a packet with a table. Thus, any re-entry invalidates this stack. To defend against this, iptables turns off kernel pre-emption and interrupts. However, we noticed that disabling interrupts on
each resource request, perhaps several times per system call, has a noticeable impact on system interactivity. Further, we found that such disabling also leads to overhead in the performance-critical main loop.

To solve this issue, we designed the Process Firewall to run with interrupts enabled. Instead of saving the stack of rule traversals with the table, as iptables does, we instead maintain a per-process rule state, by extending struct task_struct. The process can thus be safely scheduled out during rule-base traversal.

6.5.2 Process Firewall Rule Language

Table 6.3 shows the rule language of the Process Firewall. This is analogous to iptables. There are statements that identify the entire firewall (pftables), tables, and chains (described above). Individual rules have the same structure as an iptables rule, consisting of default matches (def_match), custom matches (match), and targets. However, these rule elements are specific to the Process Firewall, representing the difference between the network firewall concepts and those used in the Process Firewall. For example, default matches specify the five context values: (1) process label; (2) resource label; (3) resource identifier (signal or inode number); (4) program binary; and (5) entrypoint (program and entry_point). Entrypoint program counters are specified relative to program binary base, handling ASLR code randomization. The special keyword SYSHIGH denotes the set of all trusted computing base (TCB) subjects (for -s) or objects (for -d) for SELinux [138, 130]. Match and target modules in a rule can refer to a context in their arguments (e.g., C_INO for inode number); this is replaced by the actual context value at runtime.

We have developed several match, target and context modules to handle the resource access attacks in Table 6.2. The STATE match and target modules allow matching and setting arbitrary

---

4The SELinux MAC system assigns labels to processes (subject labels) and resources (object labels). For example, the process sshd has the label sshd_t while the file /etc/shadow has the label shadow_t. In SELinux, the relevant part of all subject and object labels are called types. The _t denotes a type.
key-value pairs in a process-specific (in Linux, also thread-specific) dictionary, implemented by extending the **struct task_struct**. This stores, for example, the resource identifier (inode number) accessed in previous system calls to defend TOCTTOU attacks, and if the process is currently handling a signal, to defend signal races. The **LOG** target module logs a variety of information about the current resource access in JSON format.

The Process Firewall rules are inserted into the kernel by the **pftables** user-space process. The **PF rule setup** module translates input rules into an enforceable form and organizes them for efficient access in the **PF rule base**. Each chain in the Process Firewall has rules that block unsafe resources (**-j DROP**) followed by a default allow policy (**-j ACCEPT**). In addition, it translates filenames into inode numbers and SELinux security labels into security IDs for fast matching.

### 6.6 Evaluation

In this section, we examine the Process Firewall’s ability to block exploits, methods to generate rules for the Process Firewall in a manner that does not produce false positives, and Process Firewall performance.

#### 6.6.1 Security Evaluation

To evaluate the effectiveness of the Process Firewall in blocking resource access attacks, we deployed it on an Ubuntu 10.04 distribution and tried to exploit resource access attacks against programs. We tested 9 exploits shown in Table 6.4 that are representative instances of many resource access exploits. Four exploits (E1-E4) were chosen to check effectiveness of four rules that were automatically suggested by our rule suggestion procedure (Section 6.6.3). Note that these rules were suggested with no knowledge of the exploits we tested them against. E5 tested the manually-inserted, non-reentrant signal handler rules (R8-R11). E6, E7 were chosen to check effectiveness of rules that were automatically generated from known vulnerabilities, to externally protect unpatched programs. E8, E9 were new vulnerabilities automatically blocked by the Process Firewall itself. We verified that all exploits were successful when the Process Firewall was disabled. When enabled, the Process Firewall successfully blocked these resource access attacks.

#### 6.6.1.1 Common Vulnerabilities

**E1: Apache.** CVE-2006-1564 is an untrusted library load vulnerability based on an insecure RUN-PATH discussed in Section 6.2.1. To simulate this condition, we manually set RPATH to the insecure value. Rule R1 blocked this attack, as the SELinux label of **/tmp/svn** (**tmp_t**) was not in
Table 6.4: The exploits tested against the Process Firewall.

<table>
<thead>
<tr>
<th>#</th>
<th>Program</th>
<th>Reference</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1</td>
<td>Apache</td>
<td>CVE-2006-1564</td>
<td>Untrusted Library</td>
</tr>
<tr>
<td>E2</td>
<td>dstat</td>
<td>CVE-2009-4081</td>
<td>Untrusted Search Path</td>
</tr>
<tr>
<td>E3</td>
<td>libdbus</td>
<td>CVE-2012-3524</td>
<td>Untrusted Search Path</td>
</tr>
<tr>
<td>E4</td>
<td>Joomla! gCalendar</td>
<td>CVE-2010-0972</td>
<td>PHP File Inclusion</td>
</tr>
<tr>
<td>E5</td>
<td>openssh</td>
<td>CVE-2006-5051</td>
<td>Signal Handler Race</td>
</tr>
<tr>
<td>E6</td>
<td>dbus-daemon</td>
<td>Unpatched</td>
<td>TOCTTOU</td>
</tr>
<tr>
<td>E7</td>
<td>java</td>
<td>Unpatched</td>
<td>Untrusted Search Path</td>
</tr>
<tr>
<td>E8</td>
<td>Icecat</td>
<td>Unknown</td>
<td>Untrusted Library</td>
</tr>
<tr>
<td>E9</td>
<td>init script</td>
<td>Unknown</td>
<td>Link following</td>
</tr>
</tbody>
</table>

Table 6.5: Process Firewall rules discussed in text (R1 - R11) and templates used to generate rules (T1, T2).

the set of valid labels for the ld.so entrypoint that opens library files. Section 6.2.1 listed many other reasons for untrusted library search paths – all are blocked by the single rule R1.

E2: dstat. dstat is a Python script that outputs a variety of performance statistics. It had an untrusted module search path (Python os.path) that included the working directory, enabling adversaries to plant a Trojan horse Python module. Rule R2 constrains Python scripts to load only trusted Python scripts labeled/usr_t, lib_t, corresponding to /usr/lib/ and /usr/share directories, which blocked this attack. Python programmers are often the cause for such bugs, but
other reasons exist – in 2008, the Python interpreter itself set insecure search paths (CVE-2008-5983), affecting a variety of scripts. All such attacks are blocked by R2.

**E3: `libdbus`**. D-Bus is a message bus system that various applications use to communicate. Clients use `libdbus` to talk to the D-Bus server socket. However, `libdbus` programmers did not expect to be called from `setuid` binaries, so they did not filter an environment variable that specifies the path of the D-Bus system-wide socket. This is a typical example of programmer assumptions not being met by system deployment, leading to a resource access attack. Rule R3 restricts the entrypoint in `libdbus` to connect to only the trusted message bus labeled `system_dbus_var_run_t` (in directory `/var/run/dbus`) for high-integrity processes, thus blocking the attack for all vulnerable `setuid` programs.

**E4: PHP scripts**. PHP local file inclusion (LFI) is a widespread attack caused by improper input filtering in PHP scripts, causing the PHP interpreter to load attacker-specified untrusted code. We setup Joomla!, a popular content management system written in PHP. A large number of third-party modules have been written for Joomla!, many improperly filtering input filenames, enabling the adversary to launch PHP LFI attacks (e.g., 82 CVEs in 2010 alone). Rule R4 restricts the instruction including files in the PHP interpreter to only open those of appropriate SELinux labels (`httpd_user_script_exec_t` by default on Ubuntu). We tested that an attack on the gCalendar component (Table 6.4) was blocked, but R4 should block all such attacks.

Openssh (E5) has a non-reentrant signal handler vulnerability that can be blocked using one set of system-wide rules (R9-R12). D-Bus (E6) and the Java compiler (E7) had unpatched vulnerabilities (E7 was known for at least two years but still unpatched). Rules R5, R6 in Table 6.5 block E6, and rule R7 blocks E7. The Process Firewall rules thus perform a function similar to dynamic binary patching [139].

### 6.6.1.2 New Vulnerabilities Found

**E8: GNU icecat**. One of the machines on which we installed the Process Firewall had the GNU Icecat browser. This had an insecure environment variable that caused it to search for libraries in the current working directory. The Process Firewall silently blocked this attack (rule R1); we noticed it later in our denial logs. We reported it to the maintainer who accepted our patch.

**E9: init script**. When examining accesses matching our `safe_open` rules (that we apply system-wide), we found one Ubuntu init script that unsafely created a file. The bug was accepted by Ubuntu and assigned a CVE.
6.6.2 Process Firewall Performance

We examine here the performance impact of the Process Firewall in two ways: (1) comparing the performance impact of blocking resource access attacks in the program vs. the Process Firewall and (2) the performance overhead of the Process Firewall relative to an unprotected system. First, we found that enforcing strong defenses against resource access attacks in the Process Firewall could be done much more efficiently in the Process Firewall than in programs. Second, we found that the Process Firewall including over 1000 rules incurs no more than a 4% overhead over a variety of macrobenchmarks and less than 11% overhead on any one system call.

System Call Performance. Figure 6.4 shows the performance of four variants of the open system call as a function of path length that provide differing protections against link following attacks. The average path length for our system was 2.3. The baseline open does not perform any checks. open_nofollow uses the O_NOFOLLOW flag, which prevents such attacks, but is non-portable and may also block desirable uses of symbolic links. open_nolink opens a file if it is not a link, by the sequence lstat-open. open_race eliminates the race between lstat and open by performing an additional fstat after opening the file. Lastly, safe_open performs such checks for each path component and is necessary to completely prevent link following attacks [8]. safe_open_PF is the equivalent of safe_open implemented using Process Firewall rules. While safe_open had overheads of up to 103% over the baseline open (for n = 7), our equivalent in the Process Firewall had a maximum overhead of only 2.3%. The overhead of safe_open is because it needs to perform at least 4 additional system calls for each path component. This shows how Process Firewall rules can powerfully extend the system call API with arbitrary resource constraints, while still maintaining performance. This also eliminates the need for checks in code, and thus, races.

Removing Program Checks. We use the Apache webserver to examine performance benefit in moving checks out of the program into the system at a macro-level on real-world programs. To protect against symbolic link following vulnerabilities, the option SymLinksIfOwnerMatch constraints Apache to follow a symbolic link only if both the symbolic link and the target file it refers
to are owned by the same user. This reduces performance by forcing Apache to perform additional `lstat` system calls on each component of the pathname. Thus, the Apache documentation [4] recommends switching this option off for better performance. Furthermore, the documentation notes that this option can actually be circumvented through races.

Figure 6.5 compares the performance of these checks against an equivalent Process Firewall rule enforcing `SymLinksIfOwnerMatch` (R8 in Table 6.5). As both path length and number of concurrent clients increase, we note a performance improvement of Process Firewall rules over program checks – for a path length of 1 (`/index.html`), we noted a performance improvement of 3.02% for 200 concurrent clients; for path lengths 3, 5, and 9, it is 4.12%, 6.35% and 8.36%, respectively. The Process Firewall rule is thus both more efficient and secure.

**Process Firewall Performance.** We perform experiments on a Process Firewall implemented for Linux kernel 2.6.35 on a Dell Optiplex 980 with 2GB of RAM. Table 6.7 documents our macrobenchmarks. We measure both the Process Firewall with no rules (PF Base) and a rule base consisting of a set of 1218 rules (PF Full) generated by setting a lower threshold (100) for rule suggestion. Some benchmarks perform a large number of system calls (Apache Build and webserver benchmarks), while Bootup exercises a variety of rules in different ways. Each macrobenchmark shows between 2 and 4% overhead.

Table 6.6 shows overhead of the Process Firewall per system call. The microbenchmarks show no more than 0.51% overhead with just the default allow rules enabled (BASE) and less than 11% for any particular system call with our full rule set and optimizations enabled.

### 6.6.3 Rule Generation

This section examines the challenge of producing rules for the Process Firewall. The goal is generate rules automatically that block attacks without introducing false positives. Since several systems
Table 6.6: Microbenchmarks using lmbench. All results are in μs with percentage overhead in brackets. Standard deviation for all results was less than 1% for all our measurements. 95% confidence intervals for the non-fork results was at maximum 0.003, whereas for the fork-related results, the maximum was 0.3. Each column except the last incorporates optimizations of the previous column. DISABLED is the Process Firewall totally disabled, BASE with only the default allow rule, FULL our full rule base without any optimizations, CONCACHE with context caching optimization, LAZYCON with lazy context evaluation, and EPTSPC with entrypoint-specific rule chains.

Table 6.7: Benchmark overhead means are collected over 30 runs. Web1-1000 indicates ApacheBench latency on a LAMP system serving random database entries with 1 and 1000 concurrent clients respectively (L is latency and T is throughput). PF Base has default allow rules, and PF Full uses our full rule set.

(e.g., systrace [83], AppArmor [73], SELinux [140]) have used runtime analysis to automatically produce rules, we explore its effectiveness for the Process Firewall. In addition, we examine the ability and efficacy of OS distributors to produce effective rulesets automatically.

6.6.3.1 Rule Generation Techniques

We explore producing rules from known vulnerabilities, from runtime traces of program test suites, and from runtime traces of the system deployment. We find that: (1) known vulnerabilities can be blocked without incurring false positives; (2) for rules generated using three program test suites, we observed no false positives but these rules create unnecessary false negatives; and (3) while generating rules from runtime traces of program deployments reduces false negatives and generates widely-applicable rules, we observed that some false positives result. We examine causes for these false positives and future work to address them.

First, we generate rules for each of the over 20 previously-unknown vulnerabilities we found using our vulnerability testing tool [141]. Our testing tool logs the process entrypoint and the unsafe resource that led to the attack. We ran our testing tool to generate log entries for two verified exploits (E6, E7 in Table 6.4). We used the log entries to generate Process Firewall rules (R3, R4 for E7 in Table 6.5) to block the exploits and verified that they worked. The advantage of using known vulnerabilities to generate Process Firewall rules is that the combination of unsafe
resource and entrypoint is known to require defense to protect the program, so no false positives are possible. We generalize the rules to deny access to all unsafe resources (see Table 6.2) for the program entrypoint based on the type of vulnerability, using the SELinux MAC policies of the program itself and system services (e.g., the untrusted search path rule R7 in Table 6.5 is generalized to block all adversary-accessible resources). These policies are essentially fixed, so as long as we have a conservative view of what is unsafe these rules do not cause false positives.

To generate rules to block unknown vulnerabilities, we explore rule generation from runtime traces using program test suites. Many programs include test suites written by developers to exercise their programs’ functionality in a variety of ways. For example, PHP has a set of almost 8000 tests that exercise various configurations. We generated Process Firewall rules from runtime traces of the Apache, PHP, and MySQL test suites and did not observe false positives when enforcing these rule sets. However, test suites exercise programs under multiple program environments – configurations, command line arguments, and environment variables [142]. These environments may access resources that are not relevant to the expected deployment, thus resulting in rules that cause false negatives. For example, the Apache test suite exercises programs under configurations that allow and disallow low-integrity user-defined configuration files (.htaccess). If the expected deployment disallows .htaccess, this rule may miss attacks where Apache is somehow tricked into using these low-integrity files. Test suites have other drawbacks – their quality is variable, and they are not available for all programs.

To reduce the number of false negatives, we examine rule generation using runtime traces from deployed program environments. However, using runtime traces may cause false positives as a trace may not exercise all valid resource accesses by an entrypoint. We analyzed how entrypoints accessed resources over a two-week long runtime trace on an Ubuntu 10.04 system with SELinux that had 5234 total entrypoints and 410,000 log entries. From Section 6.4.1, invariant rules to prevent several resource access attacks can be generated for those entrypoints that access either

<table>
<thead>
<tr>
<th>Invocation Threshold</th>
<th>High Only</th>
<th>Low Only</th>
<th>Both High and Low</th>
<th>Rules Produced</th>
<th>False Positives</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4570</td>
<td>664</td>
<td>0</td>
<td>5234</td>
<td>525</td>
</tr>
<tr>
<td>5</td>
<td>4436</td>
<td>508</td>
<td>290</td>
<td>2329</td>
<td>235</td>
</tr>
<tr>
<td>10</td>
<td>4384</td>
<td>482</td>
<td>368</td>
<td>1536</td>
<td>157</td>
</tr>
<tr>
<td>50</td>
<td>4257</td>
<td>480</td>
<td>497</td>
<td>490</td>
<td>28</td>
</tr>
<tr>
<td>100</td>
<td>4247</td>
<td>480</td>
<td>507</td>
<td>295</td>
<td>18</td>
</tr>
<tr>
<td>500</td>
<td>4233</td>
<td>480</td>
<td>521</td>
<td>64</td>
<td>4</td>
</tr>
<tr>
<td>1000</td>
<td>4230</td>
<td>480</td>
<td>524</td>
<td>34</td>
<td>1</td>
</tr>
<tr>
<td>1149</td>
<td>4229</td>
<td>480</td>
<td>525</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>5000</td>
<td>4229</td>
<td>480</td>
<td>525</td>
<td>11</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6.8: Classification of entrypoints against the number entrypoint invocations (one invocation is one system call).
only high-integrity (adversary-accessible) or low-integrity (adversary-inaccessible) resources, but not both. To generate such rules, we collected all resources accessed by each entrypoint in the runtime trace. Depending on whether these resources were only high-integrity, only low-integrity, or both, the entrypoint was classified as high, low, or both. From this classification, rules were generated for entrypoints that were: (1) classified as either high or low, and (2) invoked more than a threshold number of times. For this study, we defined any resource modifiable by processes running under the untrusted SELinux user label \texttt{user\_t} as low-integrity.

We now examine whether the runtime trace identifies a threshold beyond which no false positives are observed. False positives are caused when entrypoints are classified as either high or low, but in reality access both. Table 6.8 shows how the classification of entrypoints evolved with the number of invocations of that entrypoint in the trace. The highest number of invocations at which an entrypoint changed class from high or low to both was 1149. Thus, if we used 1149 as the threshold for producing rules, then we would not see any false positives for this particular runtime trace. While only 30 entrypoints are invoked 1149 times (or more) and are classified either high or low in this trace, the corresponding rules apply in many cases (e.g., dynamic linking, PHP File Inclusion, etc.). Rules (R1-R4) in Table 6.5 were all generated based on entrypoints that were invoked more than 1149 times.

If rules are generated using a lower threshold, many more entrypoints could be protected. To find causes for false positives at lower thresholds, we manually examined the 28 entrypoints that changed classification after more than 50 invocations. First, 18 entrypoints were in libraries. These occur because libraries are called by a variety of programs in different environments, which may use the libraries for different purposes. Thus, these rules must be predicated on the environment in which the library is used. The remaining 10 were program entrypoints. Although these programs were launched under the same environment every time, they used inputs at runtime to produce names to access resources. For example, an entrypoint in \texttt{nautilus}, a graphical file browser, lists the files in a directory the user specifies in the location bar. In our particular runtime trace, the user only accessed high-integrity files in the first 50 invocations, and later accessed a low-integrity directory. Thus, to guarantee generation of rules without false positives, we need to understand how a program produces names used in resource access system calls. We leave this for future work.

### 6.6.3.2 Rule Generation by OS Distributors

We envision that OS distributors will generate Process Firewall rules and ship them to users in application packages. In this section, we discuss how to automate rule generation and whether the techniques above are useful for effective rule generation by OS distributors.

We provide scripts to automatically generate rules from Process Firewall logs (generated by
the LOG target module). Table 6.5 shows two rule templates (T1, T2) we use for rule generation from known vulnerabilities. Template T1 constrains an entrypoint to access only a set of resources identified by their (SELinux) labels. T2 creates rules to block TOCTTOU attacks. Scripts fill in rule template fields using corresponding logged values. To generate rules from known vulnerabilities using these rule templates, we need the specific Process Firewall log entries for the vulnerable system call and the type of the vulnerability. The type of vulnerability determines the template, which we fill with logged data.

An important question is whether rules generated by OS distributors are valid in deployed environments. Our insight is that Process Firewall rules generated by OS distributors are valid if programs are run in the same environment that the OS distributors generate rules for. To find such programs, we compared the launch inputs and application package files across all program invocations. If every launch used the same command line arguments and environment variables and the package files were unmodified from installation, we concluded that the deployed environment was consistent with the OS distribution’s package. Of the 318 programs and scripts that were launched in our runtime trace, we found that 232 were launched in the same environment as the installed package each time. Based on our analysis above, OS distributors could produce Process Firewall rules without false positives for a majority of programs, where they would only need to resolve 10 false positives (e.g., by testing the programs more thoroughly). Alternatively, OS distributors could produce less comprehensive rule sets using Linux test suites or only block known vulnerabilities with little risk of false positives.

6.7 Related Work

While no previous work has looked at the class of resource access attacks in a unified manner, there exist previous system and program defenses.

System-level defenses. System-level defenses have been proposed for confinement and protection. Confinement using sandboxing cannot trust process context to make authorization decisions, as malicious processes could spoof such information. Some systems assume “partially trusted” processes and use the process call stack. However, they do not make use of system information such as adversary accessibility in addition; this precludes them from defending against resource access attacks. System-level defenses have also been proposed for protection against TOCTTOU attacks and link following attacks. However, system-level protection is fundamentally limited because it does not consider program context.

---

5 We could also have used the DAC label (owner and group) to identify resources, but we chose SELinux as its labels are finer-grained.

6 User-defined configuration files are also a sign that the environment may differ across runs.
Program Defenses. Certain program API modifications have been proposed to help programs convey constraints on resource access to the OS depending on process context. Decentralized Information Flow Control [13, 41] (DIFC) enables programmers to limit the system resources available in different process execution contexts through information flow policies. Capability systems [122, 15] circumvent resource access through namespaces altogether by using direct capabilities to resources. However, such defenses require programs to be customized and rewritten to system deployments. The Process Firewall performs this customization without requiring program modifications.

Programs restrict resource access attacks such as untrusted search paths, directory traversal and PHP file inclusion by filtering adversarial names (e.g., removing ../). However, both locating all sources of adversarial input [130] and proper input filtering of names [145] are hard problems, in addition to being deployment-specific. The Process Firewall solves these problems by using resource information to block access, instead of trying to restrict names.

6.8 Conclusion

This paper introduced the Process Firewall, a system-wide kernel protection mechanism that protects processes from a variety of resource access attacks. Noting the complexity, inefficiency and incompleteness of current program defenses, our insight is to use both system knowledge of resources and adversary access, and program context to protect processes from resource access attacks. To this end, we implemented a Process Firewall prototype for the Linux kernel, utilizing a variety of optimizations that resulted in overheads of less than 4% system-wide for a variety of macrobenchmarks. In addition, we found that it is more efficient to deploy resource access defenses in the Process Firewall than in programs. Finally, we show that the Process Firewall is easy to use, as no program changes or user configuration are required and rule bases can be created from existing vulnerabilities and runtime analysis to avoid false positives. These results show that it is practical for the operating system to protect processes by preventing a variety of resource access attacks system-wide.
Chapter 7

Protecting Resource Access by Inferring Programmer Expectations

7.1 Introduction

In this chapter of the thesis, we present an automated technique to generate policy to protect a program from resource access attacks. We make the following observations: first, we find that a fundamental cause for resource access attacks is a mismatch between the program’s expectation and its deployment in a system. For example, during a particular resource access, a programmer might expect to fetch a resource that is inaccessible to an adversary (e.g., a log file in `/var/log`) and thus not add defensive code checks, called filters, to protect against adversarial control of names and bindings. However, this expectation may not be consistent with the view of the OS distributors who actually frame the access control policy. Thus, if permissions to `/var/log` allow adversary access (e.g., through a world-writable directory), adversaries can compromise the victim program. Our second insight is that we can automatically infer if the programmer expected adversarial control at a particular resource access or not, without requiring any annotations or changes to the program. We do this by detecting the presence of name and binding filters in the program.

Concretely, we develop JIGSAW, the first system to provide automated and complete protection for current programs from resource access attacks without requiring additional programmer effort. JIGSAW infers programmer expectations and enforces these on the program’s deployment\(^1\). First, we precisely define and classify resource access attacks and propose two conceptually simple invariants based on program expectation that, if correctly evaluated and enforced, completely protect programs from resource access attacks. Second, we propose and implement runtime analysis techniques to automatically detect name and binding filters in program code. Using these filters,

\(^1\)Informally, JIGSAW enables “fitting” the program’s expectations on to its deployment.
JIGSAW constructs a novel representation, called the *name flow graph*, from which program-wide resource access expectations are derived. We show that anomalous cases in the name flow graph can be used to detect vulnerabilities to resource access attacks. Finally, JIGSAW enforces these invariants by leveraging the Process Firewall presented in Chapter 6, a Linux kernel module that (i) collects information about the system’s adversary accessibility and (ii) can introspect into the program to identify and enforce the expectations of each program resource access.

We evaluate our technique by hardening widely-used programs against resource access attacks. Our results show that in general, programmers have many implicit expectations during resource access. For example, in the Apache web server, we found 65% of all resource accesses are implicitly expected to not be under adversary control. However, this is not conveyed to OS distributors in any form, and may result in vulnerabilities. This is evidenced by the fact that we found two previously-unknown resource access vulnerabilities and a default misconfiguration in the mature Apache web server. By automatically generating rules to enforce programmer expectations, we show that we can block such vulnerabilities in any system deployment where expectations are violated. We also find that the Process Firewall can enforce these rules to block resource access attacks whilst allowing legitimate functionality for a modest performance overhead of <6%. An automated analysis as presented in this chapter can thus enforce efficient protection against resource attacks at runtime.

In summary, we make the following contributions in this chapter:

- We precisely define resource access attacks and show how they occur due to a mismatch in expectations between the programmer, the OS distributor, and the administrator,
- We develop JIGSAW, an automated approach to *completely* protect programs from resource access attacks by inferring programmer expectation using the novel abstraction of a name flow graph, and
- We evaluate our approach on widely-used programs, showing how programmers have a lot of implicit expectations, as demonstrated by our discovery of two previously-unknown vulnerabilities and a default misconfiguration in our deployment of the Apache web server. Further, we show that we can produce rules to enforce these implicit assumptions efficiently using the Process Firewall on any program deployment.

### 7.2 Problem definition

#### 7.2.1 Resource Access Attacks

A resource access occurs when a program uses a *name* to resolve a *resource* using namespace *bindings*. That is, the inputs to the resource access are the name and the bindings, and the output
Figure 7.1: Motivating example of resource access attacks using a typical processing cycle of a web server.

is the final resource. Figure 7.1 shows example webserver code that we use throughout the chapter: the webserver starts up and accesses its configuration file (line 2), from which it gets the location of its log file. It then binds a socket on port 80 (line 3), opens the log file (line 4), and waits for client requests. When a client connects, it receives the HTTP request (line 6), uses this name to fetch the HTML file (line 9). Finally, it writes the status code to its log file (line 11).

Let us examine some possible resource access attacks. Consider line 6. Here, the program receives a HTTP request from the client, and serves the page to the client. The client can supply a name such as `../../etc/passwd`, and if the server does not properly sanitize the name (which it attempts in line 7), the client is served the password file on the server. This is a directory traversal attack. Next, consider the check the server makes in line 8. Here, the server checks that the HTML file is not a symbolic link. The reason for this is that in many deployments (e.g., a university web server serving student web pages), the web page is controlled by an adversary (i.e., student). The server attempts to prevent a symbolic link attack, where a student links her web page to the password file. However, a race condition between the check in line 8 and the use in line 9, leads to a link following attack exploiting a TOCTTOU race condition.

To see how such attacks can be broadly classified, we introduce adversary accessibility to resources and adversarial control of resource access. We then define resource access attacks and derive a classification.

**Adversary accessible resources.** An adversary-accessible resource is one that an adversary has permissions to (read for secrecy attacks, write for integrity attacks) under the system’s access control policy. The complement set is the set of adversary-inaccessible resources.

**Adversary control of resource access.** An adversary controls the resource access by controlling its inputs (the name or a binding). An adversary controls the name if a resource used to fetch any part of the name is adversary-accessible. The adversary needs write permissions to these resources to control names. An adversary controls a binding if she uses her write permissions in a directory to create a binding [141].

The directory traversal attack above relies on the adversary’s ability to control the name used in resource access. The link following attack relies on the adversary’s ability to control a binding
Table 7.1: Adversaries control resource access to direct victims to adversary accessible resources when the victim expected an adversary inaccessible resource and vice-versa. Also shown are the common weakness enumeration [1] classes.

<table>
<thead>
<tr>
<th>Expected/Safe Resource</th>
<th>Malicious/Unsafe Resource</th>
<th>Attack Class</th>
<th>CWE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adversary Inaccessible (Hi) Resource</td>
<td>Adversary Accessible (Lo) Resource</td>
<td>Unexpected Attack Surface</td>
<td>CWE-426</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Untrusted Search Path</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>File/IPC Squat</td>
<td>CWE-283</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PHP File Inclusion</td>
<td>CWE-98</td>
</tr>
<tr>
<td>Adversary Accessible (Lo) Resource</td>
<td>Adversary Inaccessible (Hi) Resource</td>
<td>Confused Deputy</td>
<td>CWE-59</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Link Following</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Directory Traversal</td>
<td>CWE-22</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TOCTTOU races</td>
<td>CWE-362</td>
</tr>
</tbody>
</table>

(creating a symbolic link using write permissions in a directory).

**Resource Access Attack.** A resource access attack occurs when an adversary uses her control of inputs to resource access (the name or a binding) to direct a victim to an adversary-accessible output resource when the victim expected an adversary-inaccessible resource (and vice versa).

On the one hand, the adversary can use control to direct the victim to an adversary-inaccessible resource when the program expects an adversary-accessible resource. The directory traversal and link following attacks are examples for this classical confused deputy attack [146] (Row 2 in Table 7.1). On the other hand, the adversary can direct the victim to an adversary-accessible resource when the program expects an adversary-inaccessible resource. Untrusted search paths, where a program searches for critical resources such as libraries in insecure locations are an example of this form of attack. We call these attacks unexpected attack surface attacks (Row 1 in Table 7.1), because the programmer is not expecting adversary control at these resource accesses. Table 7.1 classifies all resource access attacks into these two types. The classification is based on the definition of these attacks given in the Common Weakness Enumeration (CWE) database [1].

### 7.2.2 Insufficiency of Traditional Defenses

Traditional defenses are both challenging to implement correctly and also insufficient to defend against resource access attacks.

A first method to defend against resource access attacks uses program code filters that accept only the expected resources. However, there are a number of challenges to writing correct code checks. First, such checks are inefficient and cause performance overhead. For example, the Apache web server documentation [4] recommends switching off resource access checks during web page file retrieval to improve performance. Second, checks are complicated. The system-call API that programs use for resource access is not atomic, leading to TOCTTOU races. There is no known race-free method to perform an access-open check in the current system call API [12].
Chari et al. [8] show that to defend link following attacks, programmers must perform at least four additional system calls per path component for each resource access. Going back to the example in Figure 7.1, the checks on lines 7 and 8 are not enough – the correct sequence to use is `lstat-open-fstat-lstat` [8]. Thirdly, program checks are *incomplete*, because adversary accessibility to resources is not sufficiently exposed to programs by the system-call API. Currently, programs can query adversary accessibility only for UNIX discretionary access control (DAC) policies (e.g., using the `access` system call), but many UNIX systems now also enforce mandatory access control (MAC) policies (e.g., SELinux [29] and AppArmor [73]) that allow different adversary accessibility.

The second method of defense changes the access control policy to allow a process access to only the set of expected resources. Unfortunately, such a complicated defense does not entirely stop resource access attacks. First, fixing access control policies is a complicated task. For example, even the minimal (targeted) SELinux MAC policy on the widely-used Fedora Linux distribution has 95,600 rules. Understanding and changing such rules requires domain specific expertise that not all administrators have. Second, access control alone is anyway insufficient to stop resource access attacks because it treats processes as a black-box and does not differentiate between different resource access system calls. In our example in Figure 7.1, the web server opens both a log file and user HTML pages. Thus, it needs permissions to both resources. However, it should not access the log file when it is serving a user HTML page, and vice versa. Traditional access control does not make this difference.

### 7.2.3 Causes for Resource Access Attacks

We identify two causes for resource access attacks. The first cause is a mismatch in expectations of adversary control of names and bindings between the program and the deployment. Consider Figure 7.2 that describes resource accesses from the web server example in Figure 7.1. Here, the programmer expects the resource access of the HTML file to be under adversary control, and to combat this, adds a name filter ² from the TCP socket (stripping `../`) as well as a binding filter (check for a link). The programmer did not expect the log file’s resource access to be adversary-controlled, and therefore did not add any filters. However, due to a misconfiguration, this programmer expectation is not satisfied in the deployment configuration, causing a resource access attack.

In general, resource access attacks are a very challenging problem to solve because they involve multiple disconnected parties. First, programmers write code assuming that a certain subset of resource accesses are under adversarial control. Resource access checks cause overhead, so the programmer generally tries to minimize the number of checks, thereby motivating fewer filters.

---

²A filter is a check in code that allows only a subset of names, bindings and resources through.
Second, there are OS distributors who define access control policies, thereby determining adversarial control of resource accesses. However, these OS distributors have little or no information about the assumptions the programmer has made about adversarial control, resulting in a set of possible mismatches. Finally, there are administrators who deploy the program on a concrete system. The configuration specifies the location of various resources such as log files. Thus, the administrator’s configuration too may not match the programmer’s expectation.

The second cause for resource access attacks is where the programmer does expect adversary-controlled resource access, but the filter may be insufficient to protect the program. Note that when a program encounters an adversary-controlled resource access, the only valid resource is an adversary-accessible resource; otherwise, the program is victim to a confused deputy attack. Thus, the program needs to defend itself by filtering against confused deputy attacks. However, both name and binding filters are difficult to get right due to difficulty in string parsing \cite{145} and inherent race conditions in the system call API \cite{12} (e.g., lines 8, 9 in Figure 7.1).

In summary, the two causes of resource access attacks are: (a) unexpected adversarial control of resource access, and (b) improper filtering of resource access when adversary control of resource access is expected. These causes correspond to attacks in Rows 1 and 2 in Table 7.1 respectively. With these two causes in mind, we proceed to a model that precisely describes our solution.

### 7.3 Model and Solution Overview

Consider the set of all resource accesses $RA$ made by a program. A resource access happens when a system call resolves a resource using a name and namespace bindings. The program has code to filter the names and bindings used during certain resource accesses (e.g., $ra_3$ in Figure 7.2). From this knowledge, we show in Section 7.5 how to derive $P$, the set of resource accesses that a program expects to be under adversarial control. This set $P$ is the expected resource access attack surface,
or simply, the expected attack surface.

Now, assume that the program is deployed and run on a system. A subset of the resource accesses made by the program is adversary-controlled in the deployment. Let $Y$ be the deployment’s access control policy. Let $S$ be the set of resource accesses that are adversary-controlled under $Y$ ($ra_2$ in Figure 7.2). This set $S$ defines the deployment resource access attack surface, or simply, the deployment attack surface.

Given $Y$, the expected attack surface $P$ is safe for the deployment $S$ if $S \subseteq P$, i.e., if all resource accesses in the deployment attack surface are part of the program’s expected attack surface. Intuitively, this means that the program has filters to protect itself whenever a resource access is adversary-controlled. In terms of propositional logic unexpected adversary control is defined as:

**Invariant: Unexpected Adversary Control** $(r)$:

$$ (r \in S) \land (r \notin P) \implies \text{Deny} $$

(7.1)

If the safety invariant is enforced, resource access attacks are eliminated where programs do not expect adversary control. Therefore, attacks are only possible where programs expect adversary control.

Now that we are dealing with an adversary-controlled resource access ($r \in S$) that is also expected ($r \in P$), the only valid resource is an adversary-accessible resource; otherwise, the program would be victim to a confused deputy attack. We say that resource accesses in $P$ are protected from a confused deputy attack if, when the resource access is adversary-controlled (i.e., $r \in S$), it does not accept adversary-inaccessible resources. Let $R$ be the set of resources that are adversary-accessible under $Y$. Then, $P$ is protected from confused deputy attacks if:

**Invariant: Confused Deputy** $(r)$:

$$ (r \in P) \land (r \in S) \land (r \notin R) \implies \text{Deny} $$

(7.2)

Once these two rules are enforced, the only resources that are allowed are adversary-accessible resources where programs expect adversary control. Problems occur if the program does not properly handle this adversary-accessible resource. For example, if it does not filter data read from this resource properly, there may be a buffer overflow. Such memory corruption attacks are not within the scope of this work.

Let us examine how the rules above stop the attack classes in Table 7.1. Consider attacks in Row 2, where the victim expects an adversary inaccessible resource (high integrity or secrecy), but ends up with an adversary accessible (low integrity or secrecy) resource. The typical case is an untrusted search path where the program expects to load a high-integrity library, but searches
for libraries in insecure paths due to programmer oversight or insecure environment variables, and ends up with a Trojan horse low-integrity library. Here, since the programmer does not expect a low-integrity library, she does not place a binding (or name) filter. Thus, we will infer that this resource access is not part of the expected attack surface \( (\not\in P) \), and Invariant 7.1 above will stop the attack if this resource access is controlled in any way (binding or name) by an adversary \( (\in S) \). The other attacks classes in this category are blocked similarly. Next, consider attacks in Row 1. Here, the victim expects an adversary accessible resource (low integrity or secrecy), but ends up with an adversary inaccessible resource (high integrity or secrecy). In a link following attack, the adversary creates a symbolic link to a high-secrecy or high-integrity file, such as the password file. Thus, the adversary uses her control of bindings \( (\in S) \) to direct the victim to an adversary-inaccessible resource \( (\not\in R) \). In a directory traversal attack, the adversary uses her control of the name to supply sequences of 
\(. . .\) to direct the victim to a high-secrecy or high-integrity file. In both cases, Invariant 7.2 will block the attack since the adversary controls the resource access \( (\in S) \) through the name or binding, but the resource is adversary inaccessible \( (\not\in R) \).

### 7.4 JIGSAW Approach Overview

Figure 7.3 shows an outline of the design of JIGSAW. To enforce Invariants 7.1 and 7.2 above, we need to calculate \( P \), the set of resource accesses for which the programmer expects adversary control, \( S \), the set of resource accesses under adversary control in the deployment, and \( R \), the set of adversary-accessible resources. First, finding \( P \) requires inferring programmer expectations. To infer programmer expectations, we propose an intuitive heuristic – if the programmer expects adversary control at a resource access, she will place filters in code to handle such control. Given the program code, we detect the existence of any binding and name filtering separately (Step 1 in Figure 7.3), and use this information to calculate the program’s expected attack surface (Step 2). Next, we need to calculate \( S \) and \( R \). The deployment’s access control policy \( Y \) determines...
which resources and bindings are adversary accessible. We leverage existing techniques to calculate adversary accessibility given \( Y \) \[138, 109, 8\] (Step 3). At runtime, if an adversary accessible resource is used, that resource access is in \( R \). If the name is read from an adversary-accessible resource or the binding used in resolving that name is adversary accessible, then that resource access is in \( S \). Finally, we need to enforce Invariants \( 7.1 \) and \( 7.2 \) for individual resource accesses (Step 4). Any enforcement mechanism that applies distinct access control rules per individual resource access system call would be suitable. In our prototype implementation we leverage the open-source Process Firewall \[147\] which enables us to support binary-only programs (i.e., our prototype implementation does not rely on source code access).

### 7.5 Finding Expected Attack Surfaces

The first step is to determine the expected attack surface \( P \) for a program. We do this in two parts. First, we propose a heuristic that implies the expectations of programmers with respect to the adversary control of the inputs to resource access and introduce the abstraction of a name flow graph to model these expectations and enable the detection of missing filters (Sections 7.5.1 to 7.5.3). Next, we outline how one can use dynamic analysis methods to build name flow graphs by accounting for adversary control of names and bindings (Sections 7.5.4 and 7.5.5).

#### 7.5.1 Resource Access Expectations

Determining \( P \) requires knowledge of the programmers’ expectations – whether the programmers expected the resource access to be under adversary control or not. The most precise solution to this problem is to ask each programmer to specify her expectation. Unfortunately, such annotations do not exist currently. As an alternative, we use the presence of code filters to infer programmer expectation. We use the following heuristic:

**Heuristic.** If a programmer expects adversarial control of a resource access, she will add code filters to protect the program from adversarial control.

Thus, the way we infer if a programmer expects an adversary-controlled resource access is by detecting if she adds any code to filter such adversarial control. An adversary controls a resource access by controlling either the name or a binding used in the resource access. Thus, we need to detect whether a program filters names and bindings separately.

Before presenting exactly how we detect filtering, we will introduce the concept of a name flow graph for a program, which we will use to derive the expected attack surface \( P \) given knowledge of filtered resource accesses.
7.5.2 Name Flow Graph

We introduce the abstraction of a name flow graph, which represents the data flow of name values among resource accesses in the program annotated with the knowledge of whether names and/or bindings are filtered each individual resource access. Using this name flow graph, we will show that we can compute resources accesses that are missing filters automatically. A name flow graph $G_n = (V, E)$ is a graph where the resource accesses are nodes and each edge $(a, b) \in E$ represents whether there exists a data flow in the program between the data of any of the resources retrieved at the access at node $a$ and any of the name variables used in an access at node $b$. We refer to these edges as name flows.

Further, $V = V_f \cup V_u$ where $V_f$ is the set of resource accesses that filter bindings and $V_u$ the set of vertices that do not. Similarly, $E = E_f \cup E_u$, where $E_f$ is the set of name flows that are filtered, and $E_u$ the set that is not. That is, a name flow graph is a data-flow graph that captures the flow of names and is annotated with information about filters. The meaning of filtering for names and bindings is described in Sections 7.5.4 and 7.5.5, respectively.

The name flow graph for our web server in Figure 7.1 is shown in Figure 7.4. Its nodes are resource accesses and edges connect two resource accesses if the data read at the source resource access may affect the name used at the target resource access. The bold nodes are those that filter bindings, whereas the bold edges are those that filter names.

The name flow graph determines $P$, the expected attack surface. According to our heuristic in Section 7.5, a resource access is part of the expected attack surface if a programmer places both name and binding filters on the resource access to handle adversarial control. However, not all name flows need be filtered – only name flows originating from other resource accesses also under adversarial control must be. Since this definition is transitive, we need to start with some initial information about resource accesses that are part of the expected attack surface, which we do not have. However, we find that we can easily define which resource accesses should not be in $P$. That is, we can use the absence of filters to determine resource accesses that should not be under
adversarial control. This complement set of $P$ is $\overline{P}$. We define an approach to calculate $\overline{P}$ below. Any resource access not in $\overline{P}$ is then in $P$, the expected attack surface.

Formally, a resource access $u \in \overline{P}$ if any of the following conditions are satisfied:

(i) $u \in V_u$: Binding filters do not exist, or
(ii) $u \not\xrightarrow{e} v \in V_u$: There exists an unfiltered name flow edge originating at $u$, or
(iii) $(u \xrightarrow{e} v) \land (v \in \overline{P})$: There exists a name flow path originating at $u$ to a resource access in $\overline{P}$.

Consider the example in Figure 7.5. Here, resource accesses $a$ and $b$ filter bindings ($a, b \in V_f$). $c$ does not filter bindings ($c \in V_u$). $c$’s name is determined from input at $b$, and $b$’s name is determined from input at $a$. The name flow from $a$ to $b$ is filtered. By (i) above, $c \in \overline{P}$ since it does not filter bindings, and the programmer did not expect adversary control by our heuristic. Next, by (ii) above, $b \in \overline{P}$, since it is the origin of an unfiltered name flow (which adversaries should not control). Finally, by transitivity using (iii) above, $a \in \overline{P}$, because it is the origin of a name flow to a resource access that is in $\overline{P}$, and thus adversaries should not control the name read from resource access at $a$. All combinations of name and binding filters between a pair of nodes and the inference of node membership in $P$ are presented in Figure 7.6.

Figure 7.7 describes the algorithm used to calculate membership in $P$, given $V_f, V_u, E_f$, and $E_u$. It implements (i)-(iii) above. It starts by initially assigning any node that does not filter bindings to $\overline{P}$ (ii)), and the source of unfiltered name flows to $\overline{P}$ (ii)). It then uses a fixed point iteration to apply the transitive rule (iii), and adds the source of any name flow to a target already in $\overline{P}$ to $\overline{P}$. At the termination of the algorithm, any resource access not in $\overline{P}$ is in $P$.

7.5.3 Detecting Missing Filters

Using the name flow graph, we can compute cases where filtering is likely missing. Intuitively, a filter is missing if the program filters some adversarial control of resource access but not others. This can happen in two cases: (a) if an incoming name flow is filtered but the binding at the resource access is not, or (b) a binding is filtered but an outgoing name flow is not. The dotted boxes in Figure 7.6 show these cases.

Precisely, filters are possibly missing at a resource access $r$ in two cases:

Case 1: \[\exists s, e : (s \not\xrightarrow{e} r \land e \in E_f \land r \in V_u).\] There exists a filter on an incoming name flow (indicating adversarial control of name) but not a binding filter, or
Figure 7.6: Determining whether a resource access in a resource flow graph should be in $P$.

Input: Set of unfiltered names $E_u$ and bindings $V_u$
Output: $P$

1: $P \leftarrow \emptyset$
2: for $v \in V_u$ do
3: \[ P \leftarrow P \cup v \]
4: end for
5: for $e \in E_u$ do
6: \[ P \leftarrow P \cup e.\text{src} \]
7: end for
8: $c \leftarrow True$
9: while $c = True$ do
10: \[ c \leftarrow False \]
11: for $e \in E_u$ do
12: \[ \text{if } e.tgt \in P \land e.\text{src} \not\in P \text{ then} \]
13: \[ P \leftarrow P \cup e.\text{src} \]
14: \[ c \leftarrow True \]
15: end if
16: end for
17: end while

Figure 7.7: Inferring $P$ from knowledge of filtering

Case 2: $\exists s, e : (r \Rightarrow s \land e \in E_u \land r \in V_f)$. There exists a filter on a binding (indicating an adversary-accessible resource) but not on all outgoing name flows.

As an example of a missing filter indicating a vulnerability, we found that in the default configuration, the Apache web server filters the name supplied by a client (by stripping ../), but does not filter the binding used to fetch the HTML file. Therefore, an adversary can create a link of her web page to /etc/passwd, which will be served.

Not all possibly missing filters indicate a vulnerability. Some filters perform the same checks as JIGSAW. As an example, we found that libc had binding filters when it accessed (some) resources under /etc to reject adversary-accessible resources, enforcing Invariant 7.1 itself. Thus, there is no need to filter names originating from this resource (although Case 2 indicates a possibly missing filter). We call filters that perform the same checks JIGSAW, redundant.
7.5.4 Detecting Presence of Binding Filters

We now outline our technique for detecting the filtering of bindings. Our objective in detecting here is to determine resource accesses that perform any filtering of bindings. Note that we do not aim to prove the correctness of the filtering checks themselves.

To define how we detect binding filters, we discuss how bindings are involved in resource access and how programs filter them. A program accesses many bindings (directories and symbolic links) during a single resource access. In theory, any one of these is controllable by the adversary. Filtering of directories is done by checking its security label, whereas link filtering checks if the binding is a link, and optionally, the security label of the link’s target. Bindings are filtered if, in some cases, the program does not accept a binding based on checks done on any binding used during resource access. An ideal solution would detect the existence of any such check.

Both static and dynamic approaches are possible to detect binding filtering. Static analysis uses the program code to determine if checks exist. However, this is quite challenging as there are a wide variety of ways to perform checks, including, for example, lowering the privilege of the process to that of the adversary [141, 90]. Instead, we opt for a dynamic analysis that detects the effects of filtering.

To detect filters, we have to choose a test that will definitely fire the filter, if such a filter is present. Not all attacks in Table 7.1 are suitable to detect program filters. Consider the subset of attacks in Table 7.1 where the adversary uses her control of bindings to direct the victim to an adversary-accessible resource (Row 1). If the program accepts the adversary-accessible resource, it is generally not possible to determine if this was due to the program intentionally accepting this resource or due to the program assuming that there would be no adversary control of the resource access. On the other hand, consider the subset of attacks where the adversary uses control of bindings to direct the victim to an adversary-inaccessible resource (e.g., link following). Here, if the programmer were expecting adversary-controlled bindings, she has to add checks to block this resource access as this scenario is, by definition, a confused deputy attack. Thus, we can use the results of a link following attack to determine the existence of binding filters, and thus, the programmer’s expectation. In Section 7.8, we describe a dynamic analysis framework that performs these tests.

7.5.5 Detecting Presence of Name Filtering

The other way for adversaries to control resource access is to control names. We aim to determine if the program makes any attempt to filter names, which would indicate that the programmer expected to receive an adversary-controlled name. Again, note that to determine programmer
expectation, we only need to determine if there is any filtering at all, not if the filtering is correct.

To determine name filters in programs, we first describe how names originate. Names are either hard-coded in the program or received at runtime. First, hard-coded names are constants defined in the program binary or a dynamic library. For an adversary to have control of hard-coded names, she needs to control the binary or library, in which case trivial code attacks are possible. Therefore, we assume hard-coded names to not be under adversarial control. Second, programs get names from runtime input. In Figure 7.1, a client is requesting a HTML file by supplying its name. The server reads the name from this request (name source) and accesses the HTML file resource from this client input (name sink). In general, a name can be computed from input using one or more read system calls.

Next, we define the action of filtering names. There are two ways in which programs filter names. First, programs can directly manipulate the name. For example, web servers strip malicious characters (e.g., ..) from names. Second, it can check that the resource retrieved from this name is indeed accessible to the adversary. For example, the setuid mount program accepts a directory to mount from untrusted users who are potential adversaries, but checks that the user indeed has write permissions to the directory before mounting. Thus, a name is filtered between a source and a sink if, in some cases, the name read at a source is changed, or the resource access at the sink is blocked. An ideal solution would detect the existence of any such checks.

Determining name filtering is a two-step process. First, we should determine pairs of resource accesses where the name is read from one resource (source) and used in the other (sink). Next, we determine if the program places any filters between this source-sink pair.

Again, we can use static or dynamic analysis to find pairs and filters. To detect filters, Balzarotti et al. used string analysis [145], whereas techniques such as weakest preconditions [148] or symbolic execution [149] can also be used. However, static analysis techniques are traditionally sound, but may produce false positives. Therefore, we use dynamic analysis to detect evidence of filtering.

To determine both pairs and filtering, we use a runtime analysis inspired by Sekar [150]. Sekar’s aim is to detect injection attacks in a black-box manner. The technique is to log all reads and writes by programs, and find a correlation between reads and writes using an approximate string matching algorithm. Thus, given as input a log of the program’s read and write buffers, the algorithm returns true if a write buffer matches a read buffer “approximately”.

We adapt this technique to find name flows. We log all reads and names during resource access, and find matches between names and read buffers. We first try matching the full name; if no match is found, we try to match the directory path and final resource separately. Often, parts of a name are read from data of different resources. For example, a web server’s document root is read from the configuration file, whereas the file to be served is read from the client’s input. Both of these are
combined to form the name of the resource served. As with the method for finding binding filters, we use the directory traversal attack in Row 2 to trigger filtering.

Since our analysis is a black-box approach, if a possible name flow is found, the read buffer might just coincidentally happen to have the name, but not actually flow to it. Thus, we execute a verification step. We run the test suite again, but this time change the read buffer containing the name to special characters, noting if the name also changes. If it does, we have found a name flow.

7.6 Enforcing Program Expectations

Once we find the program’s expected attack surface \( P \), JIGSAW can enforce resource access protections using Invariant 7.1 and Invariant 7.2 in Section 7.3 on program deployments. To do this, a reference monitor [72] has to mediate all resource accesses and enforce these rules. To enforce these rules correctly for each resource access, a reference monitor must determine whether this resource access is in \( P \), and identify the system deployment’s attack surface \( S \) and adversary accessibility to resources \( R \).

7.6.1 Protecting Accesses in \( P \) at Runtime

The first challenge is to determine whether the resource access is in \( P \). There are two ways to do this: (a) the program code can be modified to convey its expectation to the monitor through APIs, or (b) the monitor already knows the program expectation and identifies each resource access. Capability systems use code to convey their expectation during each resource access. Capability systems [26] present capabilities for only the expected resources to the OS during access. For example, decentralized information flow control (DIFC) systems \([13, 41]\) require the programmer to choose labels for the authorized resource for each resource access. However, such systems require modifying program code and recompilation, which can be complex to do correctly.

Another option is for the reference monitor to extract information necessary for it to identify the specific resource access, and hence whether it is in \( P \). Researchers have recently made the observation that if they only protect a process, they may introspect into the process (safely) to make protection decisions \([147]\). For example, they implemented a mechanism called the Process Firewall, which is a Linux kernel module that introspects into the process to enforce rules to block vulnerabilities per system call invocation. This is similar in concept to a network firewall that protects a host by restricting access per individual firewall rules. We use this option because it does not require program code or system access control policy changes, and was shown to be much faster than corresponding program checks in some cases.

The general invariant that the Process Firewall enforces is as follows:
pf invariant(subject, entrypoint, syscall_trace, object, resource_id, adversary_access, op) \rightarrow Y|N

Here, entrypoint is the user stack at the time of the system call. Resource accesses in $P$ are identified by their entrypoint. A single system call may access multiple bindings (e.g., directories and links) and a resource. As each binding and resource is accessed at runtime, its adversary access is used in the decision. If a binding is adversary accessible, then the resource access is in $S$. If the final resource is adversary accessible, then the resource access is in $R$. If a resource access in $R$ is the source of a name, this fact is recorded in syscall_trace and the resource access using this name is in $S$. This general invariant is instantiated to enforce our invariants in Section 7.3. The invariants are converted into Process Firewall rules using templates (Section 7.8).

7.6.2 Finding Adversary Accessibility $R$

$R$ is the set of resource accesses at runtime that use adversary accessible resources, and is required to enforce Invariant 7.2 in Section 7.3. Calculating $R$ requires knowing: (a) who an adversary is, and (b) whether the adversary has permissions to resources for resource accesses at runtime. We address these questions in turn.

There have been several heuristics to determine who an adversary is. Gokyo [138] uses the system’s mandatory access control policy to determine the set of SELinux labels that are trusted for the system – the rest are adversarial. Vijayakumar et al. [130] extend this approach to identify per-program adversaries. Chari et al. [8] and Pu et al. [21] use a model based on discretionary access control – a process running with a particular user ID (UID) has as its adversaries any other UID, except for the superuser (root). We can use any of these approaches to define an adversary.

Our system shifts the manual effort needed from changing code or access control policy to defining the adversary model. This reduces manual effort due to two reasons. First, defining an adversary model is anyway necessary even while writing code filters or changing the policy. Second, we find that there are intuitive adversary models that most programmers use while writing code, such as those described above. Such adversary models are pre-defined in our system and do not require any additional manual effort.

Second, we need to determine whether an adversary has permissions to resources. As discussed in Section 7.2, an adversary-accessible resource is one that the system’s access control policy $Y$ allows an adversary permissions to (read for secrecy attacks, write for integrity attacks, and execute for both). This can be queried directly from the access control policy $Y$. Any resource access at runtime that uses adversary accessible resources is in $R$. 
7.6.3 Finding Deployment Attack Surface $S$

The deployment attack surface $S$ is the set of resource accesses a process performs at runtime that are adversary-controlled. An adversary can control resource accesses by controlling either the name or a binding (or both). An adversary controls a binding if she uses her write permissions in a directory to create a binding. An adversary controls a name if the adversary has write permission to the resource the name is fetched from.

**Finding Vulnerabilities.** We can use the rules generated to also find vulnerabilities. Vulnerabilities are detected whenever a resource access is denied by our rules but is allowed by the program.

We use the same dynamic analysis from test suites that we use to detect the presence of filters in Sections 7.5.4 and 7.5.5 to also test the program for vulnerabilities in our particular deployment. Instead of enforcing the rules, we compare denials by Invariant 7.1 or Invariant 7.2 in Section 7.3 with whether the program allows the resource access. If the rule denies resource access whereas the program accepts the resource, we flag a vulnerability. Note that this process locates vulnerabilities in our specific deployment; there might be other vulnerabilities in other deployments that we miss. In any case, our rules, if enforced, will protect these program vulnerabilities in any deployment.

7.7 Proving Security of Resource Access

We present an oracle-based argument that JIGSAW eliminates resource access attacks. That is, we can reason about the correctness of our approach assuming the correctness of certain oracles on which it depends. We claim that Invariants 7.1 and 7.2 in Section 7.3 protect a program from all resource access attacks given the correctness of oracles that determine program expectation and adversary accessibility.

According to the definition in Section 7.2.1, a resource access attack is caused when an adversary controls an input (name or binding) to direct a program to an adversary-accessible resource when the program expects an adversary-inaccessible resource (and vice-versa). Resource access attacks are thus blocked when adversary control of input cannot direct the program to retrieve unexpected output resources. First, consider that the programmer expects only an adversary-inaccessible resource at a particular entrypoint. In this case, adversary control of an input name or binding results in accessing either an unexpected adversary-accessible resource or causes a confused deputy attack. Invariant 7.1 denies all adversary control of input in this case, thus blocking both attacks. Next, consider when the programmer may expect to retrieve adversary-accessible resources at a particular entrypoint. If indeed the adversary controls the input name or binding, the only authorized output is an adversary-accessible resource; otherwise, a confused deputy attack can result.
To block this, Invariant 7.2 allows retrieval of only adversary-accessible resources when input is under adversary control. Hence, we have shown that our rules deny resource accesses if and only if adversary control of input directs the program to unexpected resources, thus blocking resource access attacks without false positives.

Our argument is contingent on the correctness of oracles for determining program expectation and adversary accessibility. The first oracle determines programmer expectation, for which we use the intuitive heuristic in Section 7.5: if a programmer does not place filters, then she does not expect adversary control of resource access, i.e., she only expects adversary-inaccessible resources. The detection of filters themselves uses runtime analysis. This faces two issues: (i) if identified filters are really present, and (ii) incompleteness of runtime analysis. First, if a detected filter is not really present, this might result in false negatives. However, to detect filters, we mimic an attack and detect if the program blocks the attack. The only way the program could have blocked the attack is if it had a filter. Second, runtime analysis is inherently incomplete and may lead to false negatives for those resource accesses not covered. However, even with the limited developer test suites currently available, we were able to generate rules to block many attacks and find vulnerabilities, even in mature programs.

The second oracle determines adversary accessibility. While there is no universally agreed-upon definition of who an adversary is, we use the intuitive DAC model that most programmers assume [8, 21]. However, our framework permits the use of different adversary models. More conservative adversary models [130] will identify the maximal number of adversary-accessible resources, which increases the cases where an adversary has control, which may expose more cases where filtering is missing entirely.

7.8 Implementation

There are two parts to our implementation. First, we need to test individual program resource accesses to detect the presence of filters. This is used by the algorithm in Figure 7.7 to generate $P$. Second, we need to enforce invariants in Section 7.3 using the Process Firewall. This involves determining $R$ and $S$; this is done as discussed in Section 7.6.2 and Section 7.6.3 respectively.

7.8.1 Testing Programs

To test programs, we develop a framework that can intercept system calls and pass control to a user-space component that performs the tests. The kernel component is a small module that intercepts system calls and returns, and forwards them to a user-space daemon through netlink.

---

A process with uid $X$ has as its adversaries any uid $Y \neq X$ (except superuser root)
sockets. The flow of operations is as shown in Figure 7.8. When a monitored program makes a system call, it is intercepted by the framework’s kernel module, and forwarded to the user-space daemon. There are two resource namespaces available per program – a “test” namespace that is modified for hypothetical tests and the original namespace (similar in principle to [141]). This daemon introspects into the monitored process to identify its resource access (using the user-space stack), and checks its history to see if filters have already been detected. If not, it then proceeds to modify the test filesystem (for binding filter detection). It then returns to the kernel module. Control passes to the process in kernel mode, which accesses the original or test filesystem (depending on whether binding filters are being tested). The system call end is also intercepted, and similarly forwarded to the user-space daemon to test for name filters (as the read buffer is now available and can be modified).

We use test suites provided with program source code to drive the programs. We repeatedly run these suites until all resource accesses have been tested for filters.

7.8.2 Enforcing Invariants

As noted, JIGSAW uses the open-source Process Firewall [147] to perform enforcement. The Process Firewall is a kernel module that uses the Linux Security Modules to mediate resource accesses. In addition, it can perform a user stack backtrace to identify the particular resource access being made. Given $P$ and the edges in the name flow graph, we have two rule templates to instantiate invariants into rules to be enforced by the Process Firewall. Figure 7.9 shows the templates. Note that the rules for confused deputy are stateful – a name or binding under adversary control is recorded by the first rule, and accessing an adversary-inaccessible resource is blocked by the second.
Rule Templates

**Unexpected Adversary Control:** \((r \in S) \land (r \in \overline{P}) \implies \text{Deny}\)

For each \(r \in \overline{P}^:\)

```
pftables -i r.ept -d LOW -o DIR_SEARCH -j DROP
```

**Confused Deputy:** \((r \in P) \land (r \in S) \land (r \in \overline{R}) \implies \text{Deny}\)

**Name:**

For each \(r_1 \in P\) such that \(E(r_1, r_2)\):

```
pftables -i r1.ept -d LOW -j STATE --set --key <random_value> --value 1
pftables -i r2.ept -d HIGH --m STATE --key <random_value> --cmp 1 --equal -j DROP
```

**Binding:**

```
pftables -d LOW -o DIR_SEARCH -j STATE --set --key <random_value> --value 1
pftables -d HIGH --m STATE --key <random_value> --cmp 1 --equal -j DROP
```

Figure 7.9: Process Firewall rule templates.

### 7.9 Evaluation

In this section, we evaluate our technique on several widely-used programs. We chose these programs because: (i) resource accesses are central to their operation, and (ii) they offer a study in contrast – OpenSSH and Postfix were architected for security [54]. To derive expectations, we used developer test suites that came with the program or created our own. We answer: (a) how common are implicit programmer expectations during resource access, (b) whether the resulting expected attack surface was safe for our deployment and vulnerabilities where not, and (c) security effectiveness of hardened programs from resource access attacks. We find that in 4 out of 5 programs, more than 55% of all resource accesses are implicitly expected to be free from adversarial control. Moreover, we discovered two previously-unknown vulnerabilities and one default misconfiguration in the Apache webserver. Finally, we find that protection can be enforced with an overhead of <6% on a variety of programs and few false positives.

#### 7.9.1 Implicit Programmer Expectations

Table 7.2 shows a summary of the results obtained by JIGSAW. We first note the percentage of resource accesses that are implicitly expected to not be in \(P\), due to the absence of name or binding filters. For 4 out of 5 programs, the programmer placed no filters for more than 50% of accesses. If any of these resource accesses somehow come under adversarial control, the program can be compromised. It is very easy for OS distributor policies or administrator configurations to not match with these programmer assumptions. By explicitly identifying such resource accesses, we are able to protect them from any adversarial access in emphany deployment. OpenSSH makes much fewer assumptions during resource access (17.6%). However, OpenSSH was re-architected after several years of experience with previous vulnerabilities. Using our technique, we can protect
Program Dev Tests? Impl. Exp. % Impl. Missing Redundant Vuls. Inv. 7.1s Inv. 7.2s

| Program          | Dev Tests? | \(|V|\) | \(|E|\) | \(|V_j|\) | \(|E_j|\) | \(\in P\) | \(\notin P\) | Impl. Exp. % | Missing | Redundant | Vuls. | Inv. 7.1s | Inv. 7.2s |
|------------------|------------|--------|--------|--------|--------|---------|----------|-------------|---------|----------|-------|----------|----------|
| Apache v2.2.22   | Yes*       | 20     | 23     | 7      | 5      | 13      | 65%      | 2           | 0       | 0        | 3     | 13       | 12       |
| OpenSSH v5.3p1   | Yes        | 17     | 17     | 14     | 0      | 14      | 3         | 17.6%       | 0       | 3        | 0     | 3        | 2        |
| Samba3 v4.7      | Yes        | 210    | 84     | 78     | 19     | 78      | 132      | 62.8%       | 0       | 5        | 0     | 132      | 40       |
| Winbind v3.4.7   | Yes        | 50     | 38     | 19     | 13     | 19      | 31       | 63.3%       | 0       | 0        | 0     | 31       | 28       |
| Postfix v2.10.0  | No         | 181    | 15     | 79     | 7      | 79      | 102      | 56.32%      | 0       | 0        | 0     | 102      | 15       |

Table 7.2: Statistics of program-wide resource accesses. Dev Tests show whether we used developer test suites or created our own. Impl. Exp. is the percentage of resource accesses (|P|/|V|) that are implicitly expected to be adversary-inaccessible. The last two columns show the number of instantiations of Invariant 7.1 and Invariant 7.2 in Section 7.3 for resource accesses in the program. *- We augmented the Apache test suite with additional tests.

Figure 7.10: Resource Flow Graph for Apache. Nodes that have the icon of adversaries beside them are those where we found adversarial control of resource access in our deployment.

7.9.2 Case Study: Apache

In total, we found 20 resource accesses for Apache. Of these, Apache code filtered bindings for 7 accesses, and the name for 5 accesses. This led to 13 out of 20 resource accesses (65%) not being in \(P\) (using the algorithm in Figure 7.7). We found three resource accesses in \(S - P\) for the Apache web server in our deployment, violating the first rule in Section 7.3. These corresponded to two previously unknown vulnerabilities in the Apache web server and one default misconfiguration. That such problems occur in even a mature program like Apache shows the importance of principled reasoning of resource access. While we found these vulnerabilities in our deployment, other deployments may have different vulnerabilities, but all will be blocked using our enforcement (Section 7.9.3).

Figure 7.10 shows the resource flow graph for Apache. Apache’s expected attack surface is centered around resource accesses during the interaction with a client to serve a web page. It assumes that the location of the main configuration file and resources specified in it are not adversary controlled. Apache’s resource flow graph is relatively complex due to long chains of resource flows,
and it is difficult to reason about safety without the help of automated tools like ours. Resource accesses that had vulnerabilities in our deployment are shaded in the graph.

The first vulnerability we found was during resource access of a user-defined `.htpasswd` file. Apache allows each user the option of enabling HTTP authentication for parts of their website. This includes the ability to specify a password file of their choice. However, the resource access that fetches this password file is not filtered. Thus, users can specify any password file – even one that they do not have access to. One example exploit is to direct this password file to be the system-wide `/etc/passwd`. Traditionally, it is difficult to brute-force the system-wide password file since prompts are rate-limited. However, since HTTP authentication is not rate-limited, this may make such brute-force attacks realistic. Such a scenario, though obvious after discovery, is very difficult to reason about manually due to Apache’s complex resource accesses. Thus, it has remained hidden all these years.

The second vulnerability is a default misconfiguration. When serving web pages, Apache controls whether symbolic links can be followed from user web pages by the option `FollowSymLinks`, which is turned on by default in Ubuntu and Fedora packages. Turning this option on implicitly assumes trust in the user to link to only her own web pages. Interestingly, the name for this resource access is filtered – only the bindings are not. One way we were able to take advantage of this misconfiguration was through the error document on specific errors, such as HTTP 404, that is specifiable in the user-defined configuration `.htaccess` file. This allows an adversary to access any resource the Apache web server itself can read, for example, the password file and SSL private keys. We found that our department web server also had this option turned on. By simply making an error document linked to `/etc/passwd`, we were able to view the contents of the password file on the server. This demonstrates another typical cause of resource access attacks – administrators misconfiguring the program and violating safety of the expected attack surface.

The third vulnerability is a link following attack on `.htaccess`. Apache allows `.htaccess` to be any file on the filesystem it has access to. This may be exploited to leak configuration information about the webserver.

Finally, we note that test suites that come with programs are traditionally focussed towards testing functionality and not necessarily resource access. For example, the stock test suite for Apache only uncovered 7 resource accesses in total, and after we augmented it, there were 20 in total. Better test suites for resource access would help test more resource accesses.

### 7.9.3 Process Firewall Enforcement

Process Firewall rules enforce the safety of the expected attack surface under the deployment attack surface. Given the program’s expected attack surface, Process Firewall rules enforce that
any adversary-controlled resource access at runtime (i.e., part of the deployment attack surface) is allowed only if the resource access is also part of the program’s expected attack surface. In addition, for those resource accesses allowed, they also protect the program against confused-deputy link and directory traversal attacks. The last two columns in Table 7.2 shows the number of Process Firewall rules we obtained (separately due to Invariants 7.1 and 7.2).

We evaluated the ability of rules to block vulnerabilities. First, we verified the ability of these rules to block the three discovered vulnerabilities in Apache. Second, we tried previously-known, representative resource access vulnerabilities against Apache and Samba. We tested an untrusted library load (CVE-2006-1564) against Apache. Here, a bug in the package manager forced Apache to search for modules in untrusted directories. Our tool deduced that the resource access that loaded libraries did not have any filtering, and thus, was not in \( P \), blocking this attack due to Invariant 7.1 in Section 7.3. In addition, we tested a directory traversal vulnerability in Samba (CVE-2010-0926). This is a confused deputy attack involving a sequence of ..\ in a symbolic link. This attack was blocked due to Invariant 7.2.

### 7.9.4 False Positives

False positives are caused by a failure of our heuristic in Section 7.3 that determines program expectation. In some cases, we found that a program had no filters at a resource access, but still expected adversary-controlled resource access. We found that this case occurs in certain small “slave” programs that perform a requested task on a resource without any resource access filters. For example, consider that the administrator (root) runs the cat utility on a file in an adversarial user’s home directory. Because cat does not filter the input bindings, the user can always launch a link following attack by linking the file to the password file, for example. However, if there is no attack, then our rule will block cat from accessing the user’s file, because the resource access has no filters and is thus not part of the expected attack surface (by our heuristic). However, we may want to allow such access, because cat has filters to protect itself from the input data to prevent attacks such as buffer overflows.

To address such false positives, we propose enforcing protection for such slave programs specially. Our intuition is that when these programs perform adversary-controlled resource access, they can be considered adversarial themselves. All subsequent resources to which data is output by these programs are then considered adversary-accessible. Other programs reading these resources should protect themselves from input (e.g., names) as if they were dealing with an adversary-accessible resource.

To enforce this approach, we implemented two changes. First, we enforce only Invariant 7.2 (confused deputy) in Section 7.3 for these programs. Second, whenever Invariant 7.1 would have
disallowed access, we instead allow access, but “taint” all subsequent output resources by marking them with the adversary’s label (using filesystem extended attributes).

We evaluated the effect of this approach during the bootup sequence of our Ubuntu 10.04 LTS system. We manually identified 15 slave programs. During boot, various startup scripts invoked these slave programs a total of 36 times. In our deployment, 9 of these invocations accessed an adversary-accessible resource. Note that our original approach would have blocked these 9 resource accesses, disrupting the boot sequence, whereas our modification allows these resource accesses. These invocations subsequently tainted 4 output resources – two log files and two files storing runtime state. We found two programs reading from these tainted files – ufw (a simplified firewall), and the wpa_supplicant daemon (used to manage wireless connections). These programs will find the tainted resources adversary-accessible, and will have to protect themselves from such input.

7.9.5 Performance

We evaluated the performance overhead on a hardened Apache webserver (v2.2.22) that had 25 rules. We used ApacheBench to find throughput at two different overheads. We found an overhead of 4.33% and 5.28% for 1 and 100 concurrent clients respectively. However, we can compensate for such overhead by compiling Apache without resource access filters, since filters are now redundant given our enforced rules. Vijayakumar et al. [147] showed that removing code filters causes a throughput increase of 8%.

7.10 Conclusion

In this chapter, we presented automated techniques to protect programs from resource access attacks. We first precisely defined resource access attacks, and then noted the fundamental cause for them – a mismatch between program expectations and the actual environment the program runs in. We proposed a safety property of attack surfaces which guarantees that resource access attacks are only possible where the program is expecting to defend against them. We use a novel insight to retrieve program expectations.

We applied this technique to harden widely-used programs. In this process, we discovered two previously-unknown exploitable vulnerabilities as well as a default misconfiguration in Apache, the world’s most widely used web server. This shows that even mature programs currently reason about resource access in an ad-hoc manner. The analysis as presented in this chapter can thus efficiently and automatically protect programs against resource attacks at runtime.
Chapter 8

Conclusion

With the emergence of targeted malware such as Stuxnet and the continued prevalence of spyware and other types of malicious software, host security has become a critical issue. Attackers exploit vulnerabilities in hosts to break into and compromise them. In this dissertation, we defined and provided techniques to automatically find and defend against a class of previously-disjoint vulnerabilities that we call resource access attacks. These attacks occur when programs fetch resources such as files, and adversaries are able to direct programs to unexpected resources by controlling the name or bindings used in resource access.

The fundamental differentiating factor of resource access attacks is that resource access attacks only become visible when programs are run in their deployment. This is because adversarial control of resource access is determined by the access control policy of the deployment. Using this insight, we provided a technique to identify the attack surface of a program with respect to its deployment SELinux policy in Chapter 4. In this study, we found a manageable attack surface of around 10% of all resource accesses, but also that such attack surfaces were not always obvious or expected. Next, in Chapter 5, we used this attack surface to test programs during resource access, and found more than 25 previously-unknown vulnerabilities across a variety of programs in the widely-used Fedora and Ubuntu distributions. While trying to fix these vulnerabilities, we noticed a reluctance on the part of both programmers to fix code and OS distributors to fix the policy, with each party holding the other responsible. In addition, code and policy fixes were non-trivial tasks. This motivated our work on the Process Firewall (Chapter 6), which combined program introspection with knowledge of the system’s adversary accessibility to protect programs from these vulnerabilities without needing code change or OS access control policy fixes. This work futher demonstrated that program code checks for resource access are more efficiently performed as Process Firewall rules. Finally, in Chapter 7, we provided a technique to produce rules for the Process Firewall by noticing that it was possible to derive the program’s expected attack surface from the filters the programmer had.
placed in code, and limit the deployment’s attack surface to only the expected attack surface. We performed detailed case-studies on a few programs, and showed the ability of our rules to protect against previously-known attacks, as well as discovering three new vulnerabilities in the widely-used Apache webserver. This showed the need to reason in principled ways about a program’s resource access. In summary, this dissertation provides a first step in identifying and providing defenses against the class of resource access attacks, thus improving host security.

8.1 Future Directions

As shown in the dissertation, testing programs in relation to their deployments is necessary to both find deficiencies and defend programs against resource access attacks. Runtime testing is one way to achieve this. However, we found that current runtime test suites that come with programs are not necessarily focussed on testing the functionality or security of resource access. Thus, we propose a testing framework to test programs that is both scalable and directed using the deployment’s adversary accessibility. In particular, in-vivo runtime testing faces challenges of scalability: (i) the system under test may not be powerful enough to compute results of test cases quickly, and (ii) testing entire programs becomes unscalable. To solve (i), my insight is that test systems need only capture, but not evaluate, test cases. A cloned program environment from the test system could be shipped to other locations for testing. In particular, we could use the massive computational power of the cloud for large-scale testing of programs. To address (ii), we use attack-surface directed testing to focus available computational resources. For example, current fuzz testing techniques could be made more scalable and directed by only testing those parts of the program reachable from attack surfaces.

Another direction to explore is the use of static analysis to find vulnerabilities and defend resource access attacks. This dissertation has primarily used runtime testing, which tests programs in their deployment. Static analyses traditionally look at source code in isolation, without considering the deployment. Thus, static analyses such as symbolic execution need to be augmented with a model of their deployment to effectively test for resource access attacks. Such techniques, if sound, might help prove that programs shall be secure in their system deployment given certain constraints, which can be enforced by systems such as the Process Firewall.
Bibliography


“PHP LFI to arbitrary code execution,,” http://www.exploit-db.com/download_pdf/17010/.

URL http://dx.doi.org/10.1147/sj.133.0230


EDUCATION

Doctor of Philosophy, Computer Science and Engineering  
Pennsylvania State University, University Park, PA  
GPA: 3.8/4  
Advisor: Dr. Trent Jaeger  

Bachelor of Engineering, Computer Science and Engineering  
Sri Venkateswara College of Engineering, Anna University, Chennai, India  
GPA: 85% (First class with distinction, 2nd in class of 66)

PROFESSIONAL EXPERIENCE

Research Assistant to Trent Jaeger, Pennsylvania State University, University Park, PA. 2008 - 2014  
- Worked in systems security. Unified a class of previously-disjoint attack classes and provided techniques to detect and defend attacks in an automated and efficient manner, using which several previously-unknown vulnerabilities were detected and defended in commonly used software. Also worked on cloud computing and virtualization security.

Research Intern NEC Labs America, Princeton, NJ. Summer 2013  
- Worked on logging and transforming runtime traces of systems within NEC into an information flow graph, and wrote a subsequent analysis framework. A novel feature of the information flow graph was modeling OS semantics that tracked information flow not directly visible in traces.

Technical Intern Qualcomm Innovation Inc. (QuicInc), Raleigh, NC. Summer 2010  
- Worked as part of team optimizing Linux kernel for Google Chrome netbooks on Qualcomm hardware. Developed software to probe and test graphic stack capabilities in a black-box way, using which several driver bugs were discovered.

Undergraduate Research Assistant, SVCE, India. 2004 - 2007  
- Networks and Distributed Systems: Proposed a modification of shortest path algorithm for distributed systems. Proposed an algorithmic enhancement to a packet classification algorithm for faster update times.

SECURITY VULNERABILITIES REPORTED

- Ubuntu init scripts (arbitrary file create vulnerability (CVE-2011-3151)),
- lightdm (privilege escalation (CVE-2011-4406)),
- Icecat browser - GNU version of Firefox (Untrusted library search path)
- x2go VNC server/client (Untrusted library search path),
- mountall (Untrusted search path),
- apachectl (privilege escalation (CVE-2013-1048))