PROTECTING SERVER PROGRAMS AND SYSTEMS:
PRIVILEGE SEPARATION, ATTACK SURFACE REDUCTION,
AND RISK ASSESSMENT

A Dissertation in
Information Sciences and Technology
by
Jun Wang

© 2015 Jun Wang

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

December 2015
The dissertation of Jun Wang was reviewed and approved* by the following:

Peng Liu
Professor of Information Sciences and Technology
Dissertation Advisor
Chair of Committee

Sencun Zhu
Associate Professor of Computer Science and Engineering & Information Sciences and Technology

Anna Squicciarini
Associate Professor of Information Sciences and Technology

George Kesidis
Professor of Electrical Engineering & Computer Science and Engineering

Andrea Tapia
Associate Professor of Information Sciences and Technology
Director of Graduate Programs, College of Information Sciences and Technology

*Signatures are on file in the Graduate School.
Abstract

In today’s digitized world, server programs and systems have become an indispensable part of people’s daily life and business, such as Web service, file service, database, etc. In the meanwhile, server programs and systems have been attracting more and more attacks and threats, resulting in the reality that they are constantly being targeted and compromised. Besides, the associated impact is becoming larger and larger, ranging from millions of stolen credit card numbers to innumerous Web servers vulnerable and waiting for an emergency security patch.

In this dissertation, we perform a three-dimensional research study emphasizing on protecting server programs and systems, including privilege separation, attack surface reduction, and risk assessment.

First, we explore applying privilege separation to enhance the security of server programs. We design and implement Arbiter, a runtime system targeting at fine-grained privilege separation in multithreaded server programs. In Arbiter, different principal threads can have different privileges to access shared data objects so that the compromise or malfunction of one thread does not lead to data contamination or data leakage of another thread. We leverage page table protection bits and devise a new memory allocation mechanism to achieve efficient reference monitoring. Programmers specify security policy through annotating the source code.

Second, reducing attack surface is an effective preventive measure to strengthen security in large-scale server systems. We propose an automated approach to accurately detect the idling (most likely unused) services and provide ways to reduce their attack surface. We implement this idea and deploy our system in a real working environment of a mid-sized enterprise to identify and constrain unused services that expose attack surface.

Finally, given a server program or system, it is important to evaluate the effectiveness of different security settings and understand the security risks of potential vulnerabilities. We study an emergent type of vulnerability, namely buffer over-read vulnerability, and propose a systematic methodology to model buffer over-read vulnerabilities and quantitatively measure the potential amount of information leakage.
# Table of Contents

List of Figures viii

List of Tables x

Acknowledgments xi

Chapter 1
   Introduction 1
      1.1 Motivations 3
      1.2 Approaches Overview 6
         1.2.1 Arbiter: Practical Fine-gained Privilege Separation in Multi-threaded Server Programs 6
         1.2.2 LIPZip: Discover and Tame Long-running Idling Processes in Server Systems 7
         1.2.3 Risk Assessment of Buffer “Heartbleed” Over-read Vulnerabilities 8
      1.3 Summary of Contributions 9
      1.4 Outline 10

Chapter 2
   Background and Related Work 12
      2.1 Privilege Separation and Least Privilege 12
      2.2 Attack Surface Reduction 15
      2.3 Risk Assessment 17

Chapter 3
   Practical Fine-grained Privilege Separation in Multithreaded Server Programs 19
      3.1 Introduction 19
         3.1.1 Prior Work and Our Motivation 20
3.1.2 Challenges and Our Approach ........................................ 24
3.2 Overview ........................................................................ 25
  3.2.1 Motivating Examples ..................................................... 25
  3.2.2 Threat Model ............................................................ 27
  3.2.3 Problem Statement ...................................................... 27
  3.2.4 System Architecture ..................................................... 28
3.3 Design ........................................................................... 29
  3.3.1 Accessibility ............................................................... 29
  3.3.2 Design Goal ............................................................... 30
  3.3.3 ASMS Mechanism ......................................................... 30
  3.3.4 Label-based Security Model ........................................... 32
  3.3.5 Protection Bits Generation ............................................. 35
  3.3.6 Thread Creation and Context Switch ............................... 36
3.4 Implementation .............................................................. 37
  3.4.1 ASMS Mechanism ......................................................... 37
  3.4.2 Page Fault Handling ..................................................... 39
  3.4.3 Miscellaneous .......................................................... 40
3.5 Application ..................................................................... 42
  3.5.1 Memcached ............................................................... 42
  3.5.2 Cherokee ................................................................. 43
  3.5.3 FUSE ..................................................................... 44
  3.5.4 Summary of Porting Effort .......................................... 45
3.6 Evaluation ....................................................................... 45
  3.6.1 Protection Effectiveness ............................................... 45
  3.6.2 Microbenchmarks ......................................................... 48
  3.6.3 Application Performance ............................................. 50
  3.6.4 CPU and Memory Overhead ....................................... 52
3.7 Discussion ..................................................................... 53
3.8 Summary ..................................................................... 54

Chapter 4
Discover and Tame Long-running Idling Processes in Server Systems ........................................ 55
4.1 Introduction .................................................................... 55
4.2 Motivation and Overview ............................................... 58
  4.2.1 Measurement Study ...................................................... 59
  4.2.2 Problem and Solution .................................................. 60
4.3 Discovery of Idling Processes .......................................... 64
  4.3.1 Periodicity Detection: Background and Challenges .......... 65
  4.3.2 Periodicity Detection on Event Data .............................. 66
Chapter 5
Risk Assessment of Buffer “Heartbleed” Over-read Vulnerabilities

5.1 Introduction ................................................................. 93
5.2 Overview ................................................................. 95
  5.2.1 Background .......................................................... 95
  5.2.2 Threat Model ........................................................ 97
  5.2.3 Motivation and Challenges ...................................... 99
  5.2.4 First Glance .......................................................... 100
5.3 Risk Assessment ........................................................... 101
  5.3.1 How to collect the potential heap buffer information leak? . 102
  5.3.2 How to quantify the potential heap buffer information leak? 103
5.4 Case Studies ............................................................... 104
  5.4.1 Padding at Allocation .............................................. 105
  5.4.2 Erasing at Deallocation (Inline) ................................. 106
  5.4.3 Erasing at Deallocation (Concurrent) ......................... 106
  5.4.4 Concurrent Erasing plus Padding .............................. 108
5.5 Experimental Results .................................................... 108
  5.5.1 Experiment Setup .................................................. 108
  5.5.2 Risk Assessment ................................................... 109
  5.5.3 Performance Comparison ....................................... 113
5.6 Discussion .................................................................... 115
5.7 Summary ..................................................................... 115
# List of Figures

3.1 Motivating examples .................................................. 26
3.2 System architecture. Shaded parts indicate Arbiter’s trusted computing base (TCB). .................................................. 28
3.3 A typical memory layout of ASMS. L1/L2/L3 indicate different accessibility. .................................................. 32
3.4 List of Arbiter API .......................................................... 36
3.5 ASMS memory allocation algorithm ........................................ 39
3.6 Page fault handling diagram in Arbiter ................................. 40
3.7 Arbiter API performance regarding number of threads and allocated ASMS size .................................................. 49
3.8 Performance comparison for Memcached ................................... 50
3.9 Performance comparison for Cherokee ..................................... 51
3.10 Performance comparison for FUSE ...................................... 52
4.1 The number of default running processes ................................. 60
4.2 System workflow ............................................................. 62
4.3 An example of a process’s original event sequence, converted sequence as signal in time domain, and autocorrelation (top to bottom). Periodic and aperiodic event sequences are presented respectively in the left and right sub figures. .................................................. 67
4.4 System Architecture .......................................................... 74
4.5 Distributions for number of instances of unique executables ......... 79
4.6 Distributions of idling processes in different aspects. (a) OS purposes: User/Server/Testbed; (b) Processes types: GUI/non-GUI .......... 80
4.7 Breakdown of system calls that Category I processes are blocked on 81
4.8 Distributions of (a) deviation attempts and (b) converge time ....... 84
5.1 A code snippet related to the Heartbleed vulnerability (from t1_lib.c in openssl-1.0.1f) .................................................. 98
5.2 Illustration of heap buffer over-read in the Heartbleed vulnerability. 98
5.3 A rough estimate on the ratio of data leakage of Heartbleed and all 
memcpy when Heartbleed attack happens (a) one, (b) two, and (c) 
three times during a one-minute user session . . . . . . . . . . . . . . . 101
5.4 Architecture of concurrent heap buffer erase . . . . . . . . . . . . 107
5.5 Experiment setup of the RUBiS benchmark . . . . . . . . . . . . . 109
5.6 Experimental results of Metric 1, 2, and 3 on baseline and the four 
techniques: T0-baseline, T1-padding, T2-erasing (inline), T3-erasing 
(concurrent), T4-erasing (concurrent) + padding. . . . . . . . . . . 110
5.7 Leaked data showing (a) some sensitive information and (b) a valid 
username-password pair . . . . . . . . . . . . . . . . . . . . . . . . . . 112
5.8 Runtime comparison on microbenchmark . . . . . . . . . . . . . . 114
List of Tables

3.1 Different assumptions of 1st and 2nd privilege separation (PS) problem 22
3.2 Accessibility generated from Figure 3.1 ............................ 29
3.3 Summary of porting effort in the amount of source code change ... 45
3.4 Microbenchmark results in Linux and Arbiter ...................... 48
3.5 Comparison of CPU utilization and labeled objects ............... 52
3.6 RSS memory overhead .............................................. 53

4.1 System calls being monitored (partial list) ....................... 71
4.2 Clarification of human involvement in the operational workflow and non-operational (evaluation) workflow .................. 78
4.3 Number of unique instances of idling processes and the corresponding category ..................................................... 79
4.4 Attack surface breakdown of 434 idling processes ............... 83
4.5 Survey results from three system administrators in a university department ......................................................... 90

5.1 Statistics of OpenSSL, Apache, and Nginx ....................... 100
5.2 Metric 4 – Quantity of Unique Sensitive Data .................. 113
5.3 Throughput and memory overhead on Apache ................... 114
Acknowledgments

First of all, I would like to give my most sincere thankfulness to my PhD advisor Professor Peng Liu. I still clearly remember the moment when we had our first conversation over the phone five years ago. He opened a new world of opportunity for me and my journey of PhD study started from then on. During the past five years, he has been continuously supportive, encouraging, inspiring, and understanding. His patient guidance has helped me grow up faster than ever both technically and philosophically. His enthusiasm, dedication, and persistence will continue to influence me on my future life and career.

Second, I would like to thank every one of my collaborators: Xi Xiong, Zhiyun Qian, Zhichun Li, Zhenyu Wu, Junghwan Rhee, Xia Ning, Mingyi Zhao, Qiang Zeng, Gaoyao Xiao, Ping Chen, Jun Xu, Jiang Ming, and Dinghao Wu. Without any of them, I cannot make my way to this dissertation. I would like to express my special thanks to Xi Xiong. His mentorship deepened my understanding in secure systems design and implementation. I also appreciate the internship opportunity at NEC Labs where my mentors Zhiyun Qian, Zhichun Li, and Zhenyu Wu helped me gain lots of technical engineering skills, which later on benefited my PhD study to a very large extent. I also want to give my special thanks to Dinghao Wu for his inspiring discussions.

I would also like to thank my committee members, Sencun Zhu, Anna Squicciarini, and George Kesidis, for their insightful advice and helpful suggestions on my thesis proposal as well as my final dissertation.

In addition, I would like to thank all my labmates, colleagues, and friends at Penn State. Kang Zhao and Fengjun Li offered me a great amount of help, making me quickly adapt to my first-year life and study at Happy Valley. My roommates Zhihao Jiang, Tao Zhang, and Pinyao Guo have provided countless help in those days. Here I would like to express my sincere blessing to Tao Zhang, who has been fighting against heart disease for more than two years and just got heart transplant. I have got lots of encouragement from his optimistic attitude and strong mind and
I wish him getting better and better.

Finally, I give my deepest thanks to my family for their support, encouragement, and understanding, especially when I need them most. My parents Lijun Dong and Baidong Wang offered me all of their love and support throughout my life. My grandfather Qingfeng Wang is one of my most respected person. I learnt so many things from him, about life, work, family and so on. My wife Fang Dong gives me continuous love, encouragement, and inspiration. She will get my deepest love forever. Lastly, I would like to say this year 2015 is a very special year for me. Because my first baby Ethan was born and I became a father. His coming brings me luck and makes me stronger.
Dedication

This dissertation is dedicated to my wife Fang Dong, for giving me continuous love, encouragement and inspiration, and my mother Lijun Dong and father Baidong Wang, for their love and support throughout my life.
In today’s digitized world, server programs and systems are more and more becoming an indispensable part of people’s life and business. People interest with server programs and systems every day. For example, online shopping and online banking involve Web servers and database servers, online file sharing entail file servers or media servers, sending and receiving Emails require mail exchange servers, etc. Therefore, the information security of organizations as well as the personal information security and privacy are highly depending on the security of various server programs and systems.

In practice, however, server programs and systems are unable to enjoy perfect security: vulnerabilities could always exist or be introduced to them. According to the Common Vulnerabilities and Exposures (CVE) database [1], the top trending types of vulnerabilities are either directly targeting or closely related to servers, such as cross-site scripting (XSS), SQL-injection, denial of service, and buffer overflow [2, 3]. On one hand, a common characteristic of server programs and systems is that they are open to public (oftentimes by listening on a network port), accepting, processing, and responding to service requests from clients. As the result, server programs and systems have been attracting lots of interests from malicious users and attackers. On the other hand, server systems and programs are constantly being compromised and exploited. In the past two years or so, we
have seen headline news on security incidents associated with server compromise and/or vulnerabilities at least once of every few months. The impact of such security breaches becomes larger and larger, ranging from millions of stolen credit card information [4] to numerous Web servers being vulnerable and waiting for an emergency security patch [5].

In the meantime, security researchers have never stopped their endeavor towards the design and implementation of secure server programs and systems. There have been a great amount of achievements as well as ongoing efforts in protecting server programs and systems. In this dissertation, we focus on three particular aspects: (1) privilege separation, (2) attack surface reduction, and (3) risk assessment. We explore and discover new ideas and insights in attempt to pursue further advancements in the protection of server programs and systems.

First, from a programmer’s perspective, one would desire to write more secure programs in a less demanding fashion. We propose Arbiter system to explore the solution space. Arbiter is targeting at multithreaded programs and utilizes privilege separation mechanisms to ensure that the malfunction or compromise of one thread does not lead to data leakage or contamination of another thread. The underlying mechanism is transparent to programmers, meaning that programmers only need minimal effort through specifying security policy in their source code.

Second, from a system administrator’s standpoint, reducing attack surface is an effective preventive measure for managing and securing large-scale server systems. We propose LIPZip, an automated approach to detect idling (most likely unused) services that are in either blocked or bookkeeping states. We also present ways to constrain such services in order to remove or reduce their attack surface.

Finally, also from a system administrator’s point of view, given a known or a potential vulnerability within a server, one would like to understand the associated security risks and how to measure the effectiveness of different mitigation techniques. We focus on a particular vulnerability, namely buffer over-read vulnerability, and present a systematic methodology to evaluate the potential risks of buffer over-read
vulnerabilities associated with server programs. Specifically, we model the buffer
over-read vulnerability and propose quantification methods to measure how much
information can be potentially leaked from server programs.

1.1 Motivations

Privilege Separation in Multithreaded Server Programs While multi-
threaded programming brings clear advantages over multiprocessed programming,
the classic multithreaded programming model has an inherent security limitation,
that is, it implicitly assumes that all the threads inside a process are mutually
trusted. This is reflected by the fact that all the threads run in the same address
space and thus share the same privilege to access resources, especially data.

However, one (compromised) thread attacking another thread of the same server
process is a real world threat. For example, an attacker could exploit certain
vulnerabilities in a multithreaded Web server to inject shellcode and access the
private data of another connection served by a different thread. Meanwhile, logic
bugs (e.g., Heartbleed [6]) might exist so that an attacker can fool a thread to steal
private data belonging to other threads.

A common characteristic of multithreaded servers is that they may concurrently
serve different users or clients, which represent distinct principals that usually do
not fully trust each other. This characteristic directly contradicts the “threads-are-
mutually-trusted” assumption. Therefore, a fundamental multithreaded application
security problem arises, that is, how to retrofit the classic multithreaded programming
model so that the “threads-are-mutually-trusted” assumption can be properly relaxed?
In other words, could different principal threads have different privileges to access
shared data objects so that the compromise or malfunction of one thread does not
lead to data contamination or data leakage of another thread?

Existing solutions appear to be insufficient to address this problem. First,
process isolation approaches, such as OpenSSH [7] and Chrome [8], split a monolithic program into least-privilege compartments to realize privilege separation. However, they require the shift of programming paradigm and thus programmers can no longer enjoy the convenience of multithreaded programming. Furthermore, a set of complex inter-process communication (IPC) protocols are needed and, thus, a considerable amount of manual effort is required to design, implement, and verify these protocols. Second, software fault isolation (SFI) [9,10] can make a segment of address space as a protection domain by using software approaches like a compiler. Nevertheless, it is difficult for SFI to map program data objects (e.g., array) into a protection domain: address-based confinement and static instrumentation cannot easily deal with dynamically allocated data. Third, mandatory access control (MAC) systems, for example SELinux [11], can also realize privilege separation. However, they focus on the OS-level operations such as accessing a confidential file, rather than separating the privileges at data level inside a program. Finally, language-based solutions, such as Jif [12], Joe-E [13], and Aeolus [14] can realize information flow control and least privilege at the granularity of program data object. However, they need to rely on type-safe languages like Java. As a result, programmers have to rewrite legacy applications not originally developed in a type-safe language.

**Reducing Attack Surface in Server Systems** Managing the security of server systems is always a challenging task, because the overall security of the entire system is usually determined by the weakest link and today’s server systems are so complex that it is hard to understand which program/process can be the weakest link. Many security breaches start with the compromise of processes running on an inconspicuous workstation. Therefore, it is beneficial to turn off unused services to reduce their corresponding attack surface. Anecdotally, many security best practice guidelines [15,16] suggest system administrators to reduce attack surface by disabling unused services. However, such knowledge needs to be
constantly updated and it is unfortunate that no formal approach has been studied to automatically identify such services.

Prior research has mostly focused on the line of anomaly detection to learn the normal behavior of processes and then constrain them, e.g., by limiting accessible IP/port ranges [17–19] or the system calls that are allowed to be made [20–23]. While such approaches may detect abnormal behaviors of a process (e.g., possibly attacks), they belong to reactive approaches rather than proactive ones. That means, they can detect anomaly only after it happens. Furthermore, the frequent false alerts in anomaly detection make it rarely deployed in practice. On the opposite, identifying idling services represents a much more cost effective solution. Once we determine a program can be safely uninstalled, we can completely eliminate the entire attack surface exposed by a process before an attack could ever happen. Unfortunately, due to lack of formal methods, in state-of-the-art practice, system administrators today could only depend on practical wisdoms from Internet forums [24], and their own experience.

**Risk Assessment of Vulnerabilities in Server Programs** Given a server program or system, it is important to understand the security risks of potential vulnerabilities and to evaluate the effectiveness of different mitigation techniques. In particular, we focus on an emergent type of vulnerability: buffer over-read vulnerability [25], which has gained much attention after the Heartbleed [6] bug causing serious information leakage and monetary lost. A buffer over-read happens when a program overruns a buffer’s boundary and reads the adjacent memory. It is typically resulted from insufficient or a lack of bound checking for buffer read.

Most of previous buffer overflow mitigation approaches focus on over-write, which are either infeasible (e.g., canary) or impractical (e.g., bounds checking) in dealing with over-read vulnerabilities. A number of programming techniques for writing solid and secure code can be helpful in mitigating the buffer over-read vulnerability, including (1) zeroing out memory blocks or buffers at deallocation.
time [26], (2) initializing buffers with special characters, and (3) padding buffers
with special characters. While these techniques are useful in mitigating information
leakage in general, there is no quantitative study on their effectiveness with regard
to information leak through buffer over-reads. Some of these techniques are known
in the field, but were not reported with any quantitative measures before to the
best of our knowledge. As an emerging type of vulnerability, we need more research
on the mitigation of the buffer over-read vulnerability as well as the quantitative
risk assessment of the server systems deployed in the field.

1.2 Approaches Overview

1.2.1 Arbiter: Practical Fine-gained Privilege Separation in Multithreaded Server Programs

We propose Arbiter [27], a system targeting at practical fine-grained privilege
separation in multithreaded server programs. Our goal is to apply least privilege
principle on (shared) data objects so that a data object can be read-writable, read-
only, or inaccessible to different threads at the same time and, more importantly, to
require minimum retrofitting effort from programmers. Our key insight is that page
table protection bits can be leveraged to do efficient reference monitoring, if the
privilege separation policy can be mapped to those protection bits. In particular,
we associate a separate page table to each thread and create a new memory segment
named Arbiter Secure Memory Segment (ASMS) for all threads. ASMS maps
the shared data objects onto the same set of physical pages and set the page
table permission bits according to the privilege separation policy. We design a
new memory allocation mechanism to achieve privilege separation at data-object
granularity on ASMS. We also provide a label-based security model and a set
of APIs for programmers to make source-level annotations to express privilege
separation policy. We design and implement Arbiter based on Linux, including a
new memory allocation mechanism, a policy manager, and a set of kernel primitives.

We apply Arbiter to three types of multithreaded server programs, i.e., an in-memory key/value store (Memcached), a web server (Cherokee), and a userspace file system (FUSE), and show how they can benefit from Arbiter in terms of security. Our own experiences indicate that porting programs to Arbiter is a smooth procedure. The changes to the program source code is 0.5% LOC on average. Regarding performance, our experiments show that the runtime throughput reduction is below 10% and CPU utilization increase is 1.37-1.55×.

1.2.2 LIPZip: Discover and Tame Long-running Idling Processes in Server Systems

We propose LIPZip [28], an automated approach to accurately detect the idling (most likely unused) services as well as a way to remove or reduce their attack surface. A simple example of an idling service is that a long-running service may listen on a certain port, while in reality no one has connected, or will connect to it. This service thus should be disabled to eliminate possible remote exploits. In general, there are two types of idling services. The first type is easy to detect – the service process is completely dormant, blocked waiting for events that would never arrive. For the second type, the service processes regularly wake up from blocked state and perform some “bookkeeping” tasks – simple operations not relevant to the service they provide. The second type is more difficult to identify, since the bookkeeping operations could make these processes appear as active and truly performing services.

We propose LIPdetect (Long-running Idling Process), an effective algorithm that is able to differentiate “bookkeeping” activities from the ones resulted from real workload. The key idea is to identify repeating events with perfect time alignment, which is the indication of being idling. Meanwhile, in case one is not 100% sure whether certain services can be disabled without negative consequences,
it is desirable to keep those services running while still reducing their attack surface. To this end, we also design and implement a tool named procZip that constrains the idling service’s execution to previous known states. Combining idling service detection (LIPdetect) and automated process constraining (procZip), our system is therefore called LIPZip.

We deploy LIPZip in a mid-sized company with both desktop and server machines. We evaluate the effectiveness of idling service detection, and measure how much attack surface can be possibly reduced. We find that around 25% of long-running processes are idling services and that 88.5% of the detected idling services can be constrained with a simple syscall-based policy, which confines the process behaviors within its bookkeeping states. Meanwhile, we work with the IT department personnel to confirm our results. We also cross validate our results by conducting a survey with three senior system administrators [29]. We receive positive feedbacks which show that about 30.6% of such services can be safely disabled or uninstalled directly. In the future, the IT department plan to incorporate the results to build a “smaller” OS installation image.

1.2.3 Risk Assessment of Buffer “Heartbleed” Over-read Vulnerabilities

We present a systematic methodology and conduct a measurement study to evaluate the potential risks of buffer over-read vulnerabilities [30]. First, we model the buffer over-read vulnerabilities and focus on the quantification of how much information can be potentially leaked. We propose four metrics to quantify information leakage, that are, volume of information-carrying bytes, information entropy, sensitive data quantity, and unique sensitive data quantity. Then we perform case studies on four different sets of approaches adopted from existing literature, which are all aimed at mitigating information leakage against buffer over-read attacks. We perform risk assessment using the RUBiS benchmark [31] which is an auction site
prototype modeled after eBay.com. We evaluate the effectiveness and performance of those mitigation techniques and conduct a quantitative risk measurement study. Surprisingly, we find that even simple techniques can achieve significant reduction on information leakage against over-read with reasonable performance penalty.

1.3 Summary of Contributions

This dissertation makes the following contributions.

- We propose a new framework for practical fine-grained privilege separation in multithreaded server programs. Not only is privilege separation ensured, the efficiency and convenience in conventional multithreaded programming paradigm are also preserved. We design a new memory segment abstraction and memory allocation mechanism for controlled sharing among multiple threads with fine-grained access control policy. We create a uniform label model and interface with simple rules for secure cross-thread sharing. Programmers only need lightweight annotations to express security policies.

- We propose an automated approach to identifying long-running “idling” services and show that we can accurately identify such processes. We deploy our solution in a real company environment on 24 hosts. We find that 25% of the long-running processes are idling on average per host and they constitute about 66.7% of attack surface of all long-running processes. 51.1% of the 92 unique binaries have had publicly known vulnerabilities. In collaboration with on-site system administrators and owners of the hosts, we analyze 434 idling processes in detail and classify 30.6% of such services can be disabled to fully eliminate the attack surface. We categorize the common reasons why they can be safely disabled or uninstalled, and show that such results can be shared across enterprises to make recommendations.
• We report the first experience on quantitative risk measurement of information leak through buffer over-read. We provide a summary of the current mitigation techniques that are applicable to buffer over-read and measure their effectiveness. The preliminary quantitative risk measurement and experience we reported can facilitate further studies on the issue.

1.4 Outline

The remaining of this dissertation is structured as follows. Chapter 2 provides a brief review of background and related work. Chapter 3 presents our approach towards practical fine-grained privilege separation in multithreaded server programs. Chapter 4 describe our work on discover and tame long-running idling processes in server systems. In Chapter 5, we present our measurement study on buffer “Heartbleed” over-read vulnerabilities. Finally, Chapter 6 summarizes and concludes this dissertation.

Chapter 3 is organized as follows. Section 3.1 introduces the background, motivation, challenge of the research problem. Section 3.2 provides a high-level overview of our approach. Section 3.3 and Section 3.4 talk about the design and implementation, respectively. Section 3.5 describes how Arbiter can be applied to enhance application security. Section 3.6 presents the evaluation. Section 3.7 discusses some limitations of our approach and Section 3.8 summarizes this chapter.

Chapter 4 is organized as follows. Section 4.1 provides an overall introduction of the proposed approach. Section 4.2 presents the motivation and approach overview. Section 4.3 introduces our methodology for discovering idling processes. Section 4.4 and Section 4.5 present the design and implementation of our system. Section 4.6 presents the evaluation results. Section 4.8 provides the summary of this chapter.

Chapter 5 is organized as follows. Section 5.1 describes the measurement study in general. Section 5.2 introduces the background, threat models, motivation, and examples. Section 5.3 describes the details of our methods and present the metrics
on information leak risk measurement. Section 5.4 shows a few case studies that we use to validate our methods. Section 5.5 presents the experimental results. Section 5.6 discusses some issues and limitations. Section 5.7 summarizes this chapter.
Protecting server programs and systems is in general a broad research topic. We focus on three particular aspects of this topic and this chapter presents the background and related work for each aspect. Section 2.1 introduces privilege separation and least privilege. Section 2.2 talks about attack surface reduction. Section 2.3 covers risk assessment of information leakage and buffer over-read vulnerabilities.

2.1 Privilege Separation and Least Privilege

The principal of "privilege separation" or "least privilege" is a primary principle for secure system design and implementation. In general, privilege separation means that every program and every user of the system should operate using the least set of privileges necessary to complete the job. For example, on Unix operating systems, an application is granted with a special privilege to perform certain operations for authenticated users. Without privilege separation, an adversary who gains unauthorized control over the application may execute the same operations as any authenticated user. To apply the principle of privilege separation, one can leverage the Unix process address space mechanism and spawn an unprivileged child process apart from the privileged parent process. Parent process performs the privileged
operations on behalf of the child process. Therefore, the adversary who gains control over the child is confined in its protection domain and does not gain control over the parent. Privilege separation also facilitates source code audits by reducing the amount of code that needs to be inspected intensively. While all source code requires auditing, the size of code that is most critical to security decreases [7].

Privilege separation has been widely adopted for secure systems design and implementation. It has been applied to system software dated from Multics to micro-kernels [32] and Xen [33,34] to build secure and resilient operating systems and virtual machine monitors. For application software, it has also been widely used to securely construct servers [7,35,36], web applications [37] and browsers [8,38,39].

**Access Control.** Access control has long been a primary mechanism to guarantee that only authorized subjects can get access to the appropriate resources or services. Usually access control matrix is used to express the specification of access control policies. In an access matrix table, each row denotes a subject (an active entity that can execute actions), each column denotes an object (a passive entity storing information), and each entry describes the access rights for the related subject/object pair. Two other expressions of access control policies, which are actually variants of access matrix, are access control lists (ACLs) and capabilities. ACLs are the columns of access matrix defining the access rights associated with each object, while the capabilities are the rows of access matrix describing every subject’s capabilities of accessing other objects [40]. Access control policy can be discretionary or mandatory. In discretionary access control (DAC), the access control policies of different objects are specified to the discretion of their owners or someone else who has been granted with controlling authorization [41]. Mandatory access control (MAC) is different from the DAC in that MAC policies are determined based on fixed regulations by a central authority, rather than the individual owner of an object. Thus the individual owners can neither change the access rights of their objects nor grant specific permissions to other subjects. There is another access control model named role-based access control (RBAC), wherein
access rights are decided according to the roles of the users [42].

In practice, UNIX systems traditionally employ discretionary access control (DAC) mechanism in which users can specify access permissions for their own files. To further provide system-wide policies and centralized control, mandatory access control (MAC) mechanisms like SELinux [11] and AppArmor [43] are proposed for commodity OSes. In MAC systems, the system administrator is responsible for all the type assignments and policy configurations, which could be inflexible and error-prone. Thus, other research approaches are proposed that allow users to create protection domains on their own. For example, Capsicum [44] provides commodity UNIX systems with practical capability primitives to support privilege separation. EROS [45] is a clean-slate designed operating system to achieve capability-based security. At application level, a typical way to achieve privilege separation is to leverage OS-level runtime access control techniques. For example, Apache [35] and OpenSSH [7] split their functionalities into different processes with different UIDs/GIDs, and leverage OS-level access control mechanisms to enforce privilege separation.

**Information Flow Control.** The conventional access control models are only able to prevent unauthorized information from accidental or malicious leakage. However, they are unable to control the dissemination of information. For example, if a program A allows program B to read the file F owned by A, how B might deal with F is totally out of A’s control. In order to prevent information from undesired propagation, it is important to analyze and regulate how information flows within a system. This kind of analysis as well as its enforcement is called information flow control (IFC) [46]. Denning [47] first proposed a lattice-based model that guarantees secure information flow in a computer system. Generally there are two major categories of information flow control policies: (1) *centralized* in which a single administrator is allowed to specify the information flow policy; (2) *decentralized* in which the owner of an arbitrary object can grant or revoke other subject’s access rights on that object.
Denning and Denning [48] first explored the possibility of static program analysis to achieve DIFC. JFlow was proposed to execute static type check during program compilation, will handle and guarantee the appropriate information flow control. For system approaches, HiStar [49] and Asbestos [50] are from-scratch OS designs of decentralized information flow control (DIFC). Flume [51] implements DIFC in commodity Linux. They can enforce the information flow control policy at the OS abstraction granularity, such as processes, files, etc. Laminar [52] and Aeolus [14] leverage Java virtual machine (JVM) to realize DIFC at runtime for Java programs.

**Process Isolation Mechanism.** Process isolation is an effective mechanism to achieve privilege separation in a monolithic application by splitting the application into least-privileged compartments [7,36,53–55]. Provos et al. [7] pioneered the methodology and design of privilege separation. Privtrans [55] can automatically partition a program into a privileged monitor and an unprivileged slave. Wedge [36] combines privilege separation with capabilities to do finer-grained partitioning.

**Software-based Fault Isolation.** Address space isolation puts each process into a protection domain, but does not do finer-grained isolation inside an address space. Software fault isolation [9,10,56] did an innovative work on making a segment of address space as a protection domain by using software approaches like a compiler. LXFI [57] and BGI [58] leverage instrumentation to realize SFI on kernel data and modules. Singularity [59] is a research operating system which leverages language (C# extension) support and static verification to achieve isolation and controlled communication.

## 2.2 Attack Surface Reduction

**Attack Surface Metrics and Reduction.** Traditionally, attack surface has been evaluated at different system layers. Manadhata et al. proposed a software system’s attack surface measurement [60]. This system uses several metrics such as system channels (e.g., sockets), system methods (e.g., API), and the transferred
data items to determine the level of risk. Szefer et al. [61] take a similar approach for virtualization. Authors proposed NoHype, a hypervisor that eliminates the hypervisor attack surface. Kurmus et al. [62] measured attack surface metrics on Linux kernels. Moreover, they reduced the attack surface and kernel vulnerabilities using automated compile-time OS kernel tailoring.

**Unnecessary Service.** Few studies have looked at what services are idling or unnecessary inside an enterprise environment. Guha et al. [63] examined network traffic traces collected from laptops in an enterprise network and looked at services that generate useless traffic (e.g., broadcast discovery messages with no response, failed TCP flows).

**Anomaly Detection and Process Behavior Models.** Numerous attempts have been made to study how to model the behavior of a process and learn its normal behavior for the purpose of performing anomaly detection [20–23]. In general, there are three categories of process behavior models: 1) sequence of system calls [20,64], 2) deterministic Finite State Automata (FSA) [21,22], and 3) probabilistic FSA (e.g., Hidden Markov Model) [23,65]. For 1), the idea is to use n-gram model to learn the common n consecutive system calls. For 2) and 3), the idea is to build a stat machine in order to learn what the next system call is allowed based on the current “state”.

**Vulnerability Discovery.** A number of studies have systematically revealed a class of vulnerabilities related to resource-access. Vijayakumar et al. [17] studied name resolution vulnerabilities in Linux systems. For instance, if a root process reads a file in a public directory, an attacker may be able to remove the file and creates a link to `/etc/passwd` to steal sensitive data. According to their study, a significant number of such vulnerabilities exist. Process firewall [19] has been proposed to help defend against such class of vulnerabilities.

**Systems Service Dependency Discovery.** Understanding the dependency of services is an effective way to understand a program’s behavior and hence to eliminate unnecessary attack surface. Orion [66], Sherlock [67], and eXpose [68]
discover the system dependency using packet headers and timing information. Sherlock in addition uses correlation of packets and conditional probability to determine dependencies. Orion uses computing spikes of delay distribution and thresholds to reduce errors.

While network-based approaches provide convenience to collect traces, often such information is limited due to lack of inside the box and suffer high false positive and negatives. In contrast, host-based approaches offer improved views by tracking the end-to-end views with more specific information on the processes and others system resources. Macroscope [69] provides improved accuracy by combining application instance information and the network information. BackTracker [70] provides backtracking capability of dependent events beyond the identification of system dependencies. BEEP [71] further improves the attack provenance tracking (e.g., backtracking) by partitioning the logs with the unit of semantic application context. By identifying the program’s event-handling loops, it determines disjoint execution slices which reduce unnecessary dependency and hence improvement in the provenance tracking.

2.3 Risk Assessment

Security Risk Analysis. There exist a number of studies focusing on the security risk of services [72–74]. Homer et al. [72] presented a sound and practical modeling of security risk utilizing attack graphs and vulnerability metrics. Chakrabarti et al. [73] models the spread of malware and assesses the benefits of immunization of certain nodes. Chan et al. [74] models the attack and defense as interdependent defense games to study the cost-effectiveness of certain security decisions. Such assessment and modeling is abstract using probability to evaluate the likelihood of compromise.

Information Leakage Quantification and Data Lifetime Analysis There exist approaches that quantify information leaks in a few different aspects [75–77].
For example, [76] proposes to minimize and constrain information leaks in out-bound web traffic by checking fixed data pattern against the HTTP protocol. Heusser and Malacaria [77] presents an automatic method for information-flow analysis that can quantify leaked information equivalence relation and is represented by a logical assertion over program variables. It has been well accepted that sensitive data (e.g., passwords, SSN, and bank accounts) scatters in memory and thus becomes a favorable target of attacks. The investigation by Chow et al. further reveals that many popular applications, such as Mozilla and Apache, take virtually no measures to limit the lifetime of sensitive data they handle [26].

**Buffer Over-read Mitigation.** Zeroing data upon deallocation is an important security practice [78,79], which, however, is not well taken in practice. Chow et al. thus propose to zero data upon deallocation automatically [26]. Harrison and Xu [80] have focused on the disclosure of cryptographic keys at both allocated and deallocated data, and proposed combined solutions to minimize the threat. Zeng et al. [81] propose a new end-to-end defense against zero-day buffer over-read vulnerabilities and evaluate their solution with the Heartbleed attack.
3.1 Introduction

While multithreaded programming brings clear advantages over multiprocessed programming, the classic multithreaded programming model has an inherent security limitation, that is, it implicitly assumes that all the threads inside a process are mutually trusted. This is reflected by the fact that all the threads run in the same address space and thus share the same privilege to access resources, especially data.

However, one thread attacking another thread of the same process is a real world threat. Here are a few examples: (1) For the multithreaded in-memory key/value store Memcached [82], it has been shown that many large public websites had left it open to arbitrary access from Internet [83], making it possible to connect to (a worker thread of) such a server, dump and overwrite cache data belonging to other threads [84]. In addition, vulnerabilities [85, 86] could be exploited by an adversary (e.g., buffer overflow attack via CVE-2009-2415) so that the compromised worker thread can arbitrarily access data privately owned by
other threads. (2) For the multithreaded web server Cherokee [87], an attacker could exploit certain vulnerabilities (e.g., format string CVE-2004-1097) to inject shellcode and thus access the private data of another connection served by a different thread. Meanwhile, logic bugs (e.g., Heartbleed [6]) might exist so that an attacker can fool a thread to steal private data belonging to other threads. (3) For the multithreaded userspace file system FUSE [88], logic flaws or vulnerabilities might also allow one user to read a buffer that contains private data of another user, which violates the access control policy. This is especially critical for encrypted file systems built upon FUSE (e.g., EncFS [89]), wherein data can be stored as cleartext in memory and a malicious user could enjoy a much easier and more elegant way to crack encrypted files than brute force.

A common characteristic of the above programs is that they may concurrently serve different users or clients, which represent distinct principals that usually do not fully trust each other. This characteristic directly contradicts the “threads-are-mutually-trusted” assumption. Therefore, a fundamental multithreaded application security problem arises, that is, how to retrofit the classic multithreaded programming model so that the “threads-are-mutually-trusted” assumption can be properly relaxed? In other words, could different principal threads have different privileges to access shared data objects so that the compromise or malfunction of one thread does not lead to data contamination or data leakage of another thread?

3.1.1 Prior Work and Our Motivation

From a programmer’s point of view, we identify two kinds of privilege separation problems. The first problem is to split a monolithic application into least-privilege compartments. For example, an SSH server only requires root privilege for its monitor (listening to a port and performing authentication), rather than the slave (processing user commands). Since the two parts are usually closely coupled, developers in the old days simply put the two into one program. Due to the
emergence of buffer overflow and other relevant attacks against the root privileged part, however, this monolithic program design is no longer appropriate. Separation of the two parts into different privileged processes with IPC mechanisms in between (e.g., via pipes) becomes a more appropriate approach. Actually, OpenSSH has already adopted this approach.

The second problem is to do fine-grained privilege separation in multithreaded applications. As introduced earlier, threads in a multithreaded program were implicitly assumed to be mutually trusted. However, the evolving of multithreaded applications tends to break this assumption by concurrently serving different principals. Usually the principals do not fully trust one another.

The first problem has been studied for many years [7,36,53–55]. Provos et al. [7] pioneered the methodology and design of privilege separation. Privtrans [55] can automatically partition a program into a privileged monitor and an unprivileged slave. Wedge [36] combines privilege separation with capabilities to do finer-grained partitioning. For the second one, however, there are no systematic research investigations that we are aware of.

This chapter focuses on the second privilege separation problem. Our goal is to apply least privilege principle on (shared) data objects so that a data object can be read-writable, read-only, or inaccessible to different threads at the same time and, more importantly, to require minimum retrofitting effort from programmers. First of all, let’s look at existing mechanisms to see whether they can be applied to solve this problem.

1) Process Isolation. Process isolation is the essential idea behind existing approaches to the first privilege separation problem. OpenSSH [7] and Privtrans [55] leverage process address space isolation while using IPC to make the privileged part and the unprivileged part work together. However, neither of them handles data object granularity. In addition, when there are many principal threads, IPC might become very inefficient. Wedge [36] advances process isolation with new ideas. It creates compartments with default-deny semantics and maps shared data
1st PS Problem (OpenSSH [7], Privtrans [55], Wedge [36], etc.)

2nd PS Problem (Arbiter)

<table>
<thead>
<tr>
<th>Sequential invocation of compartments with different privileges</th>
<th>Concurrent execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Only privileged process/thread can access sensitive data</td>
<td>Data shared among different (unprivileged) principal threads</td>
</tr>
<tr>
<td>Static capability policy</td>
<td>Dynamic (label) policy</td>
</tr>
</tbody>
</table>

Table 3.1. Different assumptions of 1st and 2nd privilege separation (PS) problem

objects into appropriate compartments. However, Wedge is proposed to address the first privilege separation problem, which has very different nature from the problem we consider, as shown in Table 3.1. Due to these differences, Wedge’s all-or-nothing privilege model with default-deny semantic is not very applicable to a multithreaded program, wherein threads by default share lots of resources. To apply Wedge on our problem, one still needs to address the challenges considered in this chapter.

Manually retrofitting a multithreaded program to use multiple processes is possible. However, commodity shared memory mechanisms, such as `shm_open` and `mmap`, do not allow one thread to specify the access right of another thread on the shared memory. Alternatively, designing a sophisticated one-on-one message passing scheme (e.g., using Unix socket) can enforce more control on data. However, the programming difficulty and complexity (e.g., process synchronization, policy handling and checking) could be much higher and thus requires lots of retrofitting effort from programmers.

Another notable idea is to redesign an application from scratch using a multi-process architecture, as what is done in Chrome [8]. However, one of our quick survey reveals that over 80% of existing web servers are multithreaded. It is impractical to redesign all those applications that are already multithreaded.

2) Software Fault Isolation. Address space isolation puts each process into a protection domain, but does not do finer-grained isolation inside an address space. Software fault isolation [9, 10] did an innovative work on making a segment of
address space as a protection domain by using software approaches like a compiler. Nevertheless, it is difficult for SFI to map program data objects (e.g., array) into a protection domain: address-based confinement and static instrumentation cannot easily deal with dynamically allocated data. LXFI \[57\] instruments *kmalloc* so that the principal and address information of dynamic kernel data objects are made aware to the reference monitor (BGI \[58\] achieves similar security properties on Windows.) However, this is done only to kernel modules and kernel data. In addition, LXFI focuses on integrity and does not check memory reads due to performance reasons. However, our goal is to prevent both unauthorized reads and writes. Therefore, we need to catch invalid reads as well.

3) Other Related Mechanisms. We investigate four additional types of related mechanisms to see whether they can handle our problem. (a) OS abstraction level access control has been extensively studied (e.g., SELinux \[11\], AppArmor \[43\], Capsicum \[44\]). However, these mechanisms treat a process/thread as an atomic unit and do not deal with data objects “inside” a process. So a granularity gap exists between these techniques and our goal. (b) HiStar \[49\] is a from-scratch OS design of decentralized information flow control (DIFC). Perhaps HiStar can meet our goal of privilege separation on data objects. However, HiStar does not apply to commodity systems. Besides, to use HiStar to achieve our goal, there still needs to be a major change in the programming paradigm. Flume \[51\] implements DIFC in Linux. However, it focuses on OS-level abstractions such as processes, files, and sockets and thus does not address the privilege separation problem at data object granularity within a multithreaded program. It can be complementary to the approach proposed in this chapter. (c) With the tagged memory in Loki \[90\] or the permission table lookup mechanism in MMP \[91\], as new features to the CPU, access to each individual memory word can be checked. Both methods can enforce privilege separation policy on data objects. However, they require architectural changes to commodity CPUs. (d) Language-based solutions, such as Jif \[12\], Joe-E \[13\], and Aeolus \[14\] can realize information flow control and least
privilege at the granularity of program data object. However, they need to rely on type-safe languages like Java. As a result, programmers have to rewrite legacy programs not originally developed in a type-safe language.

3.1.2 Challenges and Our Approach

We would like to solve this problem in a new way based on this insight: we find that page table protection bits can be leveraged to do efficient reference monitoring, if the privilege separation policy can be mapped to those protection bits. We find that this mapping is possible through a few new kernel primitives and a tailored memory management library. However, doing so still introduces three major challenges:

- **Mapping Challenge (C1)** In the current multithreaded programming paradigm, all the threads in the same process share one set of page tables. This convention, however, would disable the needed mapping from privilege separation policy to protection bits.

- **Allocation Challenge (C2)** To make the protection bits work, data objects that demand distinct privileges cannot be simply allocated onto the same page because this will result in the same access rights. Existing memory management algorithms have difficulty meeting such a requirement because they were not designed to enforce privilege separation.

- **Retrofitting Challenge (C3)** It is challenging to minimize programmers’ retrofitting effort to communicate complex privilege separation policies with the underlying system without modifying the source code drastically.

We present Arbiter to address the above challenges. To address the mapping challenge (C1), we associate a separate page table to each thread and create a new memory segment named Arbiter Secure Memory Segment (ASMS) for all threads. ASMS maps the shared data objects onto the same set of physical pages and set
the page table permission bits according to the privilege separation policy. To deal with the allocation challenge (C2), we design a new memory allocation mechanism to achieve privilege separation at data-object granularity on ASMS. To resolve the retrofitting challenge (C3), we provide a label-based security model and a set of APIs for programmers to make source-level annotations to express privilege separation policy. We design and implement Arbiter based on Linux, including a new memory allocation mechanism, a policy manager, and a set of kernel primitives.

We port three types of multithreaded server programs to Arbiter, i.e., an in-memory key/value store (Memcached), a web server (Cherokee), and a userspace file system (FUSE), and show how they can benefit from Arbiter in terms of security. Our own experiences indicate that porting programs to Arbiter is a smooth procedure. The changes to the program source code is 0.5% LOC on average. Regarding performance, our experiments show that the runtime throughput reduction is below 10% and CPU utilization increase is 1.37-1.55×.

The rest of this chapter is organized as follows. Section 3.2 provides a high-level overview of our approach. Section 3.3 and Section 3.4 talk about the design and implementation, respectively. Section 3.5 describes how Arbiter can be applied to enhance server program security. Section 3.6 presents the evaluation. Section 3.7 discusses some limitations of our approach and Section 3.8 summarizes this chapter.

3.2 Overview

3.2.1 Motivating Examples

Programmers have both intended privilege separation and intended sharing of data objects when writing multithreaded programs. We classify these intentions into three categories.

- **Category 1**: A data object is intended to be exclusively accessed by its
process_active_connections(cherokee_thread_t *thd) {
    buf = (char *) malloc (size);
    len = recv (SOCKET_FD(socket), buf, buf_size, 0);
}

(a) Cherokee-1.2.2

void dispatch_conn_new(...) {
    CQ_ITEM *item = malloc(sizeof(CQ_ITEM));
    cq_push(thread->new_conn_queue, item);
}

(b) Memcached-1.4.13 Main thread

static void *worker_libevent(...) {
    item = cq_pop(me->new_conn_queue);
}

(c) Memcached-1.4.13 Worker thread

Figure 3.1. Motivating examples

creator thread.

Figure 3.1(a) shows the request processing code snippet from Cherokee. The data object buf is allocated by a worker thread and then used to store the incoming packet. Therefore, this data object belongs to that particular worker thread and other worker threads are not supposed to access it.

• **Category 2:** A data object is intended to be accessed by a subset of threads.

Figure 3.1(b) and 3.1(c) show the connection handling code snippets from Memcached. The main thread receives a network request, allocates a data object item to store the connection information, selects a worker thread and then pushes the item into the thread’s connection queue. The worker thread wakes up, dequeues the connection information and handles the request. Ideally, the data object item is only intended to be accessed by the main thread and the particular worker thread, excluding any other worker thread.

• **Category 3:** A data object is intended to be shared among all the threads.

This data sharing intention is commonly seen, especially on metadata. For instance, the struct cherokee_server and the struct fuse store the global
configurations of Cherokee and FUSE, respectively, and are intended to be accessible to all the threads.

Overall, Category 1 and 2 are two very representative privilege separation intentions. Unfortunately, there is actually no such enforcement in real world execution environments. Only the intention in Category 3 has been taken care. We propose Arbiter, a general purpose mechanism so that every category is respected.

3.2.2 Threat Model

We consider two types of threats. First, some threads could get compromised by malicious requests (e.g., buffer overflow attacks, shellcode injection, return-to-libc attacks, ROP attacks). Second, application has certain logic bugs (a.k.a. logic vulnerabilities [92] or logic flaws [93]). For example, the logic bug exploited by HeartBleed [6] can potentially lead to a buffer overread attack, which allows an attacker to steal sensitive information of other users from a web server. In reality, both threats can lead to data leakage and data contamination of a victim thread, which usually result in the compromise of end user’s data secrecy and integrity. Besides, we assume that the application is already properly confined by well-defined OS level access control policies (e.g., which files the application can access) using SELinux, AppArmor, etc. We also assume that the kernel is inside TCB. The fact that the kernel could be compromised is orthogonal to the problem we aim to solve.

3.2.3 Problem Statement

How to deal with the two types of threats through a generic data object-level privilege separation mechanism so that all of the three categories of how a data object is intended to be accessed by threads can get respected?
3.2.4 System Architecture

Figure 3.2 shows the architecture of our system. In Arbiter, threads are created in a new way, resulting in what we call Arbiter threads. Arbiter threads resemble traditional threads in almost every aspect such as shared code segment (.text), data segment (.data,.bss), and open files, but they have a new dynamically allocated memory segment ASMS. To give threads different permissions to access the same data object, we maintain a separate page table for each thread and maps the shared data objects on ASMS to the same set of physical pages. To set the needed permissions, protection bits inside each page table will be set up according to the privilege separation policy. In kernel, these are realized by the ASMS Management component, including system call code plus a set of kernel functions, and the corresponding additions to the page fault handling routine. Due to ASMS, two objects with different accessibility will be allocated on two different pages. By accessibility, we mean which threads can access an object in what way. However, many pages could end up with being half empty by doing so. Our solution is to leverage homogeneity, that is, objects with the same accessibility are put into...
the same page. Such memory allocation is achieved by the ASMS Library.

There are three things a thread needs to go through Arbiter: (1) memory allocation and deallocation, (2) thread creation, and (3) policy configuration. For security purpose, Arbiter threads delegate these operations to the Security Manager running in a different address space via remote procedure calls (RPC).

To specify security policy, programmers will need to make annotations to the source code via the Arbiter API according to our label-based security model. The Security Manager will figure out the permissions at runtime and the page table protection bits will be set up properly before the corresponding data object is accessed by an Arbiter thread.

### 3.3 Design

#### 3.3.1 Accessibility

In our system, accessibility means which threads can access an object in what way. Conceptually, we need to map the aforementioned three categories of intentions onto accessibility before we can enforce fine-grained privilege separation.

<table>
<thead>
<tr>
<th></th>
<th>Main Thread</th>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>A’s Data buf</td>
<td>–</td>
<td>RW</td>
<td>–</td>
</tr>
<tr>
<td>B’s Data buf</td>
<td>–</td>
<td>–</td>
<td>RW</td>
</tr>
<tr>
<td>Shared Data item</td>
<td>RW</td>
<td>R</td>
<td>R</td>
</tr>
</tbody>
</table>

Table 3.2. Accessibility generated from Figure 3.1

Table 3.2 shows a formally defined accessibility generated from the motivating examples in Figure 3.1. Accessibility is defined in terms of a set of threads. Given a set of threads \( \{th_1, \cdots, th_k\} \), the accessibility of data object \( x \) is defined as a vector of \( k \) elements. For example, the accessibility vector of A’s data buf is \( <\emptyset, RW, \emptyset> \). Two data objects have the same accessibility if and only if they have the same vector in term of all of the \( k \) threads.
3.3.2 Design Goal

At a high level, our goal is that through Arbiter the accessibility originated from the privilege separation intentions can be enforced. This goal boils down to the following three design requirements. (1) From a system’s perspective, separate page tables are required in order to enforce accessibility vectors and a synchronized virtual-to-physical mapping is required to make such separation transparent to the threads. (2) From a program’s perspective, a smart memory allocation strategy is required in order to bridge the granularity gap between page-level protection and individual data objects and do it in an efficient way, for which we propose the idea of “same accessibility, same page”. These two requirements lead to the kernel-level and user-level design of ASMS (Section 3.3.3). (3) From a programmer’s perspective, it is important to correctly code accessibility in the program without changing the program drastically. We create a label-based security model and a set of APIs for this purpose (Section 3.3.4). In addition, how Arbiter converts accessibility into protection bits is introduced in Section 3.3.5. Section 3.3.6 discusses the thread creation and context switch issues incurred by our design.

3.3.3 ASMS Mechanism

Kernel Memory Region Management. To grant threads with different permissions to the shared memory, our initial thought was to leverage the file system access control mechanism user/group/others to mmap files with allowed open modes so as to realize different access rights. Since this method has to assign a unique UID for each principal thread, however, it would mess up the original file access permission configurations. In addition, mmap cannot automatically do memory allocation and configuration for multiple sets of page tables in a single invocation.

We design a new memory abstraction called Arbiter Secure Memory Segment (ASMS) to achieve efficient privilege separation. ASMS is a special memory segment
compared to other segments like code, data, stack, heap, etc. The difference is
that when creating or destroying ASMS memory regions for a calling thread, the
operation will also be propagated to all the other Arbiter threads. In other words,
ASMS has a synchronized virtual-to-physical memory mapping for all the Arbiter
threads, yet the access permissions (page protection bits Present and Read/Write)
could be different. Furthermore, only the Security Manager has the privilege of
controlling ASMS. Arbiter threads, in contrast, cannot directly allocate/deallocate
memory on ASMS. Neither can they modify their access rights of ASMS data objects
on their own. Since there is always one physical copy of ASMS, its page fault
handling is also different from copy-on-write or demand paging. Implementation
details can be found in Section 3.4.2. For a legal page fault on ASMS, the shared
page frame rather than a free frame will be mapped. For illegal page faults,
which are mostly security violations in our case, either a segmentation fault or a
programmer-specified signal handler will be triggered.

User-level Memory Management Library. A granularity gap exists be-
tween page-level protection (enabled by the per-page protection bits) and individual
program data objects. Data objects demanding distinct accessibility can no longer
be allocated on the same page. To this end, existing memory allocation algorithms
(e.g., dlmalloc [94]) cannot directly work for ASMS. According to the page table
construct, access rights of a page are described by the protection bits on the corre-
sponding page table entry. This simply implies that all the data objects on a page
can only have one kind of permission for a certain member thread. An intuitive
solution is to allocate one page per data object. However, this is not preferable
mainly because a huge amount of memory will be wasted if the sizes of data objects
are much smaller than the page size.

We design a special memory allocation mechanism for ASMS: permission-
oriented allocation. The key idea is to put data objects with identical accessibility
onto the same page, or “same accessibility, same page”. When we allocate memory
for a new data object \( x \) with accessibility vector \( v \), we search for a page containing
data objects with the same vector v and put x into that page. If that page is full, we search for another candidate page. If all candidate pages are full, we allocate a new page and put x into it. In practice, we allocate from the system one memory block instead of one page per time so as to save the number of system calls. Here a memory block means a contiguous memory area containing multiple pages. Figure 3.3 demonstrates this idea (further details in Section 3.4.1). In this way, both memory waste and performance overhead can be reduced.

3.3.4 Label-based Security Model

To accommodate programmers’ privilege separation intentions, we need a security model for specifying and enforcing accessibility vectors. Our initial attempt was to load the entire accessibility table into the Security Manager as an access control list (ACL) so that it can check and determine each thread’s permission for a data object. However, to regulate each thread’s capability of making allocation requests (e.g., thread A is not allowed to allocate objects that are accessible by everyone) and to deal with dynamic policies (e.g., thread A first grants thread B permission and later on revokes it), ACL is insufficient and further mechanisms must be employed. It is desirable to have a unified and flexible security model.

To achieve unification and flexibility, we develop a label-based security model
wherein threads and data objects are associated with labels so that data access permissions and allocation capabilities can be dynamically derived and enforced. Essentially, it is a special form of “encoding” of the accessibility table. The basic notions and rules follow existing dynamic information flow (DIFC) models [49,52] with a few adaptations. It should be noted that Arbiter itself is not a DIFC system (see Section 3.7 for more discussion).

We use labels to describe the security properties of principal threads and data objects. A label $L$ is a set that consists of secrecy categories and/or integrity categories. For a data object, secrecy categories and integrity categories help to protect its secrecy and integrity, respectively. For a thread, the possession of a secrecy category ($\ast_r$, where $\ast$ represents the name of a category) denotes its read permission to data objects protected by that category; likewise, an integrity category ($\ast_w$) grants a thread the corresponding write permission. Meanwhile, we use the notion ownership $O$ to mark a thread’s privilege to bypass security checks on specific categories. A thread that creates a category also owns that category (i.e., has the ownership). Different from threads, data objects do not have ownership.

We define the rules that govern threads’ permissions and activities as follows:

- **RULE 1 – Data Flow**: We use $L_A \subseteq L_B$, to denote that data can flow from $A$ to $B$ ($A$ and $B$ represent threads or data objects). This means: 1) every secrecy category in $A$ is present in $B$; and 2) every integrity category in $B$ is present in $A$. If the bypassing property of ownership is considered, a thread $T$ can read object $A$ iff:

$$L_A - O_T \subseteq L_T - O_T,$$

which can be written as:

$$L_A \subseteq_{O_T} L_T.$$

33
Similarly, thread $T$ can write object $A$ iff:

$$L_T \subseteq_{O_T} L_A.$$

- **RULE 2 – Thread Creation**: Thread creation is another way of data flow and privilege inheritance. Therefore, a thread $T$ is allowed to create a new thread with label $L$ and ownership $O$ iff:

$$L_T \subseteq_{O_T} L, O \subseteq O_T.$$

- **RULE 3 – Memory Allocation**: Memory allocation also implies that data flows from a thread to the allocated memory. As the result, a thread $T$ can get a memory object allocated on ASMS with label $L$ iff:

$$L_T \subseteq_{O_T} L.$$

Therefore, one could make the following label assignment to realize the accessibility vectors in Table 3.2. For instance, Thread $A$ can read but not write the Shared Data item because of $L_{item} \subseteq_{O_A} L_A$ and $L_A \nsubseteq_{O_A} L_{item}$. Neither can Thread $A$ create a thread with the Main Thread’s privilege ($O_A \nsubseteq O_{Main}$) nor allocate a forged data item ($L_A \nsubseteq_{O_A} L_{item}$). As such, our model unifies permission and capability.

<table>
<thead>
<tr>
<th>Thread</th>
<th>Main</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>label</td>
<td>{}</td>
<td>{mr}</td>
<td>{mr}</td>
</tr>
<tr>
<td>ownership</td>
<td>{mr,mw}</td>
<td>{ar,aw}</td>
<td>{br,bw}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data</th>
<th>A’s Data buf</th>
<th>B’s Data buf</th>
<th>Shared Data item</th>
</tr>
</thead>
<tbody>
<tr>
<td>label</td>
<td>{ar,aw}</td>
<td>{br,bw}</td>
<td>{mr,mw}</td>
</tr>
</tbody>
</table>

The labels are attached by a programmer to the corresponding threads or data objects through annotating the source code via Arbiter API. Figure 3.4 lists the
Arbiter’s API, which are used for labeling, threading, and memory allocation. To preserve the multithreaded programming paradigm, the function syntax is fully compatible with the C Standard Library and the Pthreads Library. For example, if a programmer uses `ab_malloc` without assigning any label (`L = NULL;`), it will behave in the same way as libc `malloc`, i.e., allocating a memory chunk read-writable to every thread. This makes it possible for programmers to incrementally adapt their programs to our system.

### 3.3.5 Protection Bits Generation

The Security Manager is responsible for converting labels to page table protection bits. The Security Manager maintains a real-time registry containing label information of every thread and every ASMS memory block. The conversion happens in two occasions: memory allocation and thread creation. First, whenever a thread wants to allocate memory with certain labels, the Security Manager determines the permissions for every thread by checking our label model, and then invokes our system calls to construct and configure ASMS memory regions accordingly. Second, when a new thread is created, the Security Manager walks through every ASMS memory block, determines the allowed permissions, and initializes the ASMS correspondingly.

Note that in Linux a page table entry is not established until the data on that page is actually accessed. Therefore, the page fault handler will eventually further convert the permissions stored in the flags of ASMS memory regions into page table protection bits (further details in Section 3.4.2). As the result, before a data object is accessed by any thread, the page table protection bits would have been set up properly.
• cat_t create_category(cat_type t);
  Create a new category of type t, which can be either secrecy category CAT_S or integrity category CAT_I.
• void get_label(label_t L);
  Get the label of a thread itself into L.
• void get_ownership(own_t O);
  Get the ownership of a thread itself into O.
• void get_mem_label(void *ptr, label_t L);
  Get the label of a data object into L.
• int ab_pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void *),
  void *arg, label_t L, own_t O);
  Create a new thread with label L and ownership O.
• int ab_pthread_join(pthread_t thread, void **value_ptr);
  Wait for thread termination.
• pthread_t ab_pthread_self(void);
  Get the calling thread ID.
• void *ab_malloc(size_t size, label_t L);
  Allocate dynamic memory on ASMS with label L.
• void ab_free(void *ptr);
  Free dynamic memory on ASMS.
• void *ab_calloc(size_t nmemb, size_t size, label_t L);
  Allocate memory for an array of elements on ASMS with label L.
• void *ab_realloc(void *ptr, size_t size);
  Change the size of the memory on ASMS.
• void *ab_mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset, label_t L);
  Map files to ASMS with label L.
• int get_privilege(pthread_t thread, void *ptr);
  Query the permission of a thread to accessing memory on ASMS.

Figure 3.4. List of Arbiter API

3.3.6 Thread Creation and Context Switch

We identify two options to create an Arbiter thread. Option 1: Conceptually, one can create a new address space for every new Arbiter thread, reconfigure ASMS permissions, and disable copy-on-write for all the other memory segments to retain memory sharing. In this case, although the context switch between two Arbiter
threads will lead to TLB flush (which is just like the context switch between two processes), it can be automatically done by existing kernel procedure and requires no further code modification.

Option 2: A possible optimization is to create a new set of page table only for ASMS when creating a new Arbiter thread. Thus only part of the TLB needs to be flushed during context switch between two Arbiter threads. While this can potentially reduce the TLB-miss rate, it would require lots of modifications to the kernel, especially on the context switch procedure to determine the type of context switch, reload the page table for ASMS, and flush the TLB partially.

In sum, there is a trade-off between “TLB-miss overhead” and “how much code modification is needed”. Both options have pros and cons. We take the first option and our evaluation shows that the performance overhead is already acceptable.

3.4 Implementation

We implement Arbiter based on Linux. This section highlights a few implementation details.

3.4.1 ASMS Mechanism

Kernel Memory Region Management. To properly create or destroy ASMS memory regions in the kernel so as to enlarge or shrink ASMS, we implement a set of kernel functions similar to their Linux equivalents such as `do_mmap` and `do_munmap`. The difference is that when creating or destroying ASMS memory regions for a calling thread, the operation will also be propagated to all the other Arbiter threads. How to configure the protection bits is determined by the arguments passed in from our special system calls (by the Security Manager), including `absys_sbrk`, `absys_mmap`, and `absys_mprotect`. They all have similar semantics to their Linux equivalents, but with additional arguments to denote the permissions.
We add a special flag `AB_VMA` to the `vm_flags` field of the memory region descriptor (`i.e., vm_area_struct`), which differentiates ASMS from other memory segments. The page fault handler also relies on this flag to identify ASMS page faults. To make sure that only the Security Manager can do allocation, deallocation, and protection modification on ASMS memory regions, we modify related system calls, such as `mmap` and `mprotect`, to prevent them from manipulating ASMS.

**User-level Memory Allocation Library.** Built on top of our special ASMS system calls is our user-level memory allocation library Memory blocks are sequentially allocated from the start of ASMS. Some data objects might have larger size and cannot fit in a regular block. In this case, large blocks will be allocated backward starting at the end of ASMS. The pattern of this memory layout is shown in the top half of Figure 3.3. Inside each block, we take advantage of the dlmalloc algorithm [94] to allocate memory chunks for each data object. The bottom half of Figure 3.3 depicts the memory chunks on pages inside a block.

The memory allocation algorithm is shown in Figure 3.5. For clarity, we omit the discussion on the strategy of memory chunk management adopted from dlmalloc.

- **Allocation:** If the size of the data is larger than a regular block size (`i.e., threshold`), a large block will be allocated using `absys_mmap` (line 5). Otherwise, the allocator will search for free chunks inside blocks with that label (line 7). If there is an available free chunk, the allocator simply returns it. If not, the allocator will allocate a new regular block using `absys_sbrk` (line 12).

- **Deallocation:** For a large block, the allocator simply frees it using `absys_munmap` (line 3) so that it can be reused later on. Otherwise, the allocator puts the chunk back to the free list (line 5). Next, the allocator checks if all the chunks on this block are free. If so, this block will be recycled for later use (line 7).
Figure 3.5. ASMS memory allocation algorithm

3.4.2 Page Fault Handling

Figure 3.6 depicts the flow diagram of our page fault handler. A page fault on ASMS typically leads to two possible results: ASMS demand paging and segmentation fault. ASMS demand paging happens when a Arbiter thread legally accesses an ASMS page for the first time. In this case, the page fault handler should find the shared physical page frame and create and configure the corresponding page table entry for the Arbiter thread. The protection bits of the page table entry are determined according to the associated memory region descriptor. In this way, subsequent accesses to this page will be automatically checked by MMU and trapped if illegal. This hardware enforced security check significantly contributes to the runtime performance of Arbiter. An illegal access to an ASMS page will result in a SIGSEGV signal sent to the faulting thread. We implement a kernel procedure do_ab_page as a subprocedure to the default page fault handler to realize the above
Figure 3.6. Page fault handling diagram in Arbiter

3.4.3 Miscellaneous

Application Startup. In Arbiter, an application is always started by a Security Manager. A Security Manager first executes and initializes the needed data structures, such as the label registry. Then, it registers its identity to the kernel so as to get privileges of performing subsequent operations on ASMS. We implement a system call `ab_register` for this purpose. Next, the Security Manager starts the application using `fork` and `exec`, and then blocks until a request coming from the Arbiter threads. The application process can create child thread by calling `ab_pthread_create`, which is implemented based on the system call `clone`. The label and ownership of the new thread, if not specified, default to its parent’s.

RPC. A reliable RPC connection between Arbiter threads and the Security Manager is quite critical in our system. We implement the RPC based on Unix socket. A major advantage of Unix socket for us is about security: it allows a
receiver to get the sender's Unix credentials (e.g., PID), from which the Security Manager is able to verify the identity of the sender. This is especially important in situations where the sender thread is compromised and manipulated by the attacker to send illegal requests or forged information on behalf of an innocent thread.

**Authentication and Authorization.** The Security Manager needs to perform two actions before processing an RPC: authentication and authorization. Authentication helps to make sure the caller is a valid Arbiter thread. This is done by verifying the validity of its PID acquired from the socket. Authorization ensures that the caller has the needed privilege for the requested operation. For example, RULE 2 must be satisfied for a thread creation request, and RULE 3 must be satisfied for a memory allocation request. If either of the two verifications fail, the Security Manager simply returns the RPC with an indication of security violation.

**Futex.** Due to our implementation of thread creation, a problem arises with the futexes (i.e., fast userspace mutex) located on data segment (including both .data and .bss). Multithreaded programs often utilize mutexes and condition variables for mutual exclusion and synchronization. In Pthreads, both of them are implemented using futex. Originally, kernel assigns the key (i.e., identifier) of each futex as either the address of mm_struct if the futex is on an anonymous page or the address of inode if the futex is on a file backed page. In Arbiter, since data segment is anonymous mapping but the mm_struct's of the Arbiter threads are different, kernel will treat the same mutex or condition variable as different ones. Nonetheless, we can force programmers to declare them on ASMS (which resembles file mapping) that does not have this issue. However, we decide to reduce programmers' effort by modifying the corresponding kernel routine get_futex_key and set the key to a same value (i.e., the address of mm_struct of the Security Manager). As such, the futex identification problem is resolved.
3.5 Application

We explore Arbiter’s applicability through case studies across various multithreaded applications. We find that the inter-thread privilege separation problem are indeed real-world security concerns. This section introduces our case studies on three different applications: (1) Memcached, (2) Cherokee, and (3) FUSE.

3.5.1 Memcached

Overview. Memcached [82] is an in-memory data object caching system. It caches data objects from the results of database queries, API calls, or page renderings into memory so that the average response time can be largely reduced. There are mainly three types of threads in a Memcached process: main thread, worker thread, and maintenance thread. Upon arrival of each client request, the main thread first does some preliminary processing (e.g., packet unwrapping) and then dispatches a worker thread to serve that request. Periodically, maintenance threads wake up to maintain some important assets like the hash table.

Security Concern. We identify two potential security concerns. (1) It is reported that a number of large public websites had left Memcached open to arbitrary access from the Internet [83]. This is probably due to the fact that the default configuration of Memcached allows it to accept requests from any IP address plus its authentication support SASL (Simple Authentication and Security Layer) is by default disabled (as Memcached is designed for speed, not security). It has been shown possible to connect to such a server, extract a copy of cache, and write data back to the cache [84]. (2) The vulnerabilities in Memcached [85,86] could be exploited by an adversary (e.g., buffer overflow attack via CVE-2009-2415) so that the compromised worker thread can arbitrarily access data privately owned by other threads.

Retrofitting. We adapt Memcached to realize the accessibility shown in
Table 3.2. In particular, we assume that a Memcached server is used to serve two applications or two users, A and B. Both A and B privately own their cached data objects that are not supposed to be viewed by the other. For the Shared Data, we make \texttt{CQ\_ITEM} and a few other metadata read-writable to the main thread but read-only to the worker threads. We slightly change the original thread dispatching scheme so that requests from different principals can be delivered to the associated worker threads. This modification does not affect other features of Memcached.

### 3.5.2 Cherokee

**Overview.** Cherokee [87] is a multithreaded web server designed for lightweight and high performance. Essentially there is only one type of thread in Cherokee: worker thread. Every worker thread repeats the same procedure, that is, it first checks and accepts new connections, adds the new connections to the per-thread connection list, and then processes requests coming from these connections. All the requests coming from the entire life cycle of a connection will be handled by the same thread.

**Security Concern.** (1) An attacker could exploit the vulnerabilities of the Cherokee (e.g., format string vulnerability CVE-2004-1097) to inject shellcode and thus access the data of another connection served by a different thread. (2) Logic bugs might exist in the web server so that an attacker can fool the thread to overread a buffer, which may contain the data belonging to another connection/thread. A recent bug of this type is the Heartbleed bug in OpenSSL [6].

**Retrofitting.** Our goal is to prevent the threads from accessing each other’s private data without affecting the normal functionality. Therefore, we make the buffers allocated for individual connections only accessible by the corresponding thread. Global data structures are made accessible to all the threads, for example, the \texttt{struct cherokee\_server} which stores the server global configuration, listening sockets file descriptors, mutexes, etc.
3.5.3 FUSE

Overview. FUSE (Filesystem in Userspace) [88] is a widely used framework for developing file systems in user space. Common usages include archive file systems—accessing files inside archives like tar and zip, database file systems—storing files in a relational database or allowing searching using SQL queries, encrypted file systems—storing encrypted files on disk, and network file systems—storing files on remote computers.

When a FUSE volume is mounted, all file system operations against the mount point will be redirected to the FUSE kernel module. The kernel module is registered with a set of callback functions in a multithreaded user space program, which implements the corresponding file system operations. Each worker thread can individually accept and handle kernel callback requests.

Security Concern. (1) Logic flaws like careless boundary checking might allow one user to overread a buffer that contains private data of another user. The two users could have very different file system permissions and thus should not share the same set of files. This is especially critical for encrypted file systems (e.g., EncFS [89]), since the intermediate file data is in memory as cleartext. A malicious user can enjoy a much easier and more elegant way to steal data, compared with cracking the encrypted file on disk by brute force. (2) Although the chance is low due to the limited attack surface, we envision a type of attack in which an attacker can compromise a particular thread and inject shellcode. Then the attacker will be able to directly read the data of another user in memory.

Retrofitting. In general, we make the buffers allocated inside `process_cmd()` private to each thread. The global data structure `struct fuse` is shared among all threads, which contains information like callback function pointers, lookup table, metadata of the mount point, etc. In addition, we change the thread dispatching scheme from round robin to associating users with threads, which is similar to what we do for Memcached.
3.5.4 Summary of Porting Effort

Porting these applications to Arbiter was a smooth experience. Actually, most of our time is spent on understanding the source code and data sharing semantics. After that, we define accessibility and devise label assignments accordingly. Finally, we modify the source code, replacing related thread creation and memory allocation functions with Arbiter API. Table 3.3 summarizes the total LOC and the LOC added/changed for each application.

3.6 Evaluation

We evaluate our system from three aspects: (1) the protection effectiveness (Section 3.6.1), (2) the microbenchmark level runtime performance (Section 3.6.2), and (3) the application level runtime performance (Section 3.6.3).

3.6.1 Protection Effectiveness

As stated in our threat model, we assume that the target application is already properly confined by OS abstraction level access control mechanisms, such as SELinux or AppArmor. To this end, our system can be considered complementary to these OS abstraction level mechanisms. Here our goal is not to evaluate whether our system can achieve the OS abstraction level access control (e.g., preventing a compromised thread from accessing a confidential file). Instead, we want to see under the protection of Arbiter whether a compromised thread can still contaminate or steal the data belonging to another thread.
We assume that an adversary has exploited a program flaw or vulnerability in the three applications ported by us and thus taken control of a worker thread. We simulate various malicious attempts based on the security concerns we presented earlier in Section 3.5.

**Memcached.** We simulate two types of attacks mentioned in Section 3.5.1. (1) We simulate an attacker connecting to Memcached via `telnet`. For the vanilla Memcached, the attacker can successfully extract or overwrite any data using the corresponding keys. On the ported Memcached (protected by Arbiter), our attempts to retrieve data belonging to a different user always fail. (2) We then simulate the scenario presented in Section 3.5.1 to simulate a buffer overflow attack. We assume that B is an attacker. To simplify simulation, we hard-code our “shellcode” in the source code. Our “shellcode” try to overwrite `CQ_ITEM` and read A’s data by traversing the slablist (`(&slabclass[i])->slab_list[j]`). We find that writing to `CQ_ITEM` always fail and traversing the slablist will fail whenever encountering a slab storing A’s data.

Note that in both (1) and (2), a failed attempt always triggers a segmentation fault and thus program crash. In practice, the signal handler can be used with Arbiter to deal with such security violations in a more robust way (e.g., sending no response back or dropping the connection). In our experiments, we simply omit this part.

**Cherokee.** (1) We first simulate the format string attack. We add our “shellcode” to the source code to get another thread’s data via the `header` and `buffer` field of the connection structure (`struct cherokee_connection`), which is referenced by the victim thread’s active connection list (`&thd->active_list`). We observe that both read and write attempts fail without exception. (2) Then we simulate the logic bug. Particularly, we craft a buffer overread bug by substituting the `buf_size` parameter in the `cherokee_socket_write()` function with a number from our input. When we use a small value for `buf_size`, the buffer overread does not fail in most cases because the adjacent memory is also allocated with the same
label. This is tolerable since the attacker only gets the data of his own. When we input a value that is larger than the size of a regular block (i.e., 40KB in our case), the attack always fail. Again, in both (1) and (2), a failure always leads to a segmentation fault in the web server.

**FUSE.** The simulation of FUSE is very similar to what we do for Cherokee. Arbiter can successfully defeat both (1) logic flaw exploits and (2) code injection attacks.

**Counterattacks.** We enumerate a few typical counterattacks that are intended to bypass the Arbiter protection.

1) The adversary may want to call `mprotect` to change the permission of ASMS and then access the data.

2) The adversary may attempt to call `ab_munmap` first and then `ab_mmap` to indirectly modify the permission.

3) The adversary may call `fork` or `pthread_create` to create a normal process or thread that is out of the Security Manager’s control so as to access the data.

4) The adversary may also want to `fork` a child process and let the child process call `ab_register` to set itself as a new Security Manager. In this way, the adversary hopes to gain full control of the ASMS.

5) The adversary forges a reference and fools an innocent thread to access data on behalf of the adversary.

We try each of the above counterattacks for multiple times, but no one succeeds. The reasons are as below. For 1), it is because Arbiter forbids normal system calls including `mprotect` to operate ASMS. For 2), since the adversary does not have permission to access the data, the Security Manager simply denies the `ab_munmap` request. For 3), unfortunately ASMS will not be mapped to the normal processes or threads. For 4), there do exist ASMS now and the child process does gain full control. However, the ASMS no longer has the same physical mapping. Note that the counterattacks 3) and 4) both assume the OS level access control mechanism permits `fork` system call. If this is not the case, 3) and 4) will simply fail in the
very beginning. For 5), it would actually have a chance to succeed. However, Arbiter provides an API `get_privilege` which allows the innocent thread to verify if the requesting thread has the necessary permission. As such, Arbiter can still defeat this counterattack. In sum, we believe that within our threat model no counterattack can succeed.

### 3.6.2 Microbenchmarks

We build a set of microbenchmarks to examine the performance overhead of Arbiter API. Our experiments were run on a Dell T310 server with Intel Xeon quad-core X3440 2.53GHz CPU and 4GB memory. We use 32-bit Ubuntu Linux (10.04.3) with kernel 2.6.32 and glibc 2.11.1. Since we implement the ASMS Library based on uClibc 0.9.32, we use the same version for comparison on memory allocation. Each result is averaged over 1,000 times of repeat.

Table 3.4 shows the comparison of microbenchmarks. The overhead of memory allocation functions (e.g., `ab_free`) is non-trivial. This is because they have to go through the Security Manager via an RPC round trip, which consists of RPC round trips.
marshalling, socket latency, etc. We find that a pure RPC round trip (ab_null) itself already takes 5.84μs, which helps to justify the time consumption of most Arbiter API functions. Due to our implementation of thread creation, we directly use getpid to return the thread ID. As the result, ab_pthread_self runs even faster than its Linux equivalent. In addition to the RPC latency, the system calls made by the Security Manager also contribute to the API overhead. We examine sbrk, mmap, and mprotect and find that Arbiter incurs 28% overhead on average.

There are two other factors that might affect the overhead of Arbiter API: (1) The number of threads can affect the memory allocation overhead. Figure 3.7(a) shows that the time consumption of ab_malloc is roughly correlated with the number of threads. The time consumption increases by around 5.7% per additional thread. This is because memory allocation on ASMS for one thread is also propagated to other threads. For comparison, we also show the result of get_label. This operation does not involve any “propagation” and thus is not affected by the number of threads. (2) The size of allocated ASMS can affect the thread creation overhead. This is because thread creation involves the permission reconfiguration of ASMS. Figure 3.7(b) shows that the time consumption of ab_pthread_create increases along with the size of allocated ASMS (note the logarithmic scale on x-axis). This is also in line with our expectation.

Figure 3.7. Arbiter API performance regarding number of threads and allocated ASMS size
3.6.3 Application Performance

**Memcached.** We build a security-enhanced Memcached based on its version 1.4.13 and we use libMemcached 1.0.5 as the client library. We measure the throughput of two basic operations, SET and GET, with various value sizes and key sizes. The results are compared with unmodified Memcached. In Figure 3.8(a) and 3.8(b), we anchor the key size to 32 bytes and change the value size. In Figure 3.8(c) and 3.8(d), we fix the value size to 256 bytes and adjust the key size. Each point in the figure is an average of 100,000 times of repeat. Altogether, the average performance decrease incurred by Arbiter is about 5.6%.

**Cherokee.** We port Cherokee based on its version 1.2.2. We use the ApacheBench version 2.3 and static HTML files to measure its performance. First, we measure
the influence of file size. We choose files with sizes of 1KB, 10KB, 100KB, and 1MB. Figure 3.9(a) shows the comparison between vanilla Cherokee and the ported version. The average slowdown is 1.8%. Second, we test the system scalability by tuning the number of threads from 5 to 40. We fix the file size to 1KB during this round of test. The throughput comparison is shown in Figure 3.9(b). The average performance degradation is around 3.0%. This comparison indicates that running more threads does not necessarily induce more overhead. For each individual test, we set ApacheBench to issue 10,000 requests with the concurrency level of 10.

**FUSE.** We retrofit FUSE based on its version 2.3.0. For the custom userspace file system, we use the example implementation fusexmp provided by FUSE source package. It simply emulates the native file system. We then select 8 representative commands relevant to file system operations, namely, `cd`, `ls`, `touch`, `cp`, `mv`, `echo`, `cat`, and `rm`. Note that the `echo` command is used to write a 32-byte string to files. Each command is repeated for 10,000 times. Figure 3.10 shows the comparison between unmodified FUSE and the ported version. On average, the
slowdown is 7.4%.

Overall, the application performance overhead is acceptable. This is partially contributed by the fact that the extra cost of Arbiter API calls is amortized by other operations of these programs.

### 3.6.4 CPU and Memory Overhead

<table>
<thead>
<tr>
<th>Application</th>
<th>Original</th>
<th>Arbiter</th>
<th>Overhead</th>
<th>Labeled objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>memcached</td>
<td>49.4%</td>
<td>76.7%</td>
<td>1.55×</td>
<td>14</td>
</tr>
<tr>
<td>cherokee</td>
<td>58.8%</td>
<td>76.1%</td>
<td>1.29×</td>
<td>8</td>
</tr>
<tr>
<td>FUSE</td>
<td>42.3%</td>
<td>58.0%</td>
<td>1.37×</td>
<td>10</td>
</tr>
</tbody>
</table>

*Table 3.5. Comparison of CPU utilization and labeled objects*

In addition to the throughput comparison, we further evaluate the CPU cost. As shown in Table 3.5, Arbiter increases the CPU utilization by 1.29–1.55×. We leverage the CPU time information in /proc/[pid]/stat to do the calculation. We also count the types of labeled objects (not to be confused with runtime instances), shown in the last column of Table 3.5. Interestingly, the number of labeled objects is roughly correlated with the CPU overhead.

Although our “same accessibility, same page” strategy has already come with much less memory waste than “one object per page”, it still incurs some memory
overhead. Table 3.6 shows the average resident memory (RSS) usage of the three applications during the performance test. We measure RSS by checking the \texttt{VmRSS} value of /proc/[pid]/status around ten times per second. Given that the policy we used for the three applications are quite typical, we believe real-world memory overhead should be close to the measured overhead.

<table>
<thead>
<tr>
<th>Application</th>
<th>Original (KB)</th>
<th>Arbiter (KB)</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>memcached</td>
<td>60,664</td>
<td>64,452</td>
<td>6.2%</td>
</tr>
<tr>
<td>cherokee</td>
<td>3,916</td>
<td>4,120</td>
<td>5.2%</td>
</tr>
<tr>
<td>FUSE</td>
<td>732</td>
<td>760</td>
<td>3.9%</td>
</tr>
</tbody>
</table>

Table 3.6. RSS memory overhead

### 3.7 Discussion

We believe that Arbiter provides a generic and practical mechanism for inter-thread privilege separation on data objects. Nonetheless, it still has limitations in defending against certain security threats. When two principal users or clients are served by the same thread, Arbiter can no longer enforce privilege separation for the two principals. Thus, programmers have to be very careful dealing with user authentication and thread dispatching to associate principals with appropriate worker threads. To fully address this issue, one possible solution is to have a per-principal-user “virtual” thread to further separate the privileges. We leave this as a future work.

One limitation of our implementation is that the user-space memory allocator uses a single lock for allocation/deallocation. Therefore, the processing of allocation and deallocation requests have to be serialized. A finer lock granularity can help to improve parallelism and scalability, such as Hoard [95] and TCMalloc [96]. In fact, Arbiter’s memory allocation mechanism inherently has the potential to adopt a per-label lock. We are looking at ways to implement such a parallelized allocator.

Arbiter’s security relies on the correctness of privilege separation policy con-
figured by the programmer. However, it may not be that easy to get all the label assignments correct, especially in complex and dynamic deployment scenarios. Actually, DIFC systems also confront similar policy configuration challenges and research efforts have been made to debug DIFC policy misconfiguration [97]. Our system is also able to incorporate a policy debugging or model checking tool that can verify the correctness of label assignments.

Arbiter’s security model, including notions and rules, is inspired by DIFC. However, it should be noted that Arbiter does not perform information flow tracking inside a program, mainly due to two observations: (1) For a runtime system approach, tracking fine-grained data flow (e.g., moving a 4-byte integer from memory to a CPU register) could incur tremendous overhead, making Arbiter impractical to use; (2) The fact that information flow tracking can enhance security does not logically exclude the possibility of solving real security problems without information flow tracking. The main contribution of Arbiter is that it provides fine-grained privilege separation for data objects using commodity hardware, while still preserving the traditional multithreaded programming paradigm.

3.8 Summary

Arbiter is a system targeting at fine-grained, data object-level privilege separation for multi-principal multithreaded server programs. Particularly, we find that page table protection bits can be leveraged to do efficient reference monitoring if data objects with same accessibility are put into the same page. We find that Arbiter is applicable to a variety of real-world applications. Our experiments demonstrate Arbiter’s ease of adoption, effectiveness of protection, as well as reasonable performance overhead.
Chapter 4  
Discover and Tame Long-running Idling Processes in Server Systems

4.1 Introduction

Managing the security of large server systems is always a challenging task, because the overall security of the entire server system is usually determined by the weakest link and today’s server systems are so complex that it is hard to understand which program/process can be the weakest links. Many security breaches start with the compromise of processes running on an inconspicuous workstation. Therefore, it is beneficial to turn off unused services to reduce their corresponding attack surface. Anecdotally, many security best practice guidelines [15, 16] suggest system administrators to reduce attack surface by disabling unused services. However, such knowledge needs to be constantly updated and it is unfortunate that no formal approach has been studied to automatically identify such services.

Prior research has mostly focused on the line of anomaly detection to learn the normal behavior of processes and then constrain them, e.g., by limiting accessible IP/port ranges [17–19] or the system calls that are allowed to be made [20–23].
While such approaches may detect abnormal behaviors of a process (e.g., possibly attacks), they belong to reactive approaches rather than preventive ones. Obviously, reducing attack surface is a preventive measure that complements anomaly detection. The strong semantic of idling services enables more specific treatments (e.g., disable or even uninstall them) as opposed to the general treatments to anomalies, e.g., raising a security alert. Unfortunately, the frequent false alerts in anomaly detection make it rarely deployed in practice. Identifying idling services, on the other hand, represent a much more cost effective solution where each service only needs to be examined mostly once (e.g., to determine if it can be safely uninstalled). Once a positive decision is made, it completely eliminates the entire attack surface of a process before an attack could ever happen. Unfortunately, due to lack of formal methods, in state-of-the-art practice, system administrators today could only depend on practical wisdoms from Internet forums [24], and their own experience.

One might argue that system administrators or end-users should setup their servers and desktop machines to only run useful services and applications in the first place. However, in practice, this is a daunting task. The modern OS vendors tend to stuff more and more functionalities into their distributions, and provide a “one size fits all” system. Thus, many services might not be useful for a particular user. Even worse, today’s server systems typically run a variety of OS distributions. As a result, it is not easy for system administrators to keep up with the purpose of each and every service. Furthermore, systems keep evolving and users’ demands keep changing. Given the complex dependencies inside server systems, people tend to keep things as they are. Such practice, therefore, negligibly leaves numerous idling services running, and thus exposes unnecessary attack surface.

In this chapter, we propose an automated method to detect “idle/idling” services and also provide ways to reduce or minimize their attack surface. An idling service is a service that does not serve real workload, but rather running in a blocked or bookkeeping state. A simple example is that a long-running service may listen on a certain port, while in reality no one has connected, or will connect to it. This
service thus should be disabled to eliminate possible remote exploits. In our study, we find 25% of long-running processes are idling services.

In general, there are two types of idling services. The first type is easy to detect – the service process is completely dormant, blocked waiting for events that would never arrive. For the second type, the service processes regularly wake up from blocked state and perform some “bookkeeping” tasks – simple operations not relevant to the service they provide. The second type is more difficult to identify, since the bookkeeping operations could make these processes appear as active and truly performing services. In order to address the problem of identifying both types of idling services, we propose LIPdetect (Long-running Idling Process), an effective algorithm that is able to differentiate “bookkeeping” activities from the ones resulted from real workload.

While LIPdetect could identify and report idling services to system administrators, working with the IT department of a mid-sized company we find in some cases the system administrators are not 100% sure whether certain services can be disabled without negative consequences. It is therefore desirable to keep those services running while still reducing their attack surface. In such cases, we fall back to the traditional wisdom to constrain the runtime behavior of a process. To this end, we design a simple tool named procZip that constrains the idling service’s execution to previous known states. Combining idling service detection (LIPdetect) and automated process constraining (procZip), our system is therefore called LIPzip.

We evaluate the results of idling service detection, and perform a measurement study in an enterprise with both desktop and server machines to show how much attack surface can be possibly reduced. Meanwhile, we work with the IT department personnel to confirm our results. We also cross validate our results in a university department [29].

In summary, we made the following contributions:
• We propose an approach to identifying the long-running “idling” services and show that we can accurately identify such processes (with less than 1.5% mis-classified).

• We deploy our solution in a real server systems environment on 24 hosts. We find that 25% of the long-running processes are idling on average per host and they constitute about 66.7% of attack surface of all long-running processes. 51.1% of the 92 unique binaries have had publicly known vulnerabilities.

• In collaboration with on-site system administrators and owners of the hosts, we analyze 434 idling processes in detail and classify 30.6% of such services can be disabled to fully eliminate the attack surface. We categorize the common reasons why they can be safely disabled or uninstalled, and show that such results can be shared across organizations to make recommendations.

• We design procZip and show that even though the system administrators do not have the confidence to disable all services discovered by us, 88.5% of the idling services can be safely constrained by the procZip during a two-week evaluation, indicating that they are continuing to be idling.

The rest of this chapter is organized as follows. Section 4.2 presents the motivation and overview of our approach. Section 4.3 introduces our methodology for discovering idling processes. Section 4.4 and Section 4.5 present the design and implementation of our system, respectively. Evaluation results are presented in Section 4.6. The summary of this chapter is provided in Section 4.8.

### 4.2 Motivation and Overview

Our motivation comes from the following observations:

1. Today’s computer systems are equipped with an increasing number of functionalities (and thus also the attack surface) with newly developed technologies,
perhaps much more than necessary for everyday use.

2. Systems are constantly evolving; more applications or features are installed (or updated) over time but they are left there even when no longer needed.

3. What makes the problem worse, in the Linux world, there are a large number of different OS distributions which make the IT systems largely heterogeneous and fragmented.

As a result, it is highly non-trivial to control or even understand all the systems precisely, let alone knowing whether a process is actually performing useful work or simply “idling”. According to the IT department of the enterprise, they indeed do not keep track of what Linux distributions or applications are installed, the reason being that they do not want to get in the way of users doing what they want to do.

### 4.2.1 Measurement Study

To understand to what extent each observation is true, we conduct the following measurements based on two data sources. First, we look into the historical Linux distributions of Ubuntu and Red Hat from public images that are available on the Amazon EC2 service. Second, we collect information about 67 hosts in a real enterprise.

**Default Long-running Processes.** We look into the two popular Linux distributions Ubuntu and Red Hat over the past six years and count the number of daemon processes that are running by default after the system boots. Figure 4.1 shows that the number is steadily increasing over years. Note that the number from the server distributions is not very high due to the fact that these images are tailored to run on servers and thus many desktop features are not installed by default. In general, the increase of the default processes might render more and more idling processes.
Figure 4.1. The number of default running processes

**Heterogeneity.** Inside the enterprise, we study 67 Linux hosts including both
desktops and servers. There are 7 Fedora, 11 CentOS, and 49 Ubuntu hosts respec-
tively. These hosts include 3 unique Fedora versions, 6 unique CentOS versions, and
7 unique Ubuntu versions. This demonstrates the universal heterogeneity of Linux
distributions. Indeed, the management complexity rooted from the heterogeneity
is partly echoed by users asking questions on public forums about what a specific
process is for and why it is running [98–100].

In sum, all these evidences imply that there are potentially many idling services
in average server systems.

### 4.2.2 Problem and Solution

**Problem statement.** The key problem we are addressing is how to reduce the
attack surface incurred by long-running daemon processes while minimizing the
impact on the normal operations. The first challenge is the lack of a common
definition on what types of processes can be safely disabled or uninstalled.

**Long-running Idling process.** Our definition of long-running idling processes
comes from the following intuition. As shown in a recent study [71], long-running
processes are typically structured as a giant loop, taking certain input in the
beginning of the loop and performing different operations based on the input. Such
a long-running process will be in one of the three states: (1) The process shows no
program activity in terms of instructions or system calls, e.g., blocked on certain
system calls waiting for input. (2) The process repeatedly cycles through a loop to
check certain input channels, e.g., polling on sockets or checking the existence of a
file. However, the input they receive has low entropy (same repeated input) and
does not trigger any (useful) operation or they may not even receive any input at
all (e.g., the poll on sockets always timeout). As stated in Section 4.3, the challenge
here is that such repeating events need to happen with perfect time alignment in
order to be safely considered idling. (3) The process is taking input and perform
useful operations based on the input.

In this work, we define the idling processes as processes in state (1) or (2). Our
insight is that if we can identify a list of idling processes with decent accuracy, it is
very likely to have little impact on normal operations if we disable them.

**Workflow.** Our system operates as shown in Figure 4.2. First, we collect
data from hosts through monitoring agents that we developed. The details about
the agent is described in Section 4.4. Then the data are taken as input to detect
idling processes (details described in Section 4.3). Then we produce the list of
idling processes and characterize their attack surface (explained in the following
paragraphs). Next, we subject the process with attack surface\(^1\) to two possible
security actions.

Action 1: system administrators are usually in the best position to examine
the list of idling processes and decide if they are needed. As shown in Section 4.7,
we argue that such knowledge is required to determine the future necessity of a
process with high confidence. In our evaluation, we show that administrators in the
enterprise confirm many of the idling processes or services can indeed be disabled
or uninstalled without interrupting normal operations. We further correlate the

---
\(^1\)We are focusing on security and hence interested in attack surface. Different prioritizing
metrics are also possible (e.g., physical memory consumption).
results with an IT group in a university department and show that the results can be shared effectively across organizations.

Action 2: similar to anomaly detection techniques, an alternative action is to “quarantine” or constrain their executions within the behaviors observed in the past (e.g., checking the presence of a file). In this way, the process can still continue to receive any potential triggering events. When the triggering events do arrive in the future, we allow the process to be “freed” as long as the privileged user chooses to, e.g., entering the root credentials or confirming a link of an email sent to a pre-registered address. Here, the idea is to reduce the attack surface of processes but still promptly allowing them to continue execution when they really need to. Note that the strategy is flawed if we simply un-quarantine the process whenever we see that the process is no longer idling as it could be possibly triggered by an attack that is actively trying to exploit a vulnerability of this process. Even though this action is operationally similar to anomaly detection in general, we point out that existing models from anomaly detection, e.g., system call sequence or Finite State Automata (FSA), do not suit our problem (described in Section 4.3). So we need to develop new models and methods.

Ideally, Action 1 is generally preferred since it can completely remove the attack surface. In practice, Action 2 can be used as a backup for usability purpose.

**Attack Surface.** Similar to a previous work [17], we only consider system resources with “privilege discrepancy” such as files and sockets as attack surface.
For instance, if a process is reading a file that is owned by the same user who
started the process, we do not consider it an attack surface. As Section 4.6.2
shows, 29.3% of the idling processes we discover have direct attack surface exposed
(e.g., listening sockets or reading a file owned by a different user). More details
in determining the attack surface is presented in Section 4.6. Based on the kind
of attack surface they expose, we can prioritize them for system administrators.
Note that the attack surface we compute is only the lower bound. In fact, many
of them may expose additional attack surface once becoming active. As a result,
the percentage of idling processes that have attack surface in idling state is only a
conservative security metric.

Goals and Non-goals. The goal of this chapter is to investigate the security
impact of idling processes and propose automated methods to identify them. We
argue that the knowledge of the existence of such idling processes needs to be made
more readily available to the community.

Non-goals are:

- To develop counter-measures against specific types of attacks. Rather, based
  on well-understood metrics of attack surface, our system is to reduce the
  likelihood of successful attacks.

- To come up with another anomaly detection algorithm. Instead, since the
  semantic of “idling processes” is much stronger than the abstract “normal
  behavior” in anomaly detection, we can apply more specific treatment to
disable or uninstall such idling services. Even if we do constrain them still,
there is a much less false alert rate.

- To design a perfect metric to compute the security risks or attack surface of
  a process. Instead, we leverage known metrics to achieve our goal.
4.3 Discovery of Idling Processes

As stated in Section 4.2.2, we are targeting at two categories of long-running processes. It is trivial to detect the **Category I (Completely Idle)** processes. As long as the process does not have any CPU time (available in the procfs) for longer than a significant time period (e.g., one week) they are considered category I. This type of processes are usually blocked on certain blocking system calls and does not get a chance to return. However, it is much more subtle to detect the **Category II (Periodic/Repeating)**.

The rationale behind our method to detect category II processes is that an active process typically interacts with users and/or external inputs which should have non-trivial dynamics in size, inter-arrival time of requests, or connection timeout, etc. If a process repeats regularly over days and nights, it is highly likely that this process does not involve human-driven workflow, and thus can be considered idling. One might argue that batch jobs (e.g., cron jobs) may also repeat daily or weekly, but they should not be considered idling. We would like to clarify that we consider repeating patterns at the system call level, including parameters and return values, instead of the “job” level. So even for a repeating cron job, e.g., data backup, running daily could see different sets of files every time and therefore generate different sequence of system calls (and different parameters).

**Possible Solution from Existing Process Behavior Models.** As described in Section 4.7, there are several mainstream process behavior models proposed before. Technically, we consider our problem a special case to model certain aspects of a process. We investigate if existing models are suitable to solve our problem. Not to disrupt the flow of the chapter much, we present such discussion in more details in Section 4.7. In short, existing models do not fit the idling process detection problem due to two main reasons: 1) Many approaches model the behavior of a process based on a short history of events. For instance, the n-gram-based model only considers the n consecutive events [20]. The Finite State Automata (FSA)
models only constrain the next event based on the current state. 2) While the sequence of events have been modeled in several ways, the time information such as the arrival time of events has not been fully considered. We should be able to analyze periodic behavior which is composed of a collection of diverse arrival time of events to accurately characterize idleness.

We note that the timing of events can be critical. For example, a busy web server may run in a loop and serve the same page and carry out the same set of operations repetitively. Therefore, the sequence of events may be perfectly repeating without considering the event timing. Hence, to address our problem, one has to have a careful treatment of timing, which is typically not entailed in anomaly detection.

### 4.3.1 Periodicity Detection: Background and Challenges

To find repeated patterns from sequence data is a common problem in many research areas such as data mining and signal processing [101]. A popular way to find the period from data sequences is via Fast Fourier Transform (FFT). However, only when the sequence is known as periodic as a priori, FFT can give the period. FFT is not able to detect whether a sequence is periodic or not.

In LIPdetect, we adopt the idea of autocorrelation to detect periodicity. Autocorrelation measures the correlation relation between a signal and itself after shift with a given lag [101]. For a sequence $S = \{X_1, X_2, \cdots, X_n\}$ and a lag $k$ ($0 \leq k < n$), autocorrelation of $S$ with lag $k$ is calculated between sequence $\{X_{k+1}, X_{k+2}, \cdots, X_n\}$ and $\{X_1, X_2, \cdots, X_{n-k-1}, X_{n-k}\}$ using the following formula,

$$r(k) = \frac{1}{(n-k)\sigma^2} \sum_{t=1}^{n-k} (X_t - \mu)(X_{t+k} - \mu), \quad (4.1)$$

where $\mu$ and $\sigma^2$ are the mean and variance of the sequence $S$. The value of autocorrelation falls in $[-1, 1]$, where larger absolute values mean stronger correlations and the positive (negative) sign means positively (negatively) correlated.
If a sequence is perfectly periodic with period $p$, then the autocorrelations of the sequence will be maximized (i.e., autocorrelation equals to 1) when the lag $k$ is a multiple of $p$, that is, $k = mp$, where $m$ is an integer ($m = 1, 2, 3, \cdots$). Due to this property, autocorrelation can be used to detect periodicity of sequences. Curious readers can refer to Figure 4.3 to get an early sense of how autocorrelation looks like.

However, autocorrelation cannot be directly applied in our case to detect periodicity of a process. The primary reason is that as in Equation 4.1, $X_t$’s are numerical values and thus their mean and variance, as well as the multiplication and addition operations on the values, are well defined. However, in event sequences, $X_t$’s are categorical values (i.e., event types) and all the operations as in Equation 4.1 are not well defined on categorical values (e.g., it does not make sense to add event type 1 and event type 2 to get event type 3). Another issue with applying autocorrelation on event sequences is that autocorrelation cannot capture the time information between two events. In a process, different system calls may introduce different intermediate idle time intervals, and the length of the time intervals carries useful information in inferring process periodicity. Autocorrelation has no mechanism to encode such timing information.

4.3.2 Periodicity Detection on Event Data

We propose a new autocorrelation measure, denoted as E-Autocorrelation, particularly for event data, which have categorical values and varying time gap between events.

4.3.2.1 E-Autocorrelation

Correlation Measure. To handle categorical value, we changed the correlation measure from Equation 4.1 to:
Figure 4.3. An example of a process’s original event sequence, converted sequence as signal in time domain, and autocorrelation (top to bottom). Periodic and aperiodic event sequences are presented respectively in the left and right sub figures.
\[ r_e(k) = \frac{1}{n-k} \sum_{t=1}^{n-k} \mathbb{I}(X_t, X_{t+k}), \]  \hspace{1cm} (4.2)

where \( \mathbb{I}(\cdot, \cdot) \) is an identity function defined as follows:

\[
\mathbb{I}(x, y) = \begin{cases} 
1 & \text{if } x \text{ and } y \text{ are of same type} \\
0 & \text{otherwise.} 
\end{cases}
\]  \hspace{1cm} (4.3)

The value of \( r_e \) falls in \{0, 1\}. By \( r_e \), an event sequence is highly correlated if its shifted version and itself has many same event types aligned.

E-Autocorrelation shares the same property with the conventional autocorrelation, that is, when a periodic event sequence is shifted with a lag that is exactly a multiple of the period, \( r_e \) achieves its maximum. Thus, E-Autocorrelation is suitable for periodicity detection in event sequences.

**Two-step Periodicity Detection.** E-Autocorrelation employs a two-step approach to find periodic events in terms of both event types and timing. For example, a pattern of interest would look like open-read-close. Patterns open-read-idle-close and open-idle-read-close will be considered as different patterns since the time interval for idling is at different places or has different length.

**Step-1** is to identify periodicity from event sequences using E-Autocorrelation with all timing information ignored. For an event sequence \( \{X_1, X_2, X_3, \cdots, X_n\} \), we first calculate the E-Autocorrelation value sequence \( \{r_e(1), r_e(2), r_e(3), \cdots, r_e(n-1)\} \). Then from the E-Autocorrelation sequence, we find the values that are no less than a threshold \( \alpha \) (\( \alpha \leq 1 \)) and consider them as peaks. Note that in a perfectly periodic sequence, E-Autocorrelation values as 1 indicate periodicity. However, since the event sequences could be noisy, choosing smaller values than 1 as peaks provides a chance for our approach to better tolerate noises and thus remain robust. We use \( \alpha = 0.95 \) based on our experiments using a one-month training dataset.

After the peaks are located, whether the event sequence (again, without time information) is periodic is determined by looking at the standard deviation \( \sigma \) of
the distances between consecutive peaks, where the distances are calculated as
the difference of lag values corresponding to the peaks. Our analysis on datasets
with ground truth shows that for periodic sequences $\sigma$ is typically smaller than 0.1,
whereas for non-periodic sequences $\sigma$ is typically larger than 10 or even 100. This
indicates that $\sigma$ could serve as a parameter to differentiate periodic sequences from
non-periodic sequences. Thus, we test $\sigma$ against a pre-defined threshold $T$ (we set
$T$ as 0), and only when $\sigma \leq T$, the sequence is considered periodic.

**Step-2:** The periodic patterns recognized from Step-1 could be incorrect since
time interval information is completely ignored. It is still possible that, for example,
the sequence $\{A, B, C, A, B, C, A, B, C\}$ is determined as periodic from Step-1.
However, the duration of the first $A, B, C$ is 1,000 seconds, whereas the duration of
the second $A, B, C$ is 1 seconds and the duration of the last one is 0.001 seconds.
In this case, it is not reasonable to claim that the entire sequence is periodic. Thus,
in the second step, the results from Step-1 are post-processed so as to filter out
the cases that show no periodicity (or with incorrect period) in terms of time. To
do so, we find the events that appear first in the periodic pattern identified from
Step-1. In the example of sequence $\{A, B, C, A, B, C, A, B, C\}$, such events are the
first event (the first $A$), the fourth event (the second $A$) and the seventh event (the
third $A$) since the periodic pattern is $A, B, C$. This can be done by looking at the
lags corresponding to peaks, that is, if a peak occurs at lag value $k$, then the $k$-th
event is the one that appears first in the periodic pattern. Such events are denoted
as peak events.

Next, for each peak event at lag $k$, we need to validate the corresponding period
by checking if any event always repeats itself after time $t_k$. If the validation fails,
we drop the corresponding peak. After iterating through all peak events, we are
left with a subset of peak events, which correspond to the true periodic pattern. In
order to do the validation, we construct a time sequence $\{t_1, t_2, t_3, \cdots, t_m\}$ where $t_i$
is the timestamp of the $i$-th peak event. Then we compute the standard deviation of
a time interval sequence $\{t_2 - t_1, t_3 - t_2, \cdots, t_m - t_{m-1}\}$. If this standard deviation
is close to 0, then we consider the original sequence periodic and the true period is the mean of the time interval sequence. Otherwise, we calculate and test the standard deviation of sequence \( \{t_3 - t_1, t_4 - t_2, \ldots, t_m - t_{m-2}\} \). The above procedure is repeated until a period is identified or the length of the time interval sequence is 1 (i.e., all the peak events are tested).

Figure 4.3 shows an example of original event sequences and their autocorrelations. The raw event sequences are shown in the figures in the top. Their converted time-domain signals and autocorrelations are presented below. The left sub figures show a periodic event sequence and the corresponding autocorrelation; the right ones show a non-periodic event sequence. Our E-Autocorrelation can effectively differentiate periodic sequences from non-periodic ones and accurately identify the repeating cycles for periodic sequences.

### 4.4 System Design

In this section, we describe the design of the two components of LIPzip – data collection and process constraining.

#### 4.4.1 Data Collection

While it is possible to monitor all activities of every process by using kernel debugging or auditing facilities, such as `ptrace` or `auditd`, doing so would incur significant performance penalty and seriously impact the normal usage of the system being monitored. In theory, it is straightforward to run system call tracers such as `strace` or `auditd` to monitor system calls for all threads of all processes. However, due to the fact we cannot predict which processes are idle, the overhead to monitor all system calls of all processes can be prohibitively high – to the point that our administrators and users are not comfortable allowing a real deployment. In order to limit the monitoring overhead to an acceptable level, and allow for large scale
In the first stage, we perform light-weight monitoring for all processes on a system, by collecting partial information (such as time stamp, system call type, and error code) for a subset of security-relevant system calls. For instance, our observation indicates that read and write are among the most frequently used system calls. However, a process must first open a file before it could perform read/write operations on the file, and the usage of open and close system calls are over an order of magnitude less. And thus, by inferring read/write operations from open and close system calls, we deduct more than 90% monitoring overhead. We systematically studied the properties of every system call, and selected system calls that are crucial for our analysis, categorized into four types, as shown in Table 4.1.

<table>
<thead>
<tr>
<th>Event Type</th>
<th>System Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>process event</td>
<td>fork, vfork, clone, execve, exit, exit_group</td>
</tr>
<tr>
<td>network event</td>
<td>socket, bind, connect, accept</td>
</tr>
<tr>
<td>file event</td>
<td>open, creat, stat, access, dup, close</td>
</tr>
<tr>
<td>IPC event</td>
<td>pipe, socketpair, shmget, msgget, socket, bind, connect, accept</td>
</tr>
</tbody>
</table>

Table 4.1. System calls being monitored (partial list)

deployment in a real working environment, we design a two-stage data collection and screening mechanism.

In the first stage, we perform light-weight monitoring for all processes on a system, by collecting partial information (such as time stamp, system call type, and error code) for a subset of security-relevant system calls. For instance, our observation indicates that read and write are among the most frequently used system calls. However, a process must first open a file before it could perform read/write operations on the file, and the usage of open and close system calls are over an order of magnitude less. And thus, by inferring read/write operations from open and close system calls, we deduct more than 90% monitoring overhead. We systematically studied the properties of every system call, and selected system calls that are crucial for our analysis, categorized into four types, as shown in Table 4.1.

With data collected from the first stage monitoring, we apply our E-autocorrelation to screen all processes and discover those that are likely to be “idling”. However, due to the incomplete set of system calls, this screening stage produces a small number of false positives (around 18.2% according to our empirical observation) – i.e., some processes classified as “Category I” (complete idle) do in fact make very light-weight system calls which we do not monitor; similarly, some processes classified as “Category II” (periodic) make unmonitored system calls in non-periodic manner.

In order to remove the false positives, we subject those processes to the second stage monitoring. In this stage, we collect more comprehensive information (such as call stacks, parameters and return values) for all system calls. The heavy-weight
monitoring would not induce significant impact to the system, because the targeted processes have little or no activity, thanks to the first stage screening.

### 4.4.2 Process Constraining: procZip

We define the “constrained” state of an idling process as the following: (1) The process is allowed to continue its execution without any interruption, as long as it behaves consistently with its previously observed idling behavior; (2) When the behavior of the process deviates from its idling behavior, its execution is interrupted, and human interventions are needed to either allow the process to continue execution, or be terminated. Compared with simplistic mitigations, such as directly terminating an idling process, process constraining is a much more practical approach to reduce attack surface, because it leaves room for recourse in case of useful processes being mistakenly classified, and avoids serious negative impact on availability and user experience.

Different system mechanisms, such as *memory protection* and *code instrumentation* which build a program state machine [21,22], can be leveraged to achieve process constraining to different degrees. Developing a precise and efficient program behavior model is not the focus of our study. In this work, we simply choose to design a simple model based on *system call instrumentation* only to illustrate that idling processes can be easily constrained. In particular, we intercept all system calls of the target idling process, and build its “idling behavior” profile, which comprises of the variety of system calls as well as their *calling context (CC)* [102].

When the target process tries to make a system call that does not match the “idling behavior” profile, this incident is captured and corresponding manual intervention routine (*e.g.*, a pop-up window) is triggered. Because CC is a rich source of process internal state information, this approach could establish precise behavior constraining. Meanwhile, although such detailed system call instrumentation looks heavy weight, since the idling processes are generating little to no system calls, we
do not observe any noticeable system slowdown in practice.

For category I processes, because of their complete idleness, the constraining will be in effect immediately. In other words, any system call from these process will be considered as anomaly and thus will trigger manual intervention. For category II processes, a “profile building” phase, which lasts one repeating cycle, will be carried out before the constraining become effective. During the “profile building” phase, any system call and its CC will be recorded as allowed “idling behavior”. Afterwards, if a system call is detected with unmatched system call and/or CC in the “idling profile”, manual intervention is triggered.

4.5 Implementation

In this section, we first provide an overview of our light-weight data collection system, which logs and collects process activities. Then, we describe notable implementation details of the process constraining mechanism.

4.5.1 Data Collection System

We implement and deploy a client-server system to achieve ubiquitous and light-weight data collection. The infrastructure being used for process events collection has a three-tier architecture, i.e., the monitoring agent, the backend storage, and the analysis engine. Figure 4.4 illustrates the system architecture. The agent deployed on each host is responsible for performing both stages of data collection, as described in Section 4.4. For the first stage, the agent monitors a subset of system calls of every process on each host using two major data sources. One is the system call information provided by the kernel built-in Linux Audit subsystem [103]. The other is the auxiliary information from the proc file system. Because the majority of the monitoring overhead comes from the processing of data streams coming from the Linux Audit subsystem, we build a custom audit log handler, which achieves
14.3× speedup compared with the stock audit library. For the second stage, the agent leverages the standard \texttt{strace} utility to perform the monitoring of all system calls for selected processes.

\begin{figure}[h]
\centering
\includegraphics[width=0.7\textwidth]{system_architecture.png}
\caption{System Architecture}
\end{figure}

From real system deployment, we observe that the agent introduces unnoticeable latencies to commonly used programs (\textit{e.g.}, bash, vim, Firefox, Thunderbird, \textit{etc.}). The average resource consumption of the agent is negligible under today’s mainstream system configurations. On an idle system, the agent processes about 30 system calls per second, and consumes under 1.0\% of one processor core, and less than 100MB of memory. With intense workload, such as Linux kernel compilation, the agent processes about 5800 system calls per second, and use up to 14.8\% of one processor core and 300MB of memory. For data reporting over the network, each agent on average consumes under 13kbps of network bandwidth.

\subsection*{4.5.2 Process Constraining}

To implement the procZip, we leverage the kernel \texttt{ptrace} facility to intercept all system calls of the constrained process. For each intercepted system call, we uncover its calling context (CC) by walking up the call stack using \texttt{libunwind}, and concatenate each call site address into a chain. During the “profile building” phase,
we store all observed system call and corresponding CC in an associative array of CC-syscall pairs, which is used as the “idling profile”.

When a constrained process is detected to attempt “out-of-profile” system calls, we implement a pop-up dialog window to prompt the local user for either approval (i.e., allowing process to continue out-of-profile behavior) or termination. For our evaluation purpose, we also implement an alternative routine, which logs the event while silently allow the out-of-profile behavior to continue. This implementation enables us to non-intrusively perform evaluation on real server systems and workstations.

Of course, procZip will consume some system resources like CPU and memory on top of that of the idling process. Based on our experiment, such resource consumption is negligible.

4.6 Evaluation

Data Collection. The lightweight agent data was collected from a total of 64 Linux hosts. The data used for analysis was collected from December 2013 to July 2014, a period of over 180 days. The average days for each host is about 155 days during which each machine has rebooted at least once (due to power outage and so forth). Interestingly, most of these machines were never turned off by human even for desktop machines (this phenomenon is also observed in other environment [104]). Out of 64 hosts, we further investigate 24 hosts to study in more details about the process behaviors.

Determining Attack Surface. After determining idling processes, we prioritize high-impact attack surface cases (that are more likely to be exploited) based on the resource types and the process user ID. Similar as many existing works [17–19, 105–107], we define the attack surface as a set of communication channels (resources) provided by OS, including files, Internet sockets, and Unix domain sockets, through which an attacker can potentially gain additional privilege.
We summarize what kind of resources we consider and how we determine if they are attack surface as a set of rules: (1) If a process has opened a resource in readable mode, and (2) we consider three main resource types: Internet listening sockets (either bind to local or public interface), Unix domain listening sockets, and regular files. (2.1) If it is an Internet listening socket, we directly consider it attack surface as it can be accessed by anyone. Many known vulnerabilities are exploited by first connecting to a listening socket [107]. (2.2) If it is a Unix domain listening socket, we first check if it is file-backed. If so, the file permissions apply when connecting to such a socket, and we treat it similar to a regular file (as described next). Otherwise, if it is not file-backed, we consider it attack surface since anyone local can access it. Vulnerabilities have been reported when accessing Unix domain sockets [105]. (2.3) If it is a file, and the file is not a directory or a special file such as /dev/null and files under /proc. We consider it attack surface if one of the three following conditions is satisfied: (2.3.1) Everyone can write to the file. (2.3.2) The file owner group ID is different from the process group ID and the group-writable permission bit is set. (2.3.3) The file owner user ID is different from the process user ID and the file owner user ID is not root (as we assume the root is trusted). Vulnerabilities related to files have also been widely discussed [17, 106].

We rank the severity of the attack surface in the following order:

1. A root process with an open public network port.
2. Open files that are writable by anyone or a root process with local listening sockets accessible to everyone.
3. Other attack surface (see above).

**Evaluation Methodology.** We present the following results in the evaluation:

- How many processes are found by our LIPdetect technique as idling? What are their associated security risks (*i.e.*, attack surface)? Results are discussed in Section 4.6.1 and Section 4.6.2.
What percentage of discovered idling processes can be easily “quarantined” to reduce attack surface? Results are discussed in Section 4.6.3.

Can these idling processes be safely disabled in the eyes of system administrators and users? What are the reasons for their existence? Results are discussed in Section 4.6.4.

Can our findings be shared across different organizations? Results are presented in Section 4.6.5.

For the first result, we use the lightweight agent data to first gather a list of potential idling processes on 64 hosts. Next, we further select 24 hosts to deploy the second-stage monitoring to perform the end-to-end analysis.

For the second result, we deploy procZip to constrain the execution of the idling processes for two weeks and log any deviation attempts or new behaviors.

For the third and fourth result, we perform case study in collaboration with sysadmins and users in the enterprise and a survey with sysadmins in a university department.

It should be pointed out that the system administrators are mainly involved for evaluation purpose. In the operational workflow of our approach, system administrators are only involved (as the last step shown in Figure 4.2) when one really wants to kill a process or uninstall a program. There is no better way because it is extremely difficult to predict whether an idling process may become useful in the future. Therefore, it has to be human to judge the reasons case by case and determine whether a process can be safely killed or a program uninstalled. Our detection system itself (including data collection, idling process analysis, attack surface measurement) is automated and does not require any human intervention. To avoid confusion, we make a summary in Table 4.2 to differentiate human involvement in the operational workflow and non-operational (evaluation) workflow.
<table>
<thead>
<tr>
<th>Workflow</th>
<th>Role of Human</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operational</td>
<td>Determine safe-to-disable processes</td>
</tr>
<tr>
<td>Non-operational</td>
<td>Tag hosts and processes (Section 4.6.1)</td>
</tr>
<tr>
<td></td>
<td>Search for historical vulnerabilities of idling processes (Section 4.6.2)</td>
</tr>
<tr>
<td></td>
<td>Examine why some idling processes are not constrainable (Section 4.6.3)</td>
</tr>
<tr>
<td></td>
<td>Perform case studies to investigate the reasons of the existence of idling processes (Section 4.6.4)</td>
</tr>
<tr>
<td></td>
<td>Conduct survey to cross validate our results (Section 4.6.5)</td>
</tr>
</tbody>
</table>

Table 4.2. Clarification of human involvement in the operational workflow and non-operational (evaluation) workflow

### 4.6.1 Idling Processes Summary

First, by analyzing the lightweight agent data, we get totally 5839 long-running processes, among which 2095 were identified as potentially idling. Here, we consider a process long-running if it has been running for longer than 80% of the host uptime. Next, due to privacy and policy concerns, we select 24 hosts to deploy the second-stage monitoring. With `strace` data from 24 hosts (1774 long-running processes), 434 out of the 546 potentially idling processes are determined to be really idling. Among them, Category I has 328 (overall 18.5%) and Category II has 115 (overall 6.5%).

**Unique Binaries.** Even though the number of idling processes is high, they come from only 92 unique programs (i.e., executable names). Some idling programs are commonly seen on many hosts, while some others only exist on a few hosts. Figure 4.5 plots the fraction of instances of each unique executable in a decreasing order. It is interesting to see that there is a long tail where about 42 binaries occur less than twice in the 24 hosts, indicating a wide variety of idling processes and services. While it would take some time to investigate all of them for system administrators, one can prioritize which processes to look at first is based on how popular it is inside the enterprise. Table 4.3 lists the top ranked programs, the corresponding number of instances, and their categories.

**Idling Process Distribution.** We further study the distribution of idling
Table 4.3. Number of unique instances of idling processes and the corresponding category

<table>
<thead>
<tr>
<th>Category I</th>
<th>Instances</th>
<th>Category II</th>
<th>Instances</th>
</tr>
</thead>
<tbody>
<tr>
<td>acpid</td>
<td>19</td>
<td>atd</td>
<td>16</td>
</tr>
<tr>
<td>rpc.idmapd</td>
<td>13</td>
<td>rpcbind</td>
<td>5</td>
</tr>
<tr>
<td>hald</td>
<td>11</td>
<td>sendmail</td>
<td>5</td>
</tr>
<tr>
<td>avahi-daemon</td>
<td>11</td>
<td>mdadm</td>
<td>4</td>
</tr>
<tr>
<td>gconf-helper</td>
<td>10</td>
<td>smartd</td>
<td>4</td>
</tr>
</tbody>
</table>

Figure 4.5. Distributions for number of instances of unique executables

Intuition tells us that the characteristics of an idling process should also affect
its distribution. For instance, the idling processes on a server that is mostly accessed through remote shells might have more programs without GUI; while for a desktop host, GUI processes would instead take a larger portion. Due to the lack of a standard way to classify GUI and non-GUI programs, we manually tag each process. Figure 4.6(b) shows that the “User” machines do have more idling GUI processes than those “Server” machines.

![Figure 4.6](image_url)

**Figure 4.6.** Distributions of idling processes in different aspects. (a) OS purposes: User/Server/Testbed; (b) Processes types: GUI/non-GUI

**Syscall Patterns.** For category I processes, as shown in Figure 4.7, we observe a variety of blocking system calls. We can see that most of them are `select`, `read`, and `ppoll` which are operated on file descriptors. Such system calls have a parameter where a timeout value can be specified. We are surprised to see many programs choose to block indefinitely, given that such programming practice may lead to responsiveness issues [108].

A non-negligible fraction (16.3%) of them are not blocked on file descriptors, *e.g.*, `wait4`, `waitid`, and `futex`. At first glance, no attack surface seems to exist. However, it is plausible that after the syscall returns, the program will start taking input (*e.g.*, by calling `open`), which may constitute a attack surface.

For category II processes, out of 115, 49 repeatedly try to take input through...
Figure 4.7. Breakdown of system calls that Category I processes are blocked on

system calls like `read` and `recv`, but always failed with errors (e.g., timeout). 18 processes that successfully take input, yet do not produce any output (e.g., no `write` syscall). When the process does produce output, the output is most likely to be exactly the same across time. We measured the entropy [109] of the parameters for the output relevant system calls (e.g., `write`, `writev`, `recv`, `recvmsg`, `recvfrom`) and found that among the 32 processes that produce outputs, 14 have the same exact output, indicating zero entropy.

### 4.6.2 Attack Surface and Vulnerabilities

**Attack Surface.** Idling processes not only waste system resources but may also expose attack surface. Here, we study what is the percentage of idling processes which expose attack surface and what kind they are. Note that the attack surface we measure is only a lower bound, as the attack surface is measured only when a process is idling. Once become active, it may start accessing additional resources.
and open up new attack surface.

With that in mind, we still want to understand the attack surface exposed when these processes are idling. Overall, 127 of 434 (29.3%) processes have direct attack surface through files that are owned by a different user, or listening sockets which can be either accessed by all processes on the same host (if local socket) or by other hosts (if listening on the public interface).

Further, we find that the attack surface of the idling processes consists of 66.7% of attack surface exposed by all long-running processes, based on a snapshot of all the currently open resources on each host. Table 4.4 provides a breakdown of the attack surface. As we can see, the majority of the attack surface comes from the listening sockets (both remote-facing and local-facing). It is surprising to see that there are 94 public open TCP/UDP ports (without one connecting to them) from just 24 hosts, which raises significant security concerns, especially considering 43 of them are opened by root processes. Note that these processes are idling and no one has connected to these ports for a long time. For the listening local sockets, they are either TCP/UDP ports bound to loopback address or UNIX domain sockets. Such listening sockets present a threat on local privilege escalation attacks, which is less concerning but still important given the popularity of local privilege escalation attacks in Linux [17, 19]. Note that the number of file attack surface is smaller, indicating the more popular use of sockets as communication mechanisms in long-running processes.

Known Vulnerabilities. Out of all the processes, based on their executable names, we find that 47 out of 92 (51.1%) unique binaries have had known vulnerabilities in their development history according to the CVE database [110] and other online sources [111]. It indicates that there can be real threats to the idling processes that we discover. Just to give one example, remote attackers can trigger a vulnerability (CVE-2009-1490) in sendmail by crafting a long X-header to make a buffer overflow, which allows the attackers to execute shellcode and gain root privilege.
<table>
<thead>
<tr>
<th>Resource</th>
<th>File</th>
<th>Listening sockets (public)</th>
<th>Listening sockets (local)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Root</td>
<td>10</td>
<td>43</td>
<td>66</td>
</tr>
<tr>
<td>Non-root</td>
<td>19</td>
<td>51</td>
<td>53</td>
</tr>
</tbody>
</table>

Table 4.4. Attack surface breakdown of 434 idling processes

4.6.3 Constrainability Evaluation

Next we want to evaluate how constrainable the idling processes are. Through deploying procZip for two weeks on the 434 idling processes on 24 hosts, we find 384 processes (88.5%) did not have any “new behavior” or deviation from the previously observed behaviors – with a new syscall or an existing syscall at a different call stack, which means that they are very well constrained. 96.3% processes have less than 10 times where new behaviors are observed. More than 93.5% processes converge within one day.

Figure 4.8 shows the distributions of the number of times that new behaviors appear and the converge time across the 434 processes. The results give a strong indication that the majority of idling processes are indeed constrainable. Of course, ideally, if a process is really idling and the procZip is given the correct repeating cycle, they should be 100% constrainable and should not have any new behaviors observed. In reality, however, why are there still a few idling processes not constrainable? To answer this question, we further investigated 13 such processes from 10 hosts and found that they can be attributed to the following reasons:

First, 3 cases we find are due to the fact that some system calls are issued inside signal handler which may cause new call stack to be observed, depending on which instruction is interrupted. Note that this is not a fundamental limitation of our approach. We could support the signal handling cases by interpreting the parameters of signal handler registration calls (e.g., `signal`, `sigaction`), which allows us to know the address of the handler function. In that case, when we observe a new call stack in the future and see the address of handler function in
one of the stack frame, we can simply ignore the immediate stack frame next to it of which the return address can be anywhere in the program.

Second, due to the inherent noise tolerance characteristics of autocorrelation, 3 cases are detected with a repeating cycle shorter than the reality. These 3 cases are considered the inaccuracy of our idling process detection algorithm.

Third, 7 cases are due to real workload change which breaks its previous idling pattern. These 7 cases are considered the false positives of our approach. On one hand, we can argue that such cases are less likely to occur if we monitor their historical behavior long enough. On the other hand, we may want to admit that it is fundamentally hard by any means to predict the future.
4.6.4 Safe-to-disable Processes and Case Studies

Many idling processes are considered safe-to-disable in the eyes of system administrators and owners. We conduct detailed case studies in the enterprise to identify safe-to-disable processes and the reasons of their existence. The key challenge to perform the study is that a process needs to be examined in the typical runtime context (job-mix) of the host (or cluster of hosts) under examination. To achieve this, we have had intensive conversation with the system administrators and owners of the machines. Finally, we gather a list of processes that are considered safe-to-disable and we provide analysis and insights about such processes. Hopefully, our investigation results can serve as a helpful reference to average system administrators and end users in general.

In our case studies, 133 out of the 434 processes are considered safe-to-disable (30.6%). The top reasons are listed below (without specific order):

- **Supporting unused hardware devices**, e.g., bluetooth devices, 3G/4G data card. Given that we are in an enterprise environment with fairly standard hardware setup: USB keyboard/mouse and Ethernet cables, such processes are very unlikely to be used. After checking with our system administrators, we conclude that they do not actually support the use of bluetooth devices or 3G/4G data cards and therefore these processes can be disabled. As a more detailed example, `mdadm` is a Linux utility used to manage software RAID devices. In strace log, the `mdadm` process is repeatedly calling `dup`, `fcntl`, and `close` on a configuration file `/proc/mdstat`, while this file shows that no RAID has been configured. We further check the file system and realize that there is only one physical volume, which does not require RAID management. Interestingly, the reason that the process is running is that the host were previously used to manage RAID devices. When the RAID devices were removed, the person forgot to terminate this service. Besides this example, most of these processes are actually bundled in the default
installation image of the Linux distributions.

- **GUI-related processes that run on server machines**, *e.g.*, notification of sound level change. The servers in our environment are never intended to be used as desktop hosts. As a result, many GUI-related processes can be disabled. For instance, strace logs show that the *indicator-sound-service* process is completely blocked on a system call on three machines and never return. Actually we find out that they sit in a server room where no one is using it to output any audio, let alone observing the sound level changes.

- **Supporting unused miscellaneous software services**, *e.g.*, virtualization daemon (no VMs are supposed to run on them), super server which is not configured to start any daemon on demand. We find that some machines are pre-installed with the *xinetd* (extended Internet daemon) which listens for incoming requests over a network and launches the appropriate service for that request. By examining this process in detail, we realize that the configuration file indicates that there is no service registered at all, which indicates that this process is not intended to run. This is also confirmed with system administrators.

- **Applications not properly setup/configured**, *e.g.*, location tracking service, web servers serving default page. Many programs, when installed, are not fully configured to run with intended behaviors, possibly due to human negligence or mistakes. For example, we see an instance where the web server *thttpd* is installed and running for a long time yet idling. We further check the web directory and find that it only contains the default pages that come with the default installation. In other words, one can only browse the default page (which is a 404 page). Another example is *sendmail* which is normally used as mail servers and mail relays. Desktop users, in our environment, do not really need this program as our email systems go through the company mail
servers (Microsoft exchange server). We typically use IT-supported programs such as Thunderbird to receive emails.

Although rare, another interesting case is that certain application/service is in the process of being retired and its functionalities are to be replaced by a new one. In this case, we observe that the old application/service can still be running for some reason. For example, HAL (Hardware Abstraction Layer) was aimed for providing hardware abstractions for UNIX-like OSes. Since 2011, major Linux distributions are in the process of deprecating HAL and hald is in the process of being merged into udev and Linux kernel. Permitted by the owner, we disable hald on one desktop machine and the user did not notice any influence. The CD and USB icon can still automatically appear on the desktop when being inserted. The discovery of this type of case is very surprising and not originally anticipated by us.

Overall, a large portion of safe-to-disable processes can be attributed to the fact that OSes are packing more and more general purpose applications and utilities. Some of them are installed at a later time, yet they are either not really properly configured to begin with, or they are no longer needed but forgotten to be uninstalled.

**Boundary Cases.** There exist a large number of processes that are in the grey area even in the eyes of system administrators. Here are some examples to illustrate such cases: the irqbalance process is a daemon process that balances interrupts across multiple CPUs or cores, which is supposed to provide better performance and IO balance on SMP systems. However, we find an intensive and insightful discussion among the Ubuntu package management people in their online forum about whether to start this process by default [112]. Our survey indicates that irqbalance seems to be targeted at a specific server environment and may sometimes downgrade the performance. Further, on desktop machines, such performance concern is barely serious.

To more objectively evaluate whether the processes are indeed safe to disable.
We have asked a few users who have agreed to have a few services disabled, which include: blueoothhd, mdadm, indicator-sound-service, xinetd, thttpd, irqbalance, sendmail, smartd, hald, and winbindd over the course of a few days. The users do not observe any disruption to the daily use of the machines.

4.6.5 Cross Validation

Since server systems usually share some common characteristics, we extend our evaluation to another organization to see whether our results can also be shared. Particularly, we conduct a survey with three system administrators in a university department. One of them is the head of the IT group and the other two are both senior system administrators with multiple years of experience. The IT group is responsible for managing computers in the classrooms, faculty offices, and student laboratories. They are also in charge of the department web server, email server, databases, etc.

We create a questionnaire containing 20 programs selected from our safe-to-disable list together with the specific reasons of their existence. We ask them to 1) give a score from 0-10 (0-strongly disagree, 10-strongly agree) to indicate to what extent they can confirm our result based on their own experience, and 2) note down whether they happen to find the exact same program on their machines and also think it is unnecessary but somewhat were not aware of before.

The survey result is shown in Table 4.5. Overall, the average score is 9.6, showing a reasonably large degree of agreement. Two sysadmins even state that they find all the 20 programs on their machines that they believe are unnecessary but did not notice before. The result differs among individual sysadmins. Sysadmin 1 seems to fully agree with our results, while sysadmin 3 have many different opinions and observations. This is reasonable because they manage different set of machines that may have different environments, requirements, and preconditions. In general, we conclude that our findings can be potentially shared across organizations to help
system administrators remove unused programs/services to reduce attack surface.

4.7 Discussion

Anomaly Detection and Process Behavior Models. Numerous attempts have been made to study how to model the behavior of a process and learn its normal behavior for the purpose of performing anomaly detection [20–23]. Even though our study also models the behavior of a process, it incurs a different set of challenges. Specifically, we examine if existing models can directly be applied in our case. In general, there are three categories of process behavior models: 1) sequence of system calls [20, 64], 2) deterministic Finite State Automata (FSA) [21, 22], and 3) probabilistic FSA (e.g., Hidden Markov Model) [23, 65]. For 1), the idea is to use n-gram model to learn the common n consecutive system calls. The problem is that an idling process needs not only to have common n-grams, but also the exact time alignment of events. For 2) and 3), the model only learns what the next system call is allowed based on the current “state”, it does not learn the linkage between a long sequence of events. For instance, if the model has two states A and B, with the possible transition of $A \rightarrow B$, $B \rightarrow A$, and $A \rightarrow A$. It is unclear if this leads to a repeating sequence at all. For example, the allowed sequence could be either $A \rightarrow B \rightarrow A \rightarrow A \rightarrow B \rightarrow \ldots$ (non-repeating) or $A \rightarrow A \rightarrow B \rightarrow A \rightarrow A \rightarrow B \rightarrow \ldots$ (repeating). Furthermore, such models do not consider time information at all. Our time-augmented version of autocorrelation approach fits well the problem of identifying repeating patterns.

Idling Processes and Their Necessity. The concepts of “idling” and “unnecessary” are not exactly the same thing. An idling process describes a process that is in a state given past observations. Whether or not it is still necessary to keep in the system is a question that requires future knowledge. For instance, an idling daemon process that supports cellular network cards may be idling for the past one year. However, it is impossible to predict if someone may need to use the
<table>
<thead>
<tr>
<th>Program Name</th>
<th>Reason</th>
<th>Sysadmin 1</th>
<th>Sysadmin 2</th>
<th>Sysadmin 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Score</td>
<td>Same Finding?</td>
<td>Score</td>
</tr>
<tr>
<td>acpid</td>
<td>Handling events like pressing the power button or closing the lid. Useful for desktop especially laptop machine, but not useful for servers.</td>
<td>10</td>
<td>Y</td>
<td>10</td>
</tr>
<tr>
<td>avahi-daemon</td>
<td>Used to discover network services. We might not use it here, but some enterprises might use it to discover printers.</td>
<td>10</td>
<td>Y</td>
<td>10</td>
</tr>
<tr>
<td>bluetoothd</td>
<td>Most desktops do not use bluetooth in the company.</td>
<td>10</td>
<td>Y</td>
<td>10</td>
</tr>
<tr>
<td>cupsd</td>
<td>Used to connect to printers. Server machines may not be used to print things directly.</td>
<td>10</td>
<td>Y</td>
<td>10</td>
</tr>
<tr>
<td>dnsmsq</td>
<td>DNS Caching daemon. Not used in our configuration.</td>
<td>10</td>
<td>Y</td>
<td>10</td>
</tr>
<tr>
<td>gconf-helper</td>
<td>Sound server pulseradio’s helper. Unnecessary for servers, necessary for desktops</td>
<td>10</td>
<td>Y</td>
<td>10</td>
</tr>
<tr>
<td>gdmgreeter</td>
<td>Providing welcome screen. Server machines may not need this.</td>
<td>10</td>
<td>Y</td>
<td>10</td>
</tr>
<tr>
<td>gechoe-master</td>
<td>Location tracking software.</td>
<td>10</td>
<td>Y</td>
<td>10</td>
</tr>
<tr>
<td>hidd</td>
<td>Bluetooth keyboard and mouse support (no one seems to be using it)</td>
<td>10</td>
<td>Y</td>
<td>10</td>
</tr>
<tr>
<td>indicator-sound-service</td>
<td>GUI-related. Shows the sound level. Not useful on servers.</td>
<td>10</td>
<td>Y</td>
<td>10</td>
</tr>
<tr>
<td>iscsid</td>
<td>SCSI = small computer systems interface, e.g., used to connect network disks. There is no networked disks connected from these machines (strace shows that).</td>
<td>10</td>
<td>Y</td>
<td>(a)</td>
</tr>
<tr>
<td>mdadm</td>
<td>This is used to manage software RAID devices, however, there are no RAID configuration on these hosts.</td>
<td>10</td>
<td>Y</td>
<td>10</td>
</tr>
<tr>
<td>modem-manager</td>
<td>ModemManager is a DBus-activated daemon which controls mobile broadband (2G/3G/4G) devices and connections.</td>
<td>10</td>
<td>Y</td>
<td>10</td>
</tr>
<tr>
<td>pcsd</td>
<td>PC/SC Smart Card Daemon. Should be unnecessary for servers.</td>
<td>10</td>
<td>Y</td>
<td>10</td>
</tr>
<tr>
<td>sendmail</td>
<td>Used as email server or relay. Not useful on average desktop machines.</td>
<td>10</td>
<td>Y</td>
<td>10</td>
</tr>
<tr>
<td>smartd</td>
<td>Monitor the reliability of the hard drive and predict drive failures. Not very useful.</td>
<td>10</td>
<td>Y</td>
<td>10</td>
</tr>
<tr>
<td>telepathy-indicator</td>
<td>Most people might not need it for telephony</td>
<td>10</td>
<td>Y</td>
<td>10</td>
</tr>
<tr>
<td>winbindd</td>
<td>Provides mapping between Windows user/group SIDs and unix user/group IDs. Not useful when only replying on unix user/group IDs to manage the user account.</td>
<td>10</td>
<td>Y</td>
<td>10</td>
</tr>
<tr>
<td>xinetd</td>
<td>Super server (e.g., starts apache on demand).</td>
<td>10</td>
<td>Y</td>
<td>10</td>
</tr>
<tr>
<td>zeitgeist-daemon</td>
<td>Logging user behaviors in a central database. But strace shows that there’s no real logging. In other words, it is not configured properly.</td>
<td>10</td>
<td>Y</td>
<td>10</td>
</tr>
</tbody>
</table>

(a) 10 - if not connecting to iscsi volumes; 1 - if connecting to iscsi volumes.  
(b) No – it may be useful to send logs to a central logging service.

Table 4.5. Survey results from three system administrators in a university department.
cellular network card in the next day. Therefore, the necessity of a process can only be answered by prophet, in our case, the system administrators.

In addition, the concept of “necessary” is subjective. A process that is deemed unnecessary may be actively performing useful operations. Consider a process which outputs a pop-up notification window (e.g., for instant messenger applications) to the screen from time to time. Some user may find it useful while others will not and may even find it disturbing. Such problem is outside the scope of this chapter.

**Idling Process and Its Usage.** In this chapter, we study a number of idling processes from our automated method and then confirm that a large fraction of them can be safely disabled or uninstalled. We envision that such knowledge can be shared across enterprises or organizations. In Section 4.6.4, we notice that every safe-to-disable process has its reason to be idling. We could actually construct a knowledge base of such safe-to-disable processes. One way to share such knowledge is to compile a list of questions such as whether the organization requires the usage of special hardware (e.g., cellular network connection) or whether the host is to be used as a server or desktop. Answering these questions will then lead to an automated selection of services to install in order to minimize the attack surface while at the same time satisfy the functionality requirement.

**4.8 Summary**

In this chapter, we present LIPzip, a system that identifies idling services, an under-studied research area, and explore its security application of attack surface reduction. We define idling process as long-running processes that are in a blocked or bookkeeping state, and we propose an adaptive autocorrelation-based algorithm which can robustly detect idling process from system call information. We evaluate the effectiveness of our algorithm in the server system of a real enterprise, using a custom built light-weight and scalable data collection system. With both case studies and automatic validation through process behavior constraining, 30.7% of
the identified idling processes can be safely disabled to fully eliminate their attacker surface, and 93.5% can be safely and automatically constrained to reduce attack surface without system administrators’ manual intervention.
Chapter 5
Risk Assessment of Buffer “Heartbleed” Over-read Vulnerabilities

5.1 Introduction

The buffer over-read vulnerability [25] has gained much attention after the Heartbleed [6] bug was discovered, which threatens millions of Web services on the Internet [5]. A buffer over-read happens when a program overruns a buffer’s boundary and reads the adjacent memory. It is similar to buffer overflow which is an over-write, but had been paid less attention than over-write as it usually does not lead to memory taint or control hijack. However, sensitive and security related information such as passwords and keys can be leaked through buffer over-read.

The root cause of buffer over-read is similar to that of buffer overflow. It is typically resulted from insufficient or a lack of bound checking for buffer read (write for overflow). Extensive research has been devoted to this issue related to buffer overflow and practical tools exist for partial mitigation. For example, StackGuard [113] injects a canary right after the return address to detect stack-based buffer overflow before a function returns. Address Space Layout Randomization (ASLR) [114,115] makes the addresses of the target code such as standard libc less predictable and thus increases the difficulty of control transfer hijacking when a
buffer is overflowed. These techniques have been implemented in many compilers and systems, but they are not applicable to the over-read issue. There is also a large volume of research on direct bound checking (e.g., [116–118]) and safe type system retrofitting (e.g., [119,120]). Bounds checking and safe type system retrofitting can mitigate the over-read issue, but few is widely adopted due to either excessive runtime overhead, expensive cost of manual work, or insufficient mitigation in practice.

A number of programming techniques for writing solid and secure code have been around to preserve memory safety. Although some of them are initially intended to make the resulted software more reliable, they are helpful in mitigating the buffer over-read vulnerability. Zero out memory blocks or buffers have been advocated and implemented in many systems. In particular, Chow et al. [26] measured the performance overhead on zero out heap buffers and stack frames at deallocation time. They also experimented on zeroing out currently unused stack part periodically. Initializing and padding buffers with special characters are proposed to facilitate easier debugging and fault localization. Maguire [121] uses “0xA3” and “0xCC” for padding and initialization of buffers for Macintosh models and Microsoft applications, respectively, due to a number of reliability and debugging benefits.

These techniques are useful in mitigating buffer over-read vulnerabilities or information leak in general; there is, however, no quantitative study on their effectiveness with regard to information leak through buffer over-reads. Some of these techniques are known in the field, but were not reported with any quantitative measures before to the best of our knowledge. Moreover, one could view “Heartbleed” as a family of buffer over-read vulnerabilities and there could be unknown zero-day vulnerabilities exploitable by unknown “Heartbleed” attacks. It is thus too restricted and not objective to evaluate the entire risk of buffer over-read vulnerabilities solely based on the specific Heartbleed bug. As an emerging type of vulnerability, we need more research on the mitigation of the buffer over-read vulnerability as well.
as the quantitative risk assessment of the software systems deployed in the field.

In this chapter, we intend to fill in the gap by providing a preliminary quantitative measurement on the amount of information leak associated with buffer over-read for some popular mitigation methods such as zeroing and padding buffers. We explore and implement both in-line and concurrent versions of these methods. We apply these techniques to the RUBiS benchmark [31], and report our experience on four metrics we developed for measuring information leak risks.

In summary, our main contributions are:

- We report the first experience on quantitative measurement of potential amount of information leak through buffer over-read. The methodology can also be applied to other programs for a quick risk assessment.

- We provide a summary of the current mitigation techniques that are applicable to buffer over-read and measure their effectiveness.

- The preliminary quantitative risk measurement and experience we reported can facilitate further studies on the issue.

The remainder of the chapter is organized as follows. In Section 5.2, we introduce background, threat models, motivation, and examples. In Section 5.3, we describe the details of our methods and present the metrics on information leak measurement. We validate our methods with a few case studies in Section 5.4 and present the experimental results in Section 5.5. We discuss some other issues and limitations in Section 5.6 and summary in Section 5.7.

5.2 Overview

5.2.1 Background

Although heap buffer overflow has been extensively researched for many years, heap buffer over-read remains under-studied and under-reported [122]. Our focus is
the risk assessment of heap buffer over-read vulnerabilities. Conceptually, a buffer over-read attack involves a source buffer, a destination buffer, and the vulnerable operation(s) that are possibly resulted from a bug in the program. Therefore, the victim of over-read is the source buffer rather than the destination buffer, whereas the victim is typically the destination buffer in a buffer overflow attack.

We classify heap buffer over-read into three categories.

1. **memcpy-based.** This is a major type of buffer over-read vulnerabilities and possibly the most damage-causing (due to Heartbleed) one. This happens when the `size` argument of a `memcpy` function is accidentally or maliciously enlarged so that more information than necessary or allowed is copied to the destination. As this is the main focus of this chapter, further details will be introduced shortly in the next section. Real world examples of this type of vulnerability include CVE-2014-0160 (i.e., Heartbleed) and CVE-2009-2523.

2. **strcpy-based.** This is another common vulnerability in which unintended extra information is copied out (e.g., CVE-2009-2523). Different from `memcpy`-based over-read, `strcpy`-based over-read is mostly due to the improper null termination of the source string. Although there have been a handful of studies on `strcpy`-based over-read in terms of safe function alternatives [123], vulnerability detection [124], and writing secure program in general [78], a real world vulnerability leading to large-scale information leak does not exist yet and thus the corresponding thread model is still unclear. Therefore, in this report, we exclude this category from consideration.

3. **Byte-level over-read and others.** In addition to the category (1) and (2) that are “chunk-level” over-read, byte-level over-read vulnerabilities are also witnessed by the real world (e.g., CVE-2004-0112, CVE-2004-0184). Instead of reading out a chunk of memory as a whole piece, byte-level over-reads usually access one element (of an array or a table) at a time or loop over a
range of elements. Insufficient or even lack of bounds checking is always the root cause. Using an index that exceeds the lower limit and the upper limit will result in under-read and over-read, respectively. However, only given the program binary, it is highly non-trivial to identify the set of instructions with potential risks of being affected by such over-read. Even with the availability of source code, it is missing from existing literature on how to identify the potential culprits. More importantly, the amount of information leakage caused by byte-level over-read is usually minimal compared to category (1) and (2), so we decide to also omit this category in our risk assessment.

Therefore, this chapter only focuses on a particular type of heap buffer over-read: the \texttt{memcpy}-based buffer over-read. Although the scope is limited to some extent, we argue that this is the most attainable way to do risk assessment for heap buffer over-read vulnerabilities at the time of this writing, which we think is still in the early stage of buffer over-read research.

\subsection*{5.2.2 Threat Model}

We now explain the details of \texttt{memcpy}-based buffer over-read vulnerabilities. We will briefly introduce the Heartbleed vulnerability and use it as the illustrative example. For more details of the Heartbleed vulnerability, please refer to [6].

OpenSSL implements the TLS Heartbeat Extension [125], which allows the client to send a heartbeat request to the server, who will then reply the same payload back to keep the connection alive. Figure 5.1 shows a code snippet of \texttt{openssl-1.0.1f} for constructing heartbeat response message. In line 2586, \texttt{memcpy} function copies data of \texttt{payload} size from the source buffer \texttt{pl}, which contains the heartbeat request message, to the destination buffer \texttt{bp}. However, since both the \texttt{payload} variable and the \texttt{pl} buffer are controlled by the attacker, and there is no check to make sure that the actual length of the \texttt{pl} buffer is equal to \texttt{payload}, the attacker can provide a very short heartbeat request message while setting the
buffer = OPENSSL_malloc(1 + 2 + payload + padding);
bp = buffer;
/*Enter response type, length and copy payload*/
*bp++ = TLS1_HB_RESPONSE;
memcpy(bp, pl, payload);
bp += payload;
/* Random padding */
RAND_pseudo_bytes(bp, padding);

r = ssl3_write_bytes(s, TLS1_RT_HEARTBEAT, buffer, 3 + payload + padding);

**Figure 5.1.** A code snippet related to the Heartbleed vulnerability (from t1_lib.c in openssl-1.0.1f).

The `payload` variable is set to a large value up to $2^{16}$. Thus, the `memcpy` function will read beyond the boundary and copy sensitive data to the `bp` buffer, which is eventually sent to the attacker (line 2591).

**Figure 5.2.** Illustration of heap buffer over-read in the Heartbleed vulnerability.

Figure 5.2 further illustrates the information leak caused by the `memcpy`-based over-read. There are actually two types of information leak. The first type is to steal information previously used and left in the heap memory. For example, if the vulnerable buffer `pl` has been used to handle a previous user’s request, and the data...
in the buffer is not cleared or initialized (e.g., the uninitialized area in Figure 5.2), then `memcpy` will copy such data to the intermediate buffer which will be then sent to the attacker. Note that existing approaches on bounds checking [116–118] does not really handle this type of information leak, because they only check memory access at the end of the buffer. The second type is to steal information that are stored out of the current buffer. More specifically, if the over-read reaches a buffer that is located after the vulnerable buffer and the buffer is currently filled with sensitive data, then those data will be leaked also. Since the attacker utilizes both types of leakage to steal sensitive information, we will not separate them in later discussions.

5.2.3 Motivation and Challenges

To evaluate the risks of heap buffer over-read vulnerabilities associated with `memcpy`, our initial attempt was to capture the instances of `memcpy` that are exploitable by the attacker. If we can achieve this, the risk assessment will be accurate. In practice, however, we found it is extremely hard to do so. The main challenge is that there lacks a reliable method that can accurately pinpoint those vulnerable `memcpy`s.

Although a few commercial tools, such as Coverity [126] and CodeSonar [127], can now detect the Heartbleed bug, none of them did it before the bug was disclosed. Otherwise the Heartbleed bug might not have stayed there for that long, namely around two years. In this sense, it is reasonable to suspect that there could be more complicated and obscure zero-day buffer over-read vulnerabilities associated with some other `memcpy`s that are unknown. Thus, only using the Heartbleed vulnerability to assess the entire risk of buffer over-read vulnerabilities of a program is neither objective nor representative. If an attacker exploits a different bug to do buffer over-read, what the attacker can get might be very different from what he can get using Heartbleed. Moreover, it is fundamentally difficult to verify a
memcpy is never exploitable; and it is also hard to predict how soon an exploit will happen. Therefore, our intention is to come up with a generic methodology for buffer over-read vulnerability risk assessment before the next “Heartbleed” arrives. As the result, we choose a conservative principle for our risk assessment, that is, every memcpy could be abused by the attacker.

5.2.4 First Glance

To see how different it is to look at all memcpys instead of only considering the Heartbleed bug in regard of evaluating the risk, we first try to get some early sense. Since Apache and Nginx are the two major targets of Heartbleed attacks [5], we perform a preliminary analysis on the two Web servers.

<table>
<thead>
<tr>
<th>Program</th>
<th># memcpy</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>openssl-1.0.1f</td>
<td>184</td>
<td>50K</td>
</tr>
<tr>
<td>httpd-2.2.14</td>
<td>380</td>
<td>283K</td>
</tr>
<tr>
<td>nginx-1.3.9</td>
<td>156</td>
<td>122K</td>
</tr>
</tbody>
</table>

Table 5.1. Statistics of OpenSSL, Apache, and Nginx

First, we count the number of memcpys as well as lines of code in OpenSSL, Apache and Nginx, shown in Table 5.1. In OpenSSL, we already know that 184 memcpys out of 50K LOC lead to one Heartbleed vulnerability. Therefore, we probably can infer that there might be two heap buffer over-read vulnerabilities that are still unknown in Apache and another one hidden in Nginx.

Second, we further run the RUBiS benchmark (the detailed experiment setup will be introduced in Section 5.5) using Apache with OpenSSL to get a rough ratio of data leakage between the Heartbleed bug and all the other memcpys. Here, we run 16 user sessions for about one minute. We perform the Heartbleed attack on each user session by over-reading 32KB data for once, two times, three times, respectively. For all the other memcpys, we set the size of data over-read to be the original (intended) memory copy size. Figure 5.3 (a), (b), and (c) show the ratio of data leakage between Heartbleed and all memcpys from the three experiments. We
Figure 5.3. A rough estimate on the ratio of data leakage of Heartbleed and all memcpy when Heartbleed attack happens (a) one, (b) two, and (c) three times during a one-minute user session.

can see that the Heartbleed bug only contributes around 10% to 30% to all the potential data leakage.

As the result, given a technique targeting at heap buffer over-read vulnerabilities, it is insufficient to only evaluate the effectiveness against one particular known bug. Instead, one needs to consider all instances of memcpy to perform a comprehensive risk evaluation.

5.3 Risk Assessment

The risk assessment for the memcpy-based heap buffer over-read vulnerabilities boils down to two questions: (1) How to collect the potential heap buffer information leak? (2) How to quantify the potential heap buffer information leak? In this section, we describe our methodology to address these two questions.
5.3.1 How to collect the potential heap buffer information leak?

Since we consider every memcpy potentially vulnerable, we dynamically hook each occurrence of memcpy and simulate buffer over-read attacks by copying out additional \( N \) bytes (after the intended size bytes in the src buffer) and write to a log file. The \( N \) bytes are therefore regarded as information leak. As the size \( N \) is usually controlled by the attacker and hence hard to predict, we do not specify the exact value as part of our method. Nevertheless, there could be many heuristics and strategies on choosing the over-read size. For example, in Heartbleed attackers can read at most 64KB (i.e., \( 2^{16} \)) because the size argument is converted from a short integer of two bytes. Some heuristics can also be employed to estimate the upper limit of \( N \) used by a reasonable attacker, for instance, the maximum size of malloc request. In addition, one can also use a stochastic process (e.g., Gaussian process) to generate a sequence of variable over-read size. In sum, it is better for system administrators or security officers who adopt our method to determine the size \( N \) depending on the target platform and workload, such as the type of web server, the type of client requests, the characteristics of web pages being hosted (e.g., static or dynamic, page size, etc.). In our experiment, we select fixed size of 1KB, 16KB, and 32KB to simulate the buffer over-read attack and collect data leak.

The implementation is straightforward. We simply hook every memcpy using the LD_PRELOAD trick, calculate the range of victim buffer, and copy the data out and dump to a log file. The benefit of using LD_PRELOAD is that our risk assessment can directly handle off-the-shelf binaries without the need of source code. It should be pointed out that in real world programs the source buffer of memcpy can also be on stack and data segment. To remove these memcpys from our consideration, we check the virtual addresses of source buffers against the memory map /proc/self/maps and skip those memcpys. After running the target program with certain test inputs or benchmarks, the log file will contain the (simulated) leaked data.
5.3.2 How to quantify the potential heap buffer information leak?

The quantification of information leak is the top challenge of our risk assessment. We develop four metrics to quantify the information leakage in different aspects. In reality, since people may or may not know what data is targeted by an attacker, we come up with two assumptions: a weak assumption and a strong assumption. In the weak assumption, we assume we are uninformed of what data is targeted by the attacker. We propose Metric 1 and 2 below to perform a macroscopic measure on the gross information leakage. The strong assumption means that we are aware of the target of the attacker (e.g., private key, username-password pair). We propose Metric 3 and 4 below to conduct a fine-grained examination. Note that although we only demonstrate in Section 5.5 the application of these metrics using a particular benchmark, the metrics themselves are meant to be general. The four metrics are described as follows:

**Metric 1: Volume of Information-carrying Bytes.** Not every byte leaked to the attacker carries meaningful information. For example, if the victim buffer is filled with zeros, there is virtually no meaningful information being leaked. So we can use both the size and ratio of the information-carrying bytes to quantify how much gross information is leaked. How to differentiate meaningful bytes and meaningless bytes depends on the particular defense techniques. In our case studies (introduced shortly in Section 5.4), we pick a special ASCII control code $0x06$ (ACK) to be used in the defense techniques to clean the heap memory. For example, if a 1MB log file contains 700KB non-$0x06$ characters, the ratio of information carrying bytes would be 70%.

**Metric 2: Information Entropy (Compression Ratio).** In information theory, entropy is used to indicate the quantity of information. In the literature of quantitative information flow, various entropy measures have been explored, including Shannon entropy, guessing entropy, and min-entropy [75]. In our context,
since there is not a clear way to do word segmentation or text segmentation against the log file, we use compression ratio to approximate the information entropy of the leaked data. Particularly, we leverage three different compression algorithms (gzip, bzip2, and xz). The higher the compression ratio is, the higher entropy the leaked data contains.

**Metric 3: Sensitive Data Quantity.** Apparently, if we know the exact sensitive data (e.g., server private key) or we are able to identity sensitive data based on the patterns (e.g., email address), we can perform targeted analysis to quantify the amount of information leakage. In our experiment on the RUBiS benchmark, for instance, the user name and password of a user with ID 1234 will be “user1234” and “password1234”, respectively. As a result, we can search for such patterns from the log file and count the total number of valid username-password pairs.

**Metric 4: Unique Sensitive Data Quantity.** Since the data over-read by multiple memcpys could have overlaps, it is necessary to remove the redundancy and check the amount of unique sensitive data leaked. Different from Metric 3 which implies how big the chance is for an attacker to get something valuable, this metric reveals how much unique entities can be affected. The entities can be registered users, customers, and websites. In our experiment, for example, the number of leaked username-password pair indicates the number of users of the auction site that are affected.

### 5.4 Case Studies

To validate our risk assessment methodology, we perform case studies on four different sets of approaches adopted from existing literature, which are all aimed at mitigating information leakage in buffer over-read attacks.
5.4.1 Padding at Allocation

Assuming the same attack with the same length of over-read, one intuition to reduce the amount of information leakage is to increase the memory allocation size, in other words to pad the memory object with additional memory, and initialize the memory (including both original and padding) with zero bytes. Actually, similar padding ideas have already been proposed in DieHard [128] and OpenBSD to tolerate memory errors such as buffer overflows, dangling pointers, and reads of uninitialized data.

Ideally, if the padding bytes are longer enough, larger than the upper limit of over-read size used by attacker, the attacker can get nothing except his own memory content. This will lead to zero information leak. In reality, however, the upper limit of over-read size could be very large (e.g., 64KB in Heartbleed) and even not foreseeable in case of zero-day attacks. Too large padding size could cause dramatic increase in the memory consumption as well as performance overhead. Actually, the idea of padding bytes has been widely used in a variety of areas. There are different strategies to choose the padding size. Some techniques insert padding to achieve alignment by making the size to be a multiple of certain bytes or bits (e.g., 4 bytes in data structure padding by compiler, 8 bytes in libc malloc, 32 bits in IP packet etc.). Some other techniques pad each target to the same size to achieve anonymity and examples can be found in cryptography [129]. Sometimes, random padding is also used, for example, to defeat side-channel attacks [130]. We choose a simple heuristic, i.e., making the padding size to be the same as the original allocation size. The merit of doing so is the deterministic space overhead, that is, the heap will be at most as twice large as the original heap. To achieve this, we simply hook every malloc, double the requested size, perform real malloc, and zero out the allocated memory. The following code snippet demonstrates this procedure.

```c
void* malloc_hook(size_t size) {
    void *ret = malloc(size*2);
    /* Do other stuff here */
    return ret;
}
```

105
5.4.2 Erasing at Deallocation (Inline)

It has long been a secure programming guideline that sensitive data should be properly erased after the processing has done with it. This idea can also help to lower the chance and amount of information leakage. This idea has been explored in data lifetime research to defend against memory disclosure attacks [26]. We reimplement this technique for glibc heap memory management by hooking `free`, retrieving the chunk size, and then zeroing out the chunk. The code snippet shown below demonstrates this idea.

```c
void free_hook(void *ptr) {
    size_t size = malloc_usable_size(ptr);
    memset(ptr, 0, size);
    free(ptr);
}
```

5.4.3 Erasing at Deallocation (Concurrent)

We also attempt to reduce the performance overhead of the inline heap erasing by exploring a concurrent technique proposed in Cruiser [131]. As shown in Figure 5.4, the key idea is to migrate the heap erasing task to a concurrent eraser thread and leverage lock-free data structure to achieve non-blocking and efficient synchronization between user threads and the eraser thread. Since the design and implementation of this technique is not the focus of this chapter, we only provide a
brief overview of this technique as below.

![Diagram of concurrent heap buffer erase](image)

**Figure 5.4.** Architecture of concurrent heap buffer erase

We hook every call to `free`, push the target pointer to a queue, and return immediately. Meanwhile, we create a dedicated eraser thread to read pointers out of the queue, perform memory erasing, and then call the glibc `free` to do real deallocation. We implement the queue data structure based on the single-producer single-consumer FIFO lock-free ring buffer proposed by Lamport [132]. This data structure (not shown in Figure 5.4) enables a producer thread and a consumer thread to concurrently perform operations (i.e., enqueue and dequeue) on the ring buffer. The following pseudocode highlights the workflow of this technique.

```c
void free_hook(void *ptr) {
    if (ptr) {
        enqueue(ptr);
    }
}

void* eraser_thread() {
    while(!stop){
        void *ptr = dequeue();
```
memset(ptr, 0, malloc_usable_size(ptr));
free(ptr);
return NULL;
}

5.4.4 Concurrent Erasing plus Padding

Based on the concurrent heap erasing, we further combine it with the technique of padding at allocation. Theoretically, we simply need to integrate the two techniques together. In practice, we realize that all the newly allocated memory returned by malloc have already been erased by an earlier free. Therefore, it is unnecessary to zero out the memory again. So we only need to hook the malloc and do memory allocation with double request size.

5.5 Experimental Results

5.5.1 Experiment Setup

We base our experiment on the RUBiS [31] benchmark which is commonly used for evaluation by the systems research community, especially on emulating real world workload of websites that use dynamic content. The RUBiS benchmark is an auction site prototype modeled after eBay.com. It contains three types of user sessions, namely buyers, sellers, and visitors, as well as 26 types of interactions, such as ViewItem, PutBid, BuyNow, Sell, etc. The workload generator can simulate many clients that follow a Markov model to browse and take actions on the auction site and there is also some “thinking time” between the actions.

We select the PHP version of RUBiS implementation and set up a three-tier testbed, including a web server, a database, and a client emulator as shown in Figure 5.5. We use Apache version 2.2.12 with PHP version 5.3.2 and MySQL
Figure 5.5. Experiment setup of the RUBiS benchmark

version 5.1.73. The client emulator is compiled and run using Java 6. We collect the information leakage from the Apache web server. On the client side, we simulate 100 users concurrently browsing the website. Since the size of the log file grows so quickly, we pause the client emulator every one minute, perform the measurement, delete the log files and then resume the client emulator for another one minute. All the results reported below are averaged over 10 rounds of such iterations.

5.5.2 Risk Assessment

Metric 1. In the log files dumped from the baseline run (i.e., without applying any technique), we found that the ASCII control code $0x06$ rarely appears under our experiment setup. So we choose $0x06$ for the four techniques to do memory cleaning. We then count the number of this particular value in the log files and compute the ratio over the entire file size. In order to get a more consistent comparison, buffer over-read attacks using size 1KB, 16KB, and 32KB are simulated in the same run. That is, we hook `memcpy` and do three `write`’s to three log files that correspond to the three over-read sizes.

The results are shown in Figure 5.6(a). The ratio of information-carrying bytes for the baseline is nearly 100%, while the others are mostly below 80%. This indicates that all of the four techniques can reduce the information-carrying bytes. The larger the over-read size is, the less percentage of information-carrying bytes the attacker can get. For the over-read size of 32KB, the average ratio of information-carrying bytes reduces to 63.4%. It is interesting to point out that
Figure 5.6. Experimental results of Metric 1, 2, and 3 on baseline and the four techniques: T0-baseline, T1-padding, T2-erasing (inline), T3-erasing (concurrent), T4-erasing (concurrent) + padding.
attackers usually tend to try over-read with a longer length as they think they will gain more information. However, using a longer size will increase the possibility of failure (e.g., causing a crash). Now our results seem to indicate that the belief of “longer over-read size will lead to more data leakage” does not always hold. So an attacker using a shorter size can still gain similar amount of information. From the attackers’ perspective, however, this sounds to be a good news.

Comparing the four different techniques, the rough conclusion is that erasing plus padding is better than purely erasing, which is in turn better than purely padding. The inline erasing is oftentimes better than the concurrent erasing because in concurrent erasing, user threads calling \texttt{free} and the eraser thread erasing the memory are asynchronous events so that there is a small time window during which the data is still left on memory. However, this advantage comes with some performance penalty (see Section 5.5.3).

**Metric 2.** As shown in Figure 5.6(b), we use three compression algorithms to approximate the entropy of information contained in the leaked data. We first calculate the compression ratio for each compression algorithm on every test cases and use the compression ratio of \texttt{gz} on the baseline as the normalization factor to normalize the compression ratio of others. The over-read sizes of the test cases shown in Figure 5.6(b) are all 16KB. The difference between the average compression ratios of the three algorithms is attributed to their inherent difference in compression and encoding schemes. Overall, the techniques using erasing are more effective in reducing the information entropy than the padding technique. The average entropy of T2, T3, and T4 is almost only half of the baseline’s entropy, indicating that these techniques are indeed able to mitigate the information leakage to a great extent.

**Metric 3.** For the RUBiS benchmark, we define the user’s information as sensitive and thereby targeted by attackers. Specifically, the sensitive information of a user include first name, last name, email address, nickname (i.e., username), and password. As mentioned earlier, in the RUBiS benchmark every user has a unique
user ID and these sensitive information all share the same pattern: [keyword][user ID]. For example, all of the usernames and passwords have the pattern of “user[userID]” and “password[userID]”, respectively. Consequently, we analyze the log file and search for the sensitive information by pattern matching with regular expressions.

Interestingly, we found that not all the results that match the patterns are valid. The log file fragment shown in Figure 5.7(a) gives an example. The strings “User1642528” and “Great1642528” are the valid last name and first name of the user whose user ID is “1642528”. However, the strings “user164252” and “password16” are truncated by other data and thus invalid. We regard these invalid results as false positives and filter them out through correlating with the client emulator to check the validity of user IDs. It is worth mentioning that knowing the first part of a password greatly reduces the search space for a brute-force password attack. Since further quantifying the difficulty of password cracking is beyond the focus of this chapter, we simply regard these truncated password as invalid. In addition, to also avoid counting the sensitive information leaked by the same user (which is virtually not an information leak), we modify the client emulator to create different groups of users for every 1-minute interval. As such, we check every interval (except the first one) against the users in the previous intervals to determine the sensitive data leak.

Figure 5.6(c) shows the quantity of sensitive information retrieved from the leaked data. Since the username-password pair is normally considered much more

![Image](a)

![Image](b)

Figure 5.7. Leaked data showing (a) some sensitive information and (b) a valid username-password pair
sensitive and can be used to directly compromise a user account, we additionally measure the quantity of valid username-password pairs. Figure 5.7(b) shows a fragment containing a complete pair of username and password. For all the test results in Figure 5.6(c), we can observe a clear difference on the quantity of sensitive data leaked with and without the mitigation techniques. The huge difference suggests that even a simple technique might be able to reduce the risk to a large extent.

**Metric 4.** Lastly, we apply Metric 4 to measure the quantity of unique sensitive data leaked. Table 5.2 lists the numbers of different test cases over the three sizes. As for the over-read size, we can see that the difference it makes is quite minimal, especially for the baseline which is totally the same across 1KB, 16KB, and 32KB. On the other hand, the four techniques are further proven to be effective. The number of unique username-password pairs for T4 is even reduced to zero under the 1KB over-read size.

### 5.5.3 Performance Comparison

We further compare the performance of the four techniques in terms of microbenchmark runtime overhead, web server throughput overhead and memory overhead. First, we write a microbenchmark which allocates a series of memory ranging from 10 bytes to 20KB (∼200MB in total), does some computation, sleeps for a while (to simulate I/O), deallocates all the memories, and repeats this iteration for 50
Table 5.3. Throughput and memory overhead on Apache

<table>
<thead>
<tr>
<th>Techniques</th>
<th>Throughput Overhead</th>
<th>Memory Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Padding</td>
<td>0.4%</td>
<td>2.6%</td>
</tr>
<tr>
<td>Inline</td>
<td>0.2%</td>
<td>0%</td>
</tr>
<tr>
<td>Concurrent</td>
<td>0.1%</td>
<td>3.9%</td>
</tr>
<tr>
<td>Concurrent + Padding</td>
<td>0.1%</td>
<td>6.2%</td>
</tr>
</tbody>
</table>

times. Figure 5.8 shows the runtime of the microbenchmark in different test cases. Compared to (inline) padding and inline erasing, concurrent erasing incurs much less runtime overhead, which is even lower than the baseline. The reason is that in the concurrent erasing, the main thread does not even need to do the real stuff in *free* (which is delegated by the eraser thread).

Second, we use Apache web server and ApacheBench version 2.3 to measure the web server throughput overhead and RSS memory overhead. The concurrency of ApacheBench is 50 and the number of request is 10,000 for each run. We repeat the test 10 times and take the average for each measurement. The experiments are done on a DELL T310 server with 2.53GHz quad-core CPU and 4GB RAM. The result in Table 5.3 shows that the throughput overhead for all four techniques is negligible. As for memory overhead, padding and concurrent erasing will each increase roughly 3%, which is still very small.

![](Runtime comparison on microbenchmark.png)

**Figure 5.8.** Runtime comparison on microbenchmark
5.6 Discussion

It should be pointed out that the results of risk measurement and performance of different mitigation techniques are affected by various factors, including the characteristics of programs, the workloads, how the sensitive data is defined, etc. Even in our own experiment, for example, the performance comparison for the microbenchmark and the web server are not in line with each other. Moreover, in the real world, the measurement results of different setups may also be different. Therefore, we encourage system administrators or security officers to do their own measurement if they are interested in adopting our method to do risk assessment.

In Metric 3 and 4, we only measure the leakage of sensitive data across intervals. In theory, it is also useful to measure the leakage within the same interval (i.e., one user steals the data of a concurrent user). However, this requires hacking into the web server to associate each invocation of memcpy with the corresponding user ID and involves modifying and recompiling the program, which may not be a practical option in many cases. We plan to explore non-intrusive ways to achieve this in the future.

Another limitation is that given the limited disk space, our current approach suffers from the fast growing speed of the log files. One solution is to embed the analysis component into the data collection component so that the analysis component can consume the data on the fly and the log files are no longer needed. However, doing this integration might change the heap memory layout and thus affect the results. We leave this issue for future work.

5.7 Summary

The research on buffer over-read vulnerability is gaining more and more attentions from both academia and industry recently. This is mainly due to the vast damage that the Heartbleed bug has caused and, more importantly, it is much more difficult
to detect and fully defeat compared to buffer overflow attacks. As the Heartbleed bug has remained undiscovered for about two years, it is entirely possible that zero-day “Heartbleed” over-read vulnerabilities still exist or will be introduced. To evaluate new counter-measures against buffer over-read attacks, people would need to perform risk assessment on all the potential information leakage caused by unknown “Heartbleed” vulnerabilities.

We present a set of feasible metrics to quantitatively evaluate the potential amount of information leak associated with memcpy operations in buffer over-read attacks. We develop practical methods to collect and measure information leakage in real world programs. We report our experiments on an auction site prototype modeled after eBay.com. Our experience reveals that even some simple non-intrusive techniques can achieve reasonable defense against buffer over-read in terms of information leakage as well as performance.
Chapter 6  |
Conclusion

In this dissertation, we perform a three-dimensional research study emphasizing on protecting server programs and systems, including privilege separation, attack surface reduction, and risk assessment.

First, we present Arbiter, a system targeting at fine-grained, data object-level privilege separation for multithreaded server programs. Particularly, we find that page table protection bits can be leveraged to do efficient reference monitoring if data objects with same accessibility are put into the same page. To make a multithreaded program more secure, programmers only need a minimum amount of effort to specify security policy in source code. We find that Arbiter is applicable to a verity of real-world applications. Our experiments demonstrate Arbiter’s ease of adoption, effectiveness of protection, as well as reasonable performance overhead.

Second, we present LIPZip, a system that identifies idling services and reduces attack surface in server systems. We find that system calls can be utilized to identify and constrain idling processes that expose attack surface. We propose an adaptive autocorrelation-based algorithm which can robustly detect idling process based on system call information. We evaluate the effectiveness of our algorithm in a real production environment, using a custom built light-weight and scalable data collection system. We show that there does exist a considerable amount of idling services and demonstrate that most of the identified idling processes can be safely
and automatically constrained to reduce attack surface without human intervention and that a portion of them can even be safely disabled to fully eliminate their attack surface.

Third, motivated by the need to understand the risks associated with potential vulnerabilities in a server program and to quantitatively compare the effectiveness of different mitigation techniques, we present a set of feasible metrics to quantitatively evaluate the potential amount of information leak associated with `memcpy` operations in buffer over-read attacks. We develop practical methods to collect and measure information leakage in real world server programs. We report our experiments on an auction site prototype modeled after eBay.com. Our experience reveals that even some simple non-intrusive techniques can achieve reasonable defense against buffer over-read in terms of information leakage as well as performance.

In conclusion, we propose and demonstrate new ideas and insights for protecting server programs and systems via fine-grained privilege separation, identifying and reducing attack surface, as well as understanding and mitigating security risks. We believe protecting server programs and systems will become more and more important and we hope our work can facilitate further studies and practices on this topic.
Bibliography


“CVE vulnerabilities: file with the write mode,” http://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=file+write+mode.


Vita

Jun Wang

Jun Wang is currently a PhD candidate of Information Sciences and Technology at Penn State University. He joined the PhD program of College of Information Sciences and Technology, Penn State University in 2010 as a recipient of the Robert W. Graham Endorsed Graduate Fellowship. Before that, he received his BS degree in Electronic Science and Engineering from Nanjing University in 2010. He worked as a graduate research assistant in the Cyber Security Lab, Penn State University starting from 2010 under the supervision of Professor Peng Liu. His research interests include systems security, network security, distributed systems, and cloud computing. His work has led to four research papers (three first-authored) published and presented at major systems and security conferences including USENIX ATC and USENIX Security. He is also a student member of ACM and IEEE.