ABSTRACT

TUNDE-ONADELE, OLUFOGOREHAN ADETAYO. Detecting Security Attacks and Vulnerabilities in Cloud Server Systems. (Under the direction of Xiaohui Gu).

Cloud platforms have become the most popular infrastructures for hosting various application services. As a result, security attacks to those cloud server systems often cause massive impact to many users. Although existing intrusion detection systems can detect certain attacks afterwards, they cannot prevent those attacks from occurring before the vulnerable code is released into production systems. Moreover, understanding the underlying software defects that cause those security vulnerabilities is little studied. In this dissertation, we investigate the problems of security attack and vulnerability detection to help developers proactively diagnose and patch security bugs.

Firstly, we conduct a comparative study on the effectiveness of various static and dynamic vulnerability schemes for containers using exploits to 28 real world vulnerabilities that widely exist in docker images. Our results show that the static vulnerability scanning scheme only detects 3 out of 28 tested vulnerabilities and the dynamic anomaly detection schemes detect 22 exploits. Combining static and dynamic schemes can further improve the detection rate to 86% (i.e., 24 out of 28 exploits). We also observe that the dynamic schemes can achieve over 20 seconds lead time (i.e., a time window before attacks succeed) for a group of commonly seen attacks in containers that try to gain a shell and execute arbitrary code.

Secondly, we present our work on security attack detection and patching. We propose Self-Patch, a self-triggering patching framework for applications running inside containers. Self-Patch combines lightweight runtime attack detection and dynamic targeted patching to accurately detect and classify 81% of attacks and reduce patching overhead by up to 84% in our evaluation of over 31 real world attacks in 23 commonly used server applications.

Thirdly, we conduct a systematic study of over 110 software security vulnerabilities in 13 popular cloud server systems. We find that the vulnerable code of the studied security vulnerabilities comprise five common categories: 1) improper execution restrictions, 2) improper permission checks, 3) improper resource path-name checks, 4) improper sensitive data handling, and 5) improper synchronization handling. We further extract principal vulnerable code patterns from these common vulnerability categories.

Lastly, we use the identified principal vulnerable code patterns to proactively protect

cloud systems from security vulnerabilities due to improper code execution restrictions. XScope, an automatic pattern-driven code execution vulnerability checker, leverages insights about the root cause functions and the security patches from recent vulnerabilities to optimize the vulnerability detection accuracy. We evaluated a prototype of XScope using real world vulnerabilities on six commonly used real cloud server systems. XScope can accurately localize the vulnerable functions including the high impact Log4j vulnerability with higher detection rate and 53% lower false alarm rate than alternative schemes.

Detecting Security Attacks and Vulnerabilities in Cloud Server Systems

by
Olufogorehan Adetayo Tunde-Onadele

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina
2024

APPROVED BY:

Guoliang Jin	Rada Chirkova
Huiyang Zhou	Xiaohui Gu Chair of Advisory Committee

DEDICATION

To my parents, Akintunde Onadele and Abimbola Tunde-Onadele, and my siblings, Olufayokunwa Adekanye, Fiyinfoluwa Tunde-Onadele, and Oluferinkunwa Tunde-Onadele, for their overwhelming love and support on my graduate journey.

BIOGRAPHY

Olufogorehan (Fogo) Tunde-Onadele was born and raised in Lagos, Nigeria, where he obtained his early education. He received his B.S. degrees in Electrical and Computer Engineering as well as his M.S. degree in Computer Science from North Carolina State University in 2017 and 2019, respectively. He pursued his graduate research interests under the supervision of Dr. Xiaohui (Helen) Gu. His research contributes to improving the detection capabilities of cloud server systems against security attacks and vulnerabilities so that people can freely be productive without fear of cyberattacks. During his graduate studies, Fogo interned at Juniper Networks, IBM Research, and Samsung Semiconductor, Inc. in 2022, 2021, and 2019.

ACKNOWLEDGEMENTS

I would like to express my utmost gratitude to my advisor, Dr. Helen Gu and my committee, Dr. Guoliang Jin, Dr. Rada Chirkova. and Dr. Huiyang Zhou. I am also thankful for my collaborators for their help and hard work on the journey: Dr. Yuhang Lin, Dr. Jingzhu He, Dr. Ting Dai, and Alex Qin. I appreciate the NC State CSC department, including the career center directed by Ms. Leslie Rand-Pickett as well as the administrative team behind the scenes that make things happen.

I am very grateful for the continuous love, support and encouragement of my family, friends, and mentors that kept me going! Finally, I would like to extend an overwhelming appreciation to my parents, who labored so much to support me through my education despite all the hardship. I would never have envisioned pursuing a doctoral degree without their coaching and encouragement.

TABLE OF CONTENTS

List of Tables . . . . . **vii**

List of Figures . . . . . **viii**

Chapter 1 INTRODUCTION **1**
1.1 Motivation 1
1.2 Summary of the State of the Art 2
1.3 Summary of Contributions 2

Chapter 2 A Study on Container Vulnerability Exploit Detection **5**
2.1 Introduction 5
2.2 Methodology 8
2.2.1 Real-World Vulnerabilities 9
2.2.2 Static Vulnerability Detection Scheme 10
2.2.3 Dynamic Exploit Detection Approaches 10
2.3 Experimental Evaluation 12
2.3.1 Experiment Setup 12
2.3.2 Detection Results 13
2.4 Summary 17

Chapter 3 Self-Patch: Beyond Patch Tuesday for Containerized Applications . . . **18**
3.1 Introduction 18
3.2 System Design 20
3.2.1 System Overview 20
3.2.2 Attack Detection 21
3.2.3 Attack Classification 23
3.2.4 Targeted Patch Execution 24
3.3 Experimental Evaluation 26
3.3.1 Evaluation Methodology 27
3.3.2 Results and Analysis 29
3.4 Summary 35

Chapter 4 Understanding Security Vulnerabilities in Cloud Server Systems **36**
4.1 Introduction 36
4.1.1 Motivating Example 38
4.1.2 Contribution 38
4.2 Methodology 39
4.2.1 Real-world security bug discovery 39
4.2.2 Vulnerable Code Categories 42
4.3 Security Bug Characteristics 44

4.3.1 Improper execution restrictions . . . 44
4.3.2 Improper permission checks . . . 45
4.3.3 Improper resource path-name checks . . . 47
4.3.4 Improper sensitive data handling . . . 50
4.3.5 Improper synchronization handling . . . 52
4.4 Summary . . . 53

Chapter 5 XScope: Detecting Code Execution Vulnerabilities in Cloud Server Systems . . . 54
5.1 Introduction . . . 54
5.1.1 Motivating Example . . . 55
5.1.2 Contribution . . . 57
5.2 System Design . . . 58
5.2.1 Overview . . . 58
5.2.2 Vulnerable Code Execution Patterns . . . 60
5.2.3 Vulnerable Candidate Discovery . . . 62
5.2.4 Safe Candidate Filtering . . . 63
5.3 Evaluation . . . 67
5.4 Experimental Evaluation . . . 67
5.4.1 Evaluation Methodology . . . 67
5.4.2 Results Analysis . . . 71
5.4.3 Run-time Measurements . . . 73
5.4.4 Case Study . . . 74
5.5 Summary . . . 77

Chapter 6 RELATED WORK . . . 78
6.1 Intrusion Detection . . . 78
6.2 Vulnerability Categorization . . . 79
6.3 Vulnerability Detection . . . 80

Chapter 7 CONCLUSION . . . 84

References . . . 87

LIST OF TABLES

Table 2.1 List of Explored Real-world Vulnerabilities. 8
Table 2.2 An example of frequency vectors from a processed system call list. . . 11
Table 2.3 Detection Result of Clair and Anomaly Detection Approaches. 13
Table 2.4 The Lead Time of Anomaly Detection Approaches for CVEs that Return a Shell and Execute Arbitrary Code. "-": the approach does not detect the vulnerability. 16

Table 3.1 A frequency vector sample for the ActiveMQ application (CVE-2016-3088). An attack is triggered at $t = 1528903079912$. 21
Table 3.2 List of explored real-world vulnerabilities. 26
Table 3.3 Detection result of Self-Patch and alternative approaches. 30
Table 3.4 Top system call composition of generated patterns. 33
Table 3.5 Overall comparison result of different patching approaches. 33

Table 4.1 List of studied vulnerabilities by root cause. 41

Table 5.1 Call stack snippet during an Apache Log4j (CVE-2021-44228) exploit. 59
Table 5.2 List of security sensitive API. 67
Table 5.3 List of explored real-world vulnerabilities with unique root cause functions. 67
Table 5.4 XScope detection result comparison with vulnerable candidate discovery (VCD) only and related work, FindSecBugs. 69
Table 5.5 Summary of the alternative filtering schemes compared to XScope. . 69
Table 5.6 Detection result comparison across the alternative schemes. 70
Table 5.7 Runtime measurements of the XScope components 73

LIST OF FIGURES

Figure 3.1 System overview of Self-Patch. 20
Figure 3.2 Architecture of the autoencoder. 22
Figure 3.3 A targeted patching example for Ghostscript. 24
Figure 3.4 True positive rate result of anomaly detection approaches. 31
Figure 3.5 False positive rate result of anomaly detection approaches. 31
Figure 3.6 Lead time result of anomaly detection approaches. 32
Figure 3.7 Container memory cost of patching. 34
Figure 3.8 Container disk cost of patching. 34

Figure 4.1 The Apache Log4j CVE-2021-44228 bug (CVSSv3: 10.0, CVSSv2: 9.3). The vulnerable function *lookup* does not restrict the lookup of JNDI URIs before instantiating the requested class with the security-sensitive *getObjectFactoryFromReference* function. This 'improper execution restrictions' bug has the 'execute arbitrary code' impact. 37
Figure 4.2 Distribution of security bugs by threat impact. 40
Figure 4.3 Distribution of security bugs by vulnerable code category. 41
Figure 4.4 The Apache Struts 2 CVE-2018-11776 bug (CVSSv3: 8.1, CVSSv2: 9.3). The vulnerable function *invokeMethod* calls the security-sensitive Java *invoke* method to execute commands inserted into the namespace section of a URI without restriction. This 'improper execution restrictions' bug has the 'execute arbitrary code' impact. 43
Figure 4.5 The ActiveMQ CVE-2018-11775 bug (CVSSv3: 7.4, CVSSv2: 5.8). The function *initializeStreams* never sets an identification algorithm so the variable *identityAlg* does not meet the condition to call the security-sensitive *checkIdentity* function to better secure TLS connections. This 'improper permission checks' bug has the 'disclose credential information' impact. 46
Figure 4.6 The Apache Tomcat CVE-2017-12615 bug (CVSSv3: 8.1, CVSSv2: 6.8). The vulnerable function *file* does not verify safe file extensions after calling the security-sensitive *normalize* function that can modify the file extension. This 'improper resource path-name checks' bug has the 'execute arbitrary code' impact. 48
Figure 4.7 The WildFly CVE-2020-25640 bug (CVSSv3: 5.3, CVSSv2: 3.5). The vulnerable function *toString* outputs the security-sensitive *pwd* password variable. This 'improper sensitive data handling' bug has the 'disclose credential information' impact. 50

Figure 4.8 The Elasticsearch CVE-2019-7614 bug (CVSSv3: 5.9, CVSSv2: 4.3). The vulnerable function *put* modifies the *itemTuple* variable outside its critical section, which leads to a race condition that can mix user data. This 'improper synchronization handling' bug has the 'disclose credential information' impact. 51

Figure 5.1 A real-world execution vulnerability in Apache Log4j (CVE-2021-44228). The function does not perform security checks on the `name` variable during a JNDI lookup. This allows malicious user inputs in `name` to eventually execute. 56

Figure 5.2 System overview of XScope. 58

Figure 5.3 An example input validation (with regex) patch for Apache Struts CVE-2018-11776 60

Figure 5.4 An example allowlist patch for Apache Log4j CVE-2021-44228 61

Figure 5.5 An example security variable patch for Apache Commons CVE-2017-7504 61

Figure 5.6 An example call graph illustration of Apache Log4j (CVE-2021-44228), representing function calls from caller to callee. We trace the call graph backward from the key system-level execution functions to find vulnerable function candidates in the application libraries. . . . 63

Figure 5.7 Data-flow illustration using Apache Log4j (CVE-2021-44228). A candidate is considered vulnerable if there are unfiltered data-flow paths from a program input source to the candidate such as the path: *source isr* $\rightarrow$ *subpath 2* $\rightarrow$ *subpath A*. 64

Figure 5.8 Data-flow scenario with a trivial check in the candidate function. XScope filters out paths with non-trivial if predicates to avoid missing true positives. 66

CHAPTER

1

INTRODUCTION

1.1 Motivation

Cloud servers provide a cost-effective platform for deploying software applications in a pay-as-you-go fashion. However, due to its multi-tenant sharing nature, the cloud environment is especially vulnerable to security attacks. Due to its widespread deployment, any security vulnerability in cloud server systems can cause extensive impact on the end users Shu et al. (2017).

For instance, vendors of the popular Java logging library, Apache Log4j, reported a serious vulnerability on December 9, 2021, affecting industries worldwide Wetter and Ringland (2021); Korn (2021). The vulnerability, named Log4Shell, allowed attackers to execute any commands in cloud systems that contained the library, resulting in about 200,000 global attacks within one day of the disclosure Ltd (2021). The open source insights team from Google Cloud estimates that Log4Shell affected 8% of all artifacts in the Maven Central repository, which is four times the average vulnerability impact Wetter and Ringland (2021). Thus, it is important to build efficient and non-intrusive solutions that diagnose unseen

security attacks and deliver bug fixes to protect distributed computing environments.

Moreover, software bugs that execute arbitrary malicious code are notoriously bad for cloud systems. In our comprehensive study, we find that such improper execution restrictions vulnerabilities Tunde-Onadele et al. (2022) are the predominant code vulnerability category affecting cloud server systems. Thus, developing an automatic security vulnerability detection tool that identifies vulnerable code patterns can help developers proactively localize the vulnerable functions in complex large-scale cloud server programs.

1.2 Summary of the State of the Art

Existing work has been focusing on detecting security attacks using intrusion detection systems (IDS) that take a variety of approaches including machine learning techniques Shen et al. (2018); Lin et al. (2022, 2020); Tunde-Onadele et al. (2020); Yen et al. (2013); Dash et al. (2016). IDS approaches can only detect a security attack after the attack happens. Such traditional intrusion detection approaches are reactive in nature. Moreover, they often cannot reveal how an attack has been triggered or pinpoint the vulnerable code that is the culprit of the security attack.

Other previous work has been done to detect code vulnerabilities with static program analysis Thomé et al. (2017b); Livshits and Lam (2005); Enck et al. (2014); Zheng and Zhang (2013) or symbolic execution Thomé et al. (2017a); Fratantonio et al. (2016). However, existing vulnerability detection solutions often suffer from either high false positives or high false negatives due to too general or too narrow rule-based approaches.

1.3 Summary of Contributions

In this dissertation, we make the following contributions:

- We present a comparative study on the effectiveness of various vulnerability detection schemes for containers. Specifically, we evaluate a set of static and dynamic detection schemes using 28 real world vulnerabilities that widely exist in docker images. Our results show that the static vulnerability scanning scheme only detects 3 outs of 28 tested vulnerabilities and the dynamic anomaly detection schemes detect 22 vulnerability exploits. Combining static and dynamic schemes can further improve

the detection rate to 86% (i.e., 24 out of 28 exploits). We also observe that the dynamic anomaly detection scheme can achieve more than 20 seconds lead time (i.e., a time window before attacks succeed) for a group of commonly seen attacks in containers that try to gain a shell and execute arbitrary code.

- We present Self-Patch, a new self-triggering targeted patching framework for container-based distributed computing environments. To achieve this goal, the Self-Patch framework consists of three coordinating components: 1) an online attack detection module which can dynamically detect abnormal attack activities by extracting feature vectors from system call traces and applying unsupervised machine learning methods over the features; 2) an attack classification scheme which classifies a detected attack into a specific type linked to a certain CVE; and 3) a targeted patch execution module which can install proper software patches to fix the vulnerability. We have implemented a prototype of Self-Patch and evaluated it over 31 real-world vulnerabilities discovered in 23 common server applications. Our experimental results show we can increase detection rate to over 80% and reduce false alarm rate to 0.7%. In contrast, traditional whole software upgrade schemes can either only detect 6% attacks or incur more than 20% false alarms. Self-Patch can also reduce the memory overhead by up to 84% and disk overhead by up to 40%.

- We present a comprehensive study over 110 recent real world security bugs in 13 popular cloud server systems. Our study first identifies five common vulnerability categories among those 110 studied security bugs: 1) *improper execution restrictions*, 2) *improper permission checks*, 3) *improper resource path-name checks*, 4) *improper sensitive data handling*, and 5) *improper synchronization handling*. Furthermore, we extract key software code patterns in each category and describe a set of pattern-driven strategies for detecting those security bugs before they are released to production cloud environments.

- We present XScope, a new pattern-driven fine-grained vulnerability detection framework for proactively detecting security bugs due to improper code execution restrictions. XScope not only detects code execution vulnerabilities but also localizes the vulnerable functions in complex large-scale cloud server programs consisting of tens of thousands of functions. Furthermore, XScope combines call graph analysis and data-flow analysis to minimize false positive rates while maintaining a high detection

rate. We have implemented a prototype of XScope and tested it using real world vulnerabilities including the high impact Log4j vulnerability on six commonly used cloud server systems. Our experimental results show that XScope can achieve a 100% detection rate while existing security checking tools like FindSecBugs can only detect 38% of those CVEs. Moreover, XScope can reduce the false positive rate by 53% for those CVEs that can be detected by both XScope and FindSecBugs.

The rest of the dissertation is organized as follows. Section 6 compares our work with related work Section 2 describes our preliminary study evaluating state-of-the-art static and dynamic security attack detection schemes for containerized applications. Section 3 presents our self-triggering patching framework for container-based distributed computing environments. Section 4 discusses our comprehensive study and categorization of real world security bugs in cloud server systems. Section 5 details our pattern-driven code execution vulnerability detection framework for proactively protecting cloud systems. Section 7 concludes the dissertation.

CHAPTER

2

A STUDY ON CONTAINER VULNERABILITY EXPLOIT DETECTION

2.1 Introduction

Containers have recently become a popular application deployment platform that can package an application and its dependencies (e.g., source code, system libraries) with lower overhead than virtual machines. However, due to its easy deployment nature, containers are prone to various security vulnerabilities. Previous work has shown security vulnerabilities widely exist in both official and community images Gummaraju et al. (2015); Shu et al. (2017); Docker Image Vulnerability Research (2017). Vulnerabilities in outdated packages can be exposed to various types of attacks (e.g., denial of service, gain privilege, execute code) and vulnerabilities can propagate due to dependency relationships between images Shu et al. (2017). Hence, security has become one of the top concerns for the user to use containers in production environments Bettini (2015).

Existing container vulnerability detection schemes can be broadly classified into two

groups: 1) static container image analysis and 2) dynamic runtime detection. The static schemes mainly focus on static vulnerability detection using container image scanning Clair (2017); Dockscan (2018); Banyan Collector (2018); OpenSCAP Container Compliance (2016). Static image scanners can detect known vulnerabilities by matching the packages and their versions with remote Common Vulnerabilities and Exposures (CVE) databases. However, the identified package list might not always accurately include all the packages installed, and customized code or scripts are not analyzed through static analysis. Moreover, vulnerabilities that are not included in existing CVE databases will not be detected (e.g., vulnerabilities not publicly disclosed, zero-day vulnerabilities). Dynamic runtime detection tools monitor container behaviors and detect anomalous activities during runtime Twistlock (2017); NeuVector (2018); Sysdig Falco (2018). However, most of these tools are policy-based, which cannot adapt to changing behaviors. For example, Sysdig Falco Sysdig Falco (2018) employs pre-defined policies that describe the allowed or disallowed behaviors for a process, in terms of system calls, their arguments, and host resources accessed.

In this chapter, we conduct a study over different vulnerability detection techniques and evaluate their effectiveness on detecting security vulnerabilities of the applications running inside containers. Particularly, we focus on out-of-box detection techniques which do not require any modifications to monitored applications and are more resilient to attacks than inside-box schemes. We consider both static and dynamic detection techniques and perform comparisons among them in terms of detection accuracy and overhead.

Compared to traditional host environments, containers present a set of new challenges to vulnerability exploit detection: 1) containers are often short-lived, which implies that the detection scheme needs to produce real time alerts without requiring a large amount of training data; 2) containers are often dynamic, which requires that the detection should not make any assumption about the container such as available resources or application workloads; and 3) containers are often light-weight, which requires that the detection algorithm should not impose high overhead to the container.

We first study the open source static analysis engine Clair Clair (2017) as an example for static analysis tools. Clair inspects containers layer-by-layer for known vulnerabilities, which continuously imports vulnerability data from a set of resources (e.g., Debian Security Bug Tracker, Ubuntu CVE Tracker, Red Hat Security Data). Container images are indexed into a list of features (e.g., installed packages, package versions), and Clair queries the vulnerability data to correlate the indexed features with vulnerability database to generate a list of vulnerabilities that threaten the images. We then study a set of dynamic detection schemes

using unsupervised anomaly detection algorithms (e.g., clustering Kanungo et al. (2002), k nearest neighbor Altman (1992), self-organizing map Kohonen (1998)). Compared to supervised machine learning, unsupervised anomaly detection approaches do not require labeled training data and can capture previously unseen attacks. We evaluate these different detection schemes using real-world vulnerabilities that are triggered in commonly used server applications such as Tomcat, Apache, and ElasticSearch.

Specifically, this chapter makes the following contributions:

- We reproduce 28 commonly seen real world security vulnerabilities discovered in Docker Hub images and conduct a comparative study over both static and dynamic vulnerability detection schemes using those security vulnerabilities.
- We collect the detection accuracy of CoreOS Clair, an open source static Docker image vulnerability detection tool. Our results show that Clair can only detect 3 out of the 28 vulnerabilities.
- We implement a system call collection and feature extraction system and apply a set of widely used unsupervised anomaly detection schemes (i.e., k nearest neighbors, k-means clustering, k-nearest neighbors combined with principal component analysis for dimension reduction, self-organizing map) to catch triggered attacks online.

Our results show that it is promising to use dynamic anomaly detection schemes to catch vulnerability exploits in containers: self-organizing map based anomaly detection can catch 22 out of 28 tested vulnerability exploits while incurring a low false positive rate (1.7% on average). Moreover, the dynamic anomaly detection scheme can achieve more than 20 seconds lead time (e.g., a time window before attacks succeed) for a group of attacks that try to gain a shell and execute arbitrary code. We also find that it is beneficial to combine static and dynamic vulnerability detection schemes, which can further improve the detection coverage to catch 24 exploits.

The rest of the section is organized as follows. section 2.2 presents our empirical study methodology. section 2.3 describes the experimental results. Finally, the section concludes in section 2.4.

Table 2.1 List of Explored Real-world Vulnerabilities.

Threat Impact	CVE ID	CVSS Score	Application	Exploitation Tool
Return a shell and execute arbitrary code	CVE-2015-8103	7.5	JBoss	JexBoss
	CVE-2017-7494	10.0	Samba	Metasploit
	CVE-2016-10033	7.5	PhpMailer	Metasploit
	CVE-2015-2208	7.5	phpMoAdmin	Metasploit
	CVE-2016-9920	6.0	Webmail	PoC
	CVE-2015-1427	7.5	Elasticsearch	Metasploit
	CVE-2014-3120	6.8	Elasticsearch	Metasploit
	CVE-2012-1823	7.5	PHP	Metasploit
	CVE-2017-11610	9.0	Supervisor	Metasploit
	CVE-2017-8291	6.8	Ghostscript	PoC
	CVE-2015-3306	10.0	ProFTPd	Metasploit
	CVE-2017-12615	6.8	Apache Tomcat	PoC
	CVE-2016-3088	7.5	Activemq	Metasploit
	CVE-2017-12149	7.5	JBoss	PoC
	CVE-2015-8562	7.5	Joomla	Metasploit
Execute arbitrary code	CVE-2014-6271	10.0	Bash	Metasploit
	CVE-2017-5638	10.0	Struts	PoC
	CVE-2017-12794	4.3	Django	PoC
	CVE-2016-3714	10.0	ImageMagick	Metasploit
Disclose credential information	CVE-2017-7529	5.0	Nginx	PoC
	CVE-2015-5531	5.0	Elasticsearch	Metasploit
	CVE-2014-0160	5.0	OpenSSL	Metasploit
	CVE-2017-8917	7.5	Joomla	sqlmap
Consume excessive CPU	CVE-2016-6515	7.8	OpenSSH	PoC
	CVE-2014-0050	7.5	Apache Tomcat	PoC
Crash the application	CVE-2016-7434	5.0	NTP	PoC
	CVE-2015-5477	7.8	BIND	Metasploit
Escalation of privilege	CVE-2017-12635	10.0	Couchdb	Burp Suite

PoC: Proof of Concept code.

2.2 Methodology

In this section, we describe our study methodology. We first introduce the real-world vulnerabilities studied. We then describe the set of static and dynamic vulnerability detection schemes considered.

2.2.1 Real-World Vulnerabilities

Table 2.1 shows the 28 real-world vulnerabilities collected in 24 different applications from the commonly used vulnerability repository, i.e., Exploit Database Offensive Security's Exploit Database Archive (2018). We categorize all the 28 vulnerabilities into six groups based on their threat impact: 1) return a shell and execute arbitrary code, 2) execute arbitrary code, 3) disclose credential information, 4) consume excessive CPU, 5) make applications crash, and 6) perform escalation of privilege. These categories are among the top vulnerability types discovered in Docker Hub Shu et al. (2017). Most of these vulnerabilities are reported within the past three years and marked with "High" or "Critical" severity rankings, denoted by CVSS scores [1]. Our application set also exhibits a wide coverage, ranging from back-end database systems to front-end web servers to represent different server applications running inside containers.

We exploit the vulnerabilities by either executing the Proof of Concept (PoC) code or using penetration tools (i.e., Metasploit Metasploit penetration testing framework (2018), JexBoss JexBoss (2018), sqlmap Sqlmap (2018), and Burp Suite Burp Suite Scanner (2018)). To emulate dynamic applications in real world, we employ commonly used workload generator tools (e.g., Burp Suite Burp Suite Scanner (2018), JMeter Apache JMeter (2018)) to send requests to victim containers.

For web server applications such as Apache Tomcat, Django and Nginx, we request pages from web servers with JMeter's HTTP sampler. This sampler enables the selection of the appropriate HTTP traffic type (e.g., `GET`, `POST`, etc.) for an application. Web requests are also sent to Joomla and Couchdb front ends to induce database operations (e.g., create, update and delete documents). For FTP servers such as ProFTPd, files are downloaded from and uploaded to the FTP server using the FTP sampler. The date requests are sent to the OpenSSH application via the JMeter plugin (i.e., SSH command). The Domain Name Server (DNS) and Network Time Protocol (NTP) requests are sent to the BIND and NTP applications via the JMeter plugin (i.e., UDP request). The smbclient is used with JMeter's OS process sampler to produce Server Message Block (SMB) network traffic for the Samba application. As for the Elasticsearch, we send search requests via Burp Suite.

[1] Common Vulnerability Scoring System (CVSS) scores are provided by National Vulnerability Database. The higher the score, the higher the severity (i.e., "None": 0.0; "Low": 0.1-3.9; "Medium": 4.0-6.9; "High": 7.0-8.9; "Critical": 9.0-10.0).

2.2.2 Static Vulnerability Detection Scheme

We use Clair, a widely used open source tool for static analysis of vulnerabilities in docker containers as an example of static vulnerability detection schemes. Clair works by scanning docker images and matching detected packages and their versions with a remote CVE database. Vulhub Vulhub: Docker-Compose File for Vulnerability Environment (2018) provides Dockerfiles for users to build vulnerable images. A Dockerfile is a script that contains all the commands that execute in succession to build container images. Dockerfiles in Vulhub use two different ways to install vulnerable applications, i.e., through the source code and by a package manager such as `apt-get install` or `dpkg install` to install a deb file. Vulnerable container images created from local Dockerfiles can be tagged and pushed to the Quay.io registry. Vulnerability scanning is automatically performed by Quay.io, and it takes about several minutes to produce the results. For each image pushed to the Quay.io registry, Clair scans the images and reports the total number of detected CVEs along with the distribution of the CVEs according to the severity rankings. For each reported CVE entry, Clair also lists a set of related information, e.g, the available CVSS score, package name, package version, and the suggestion of fixed versions of the vulnerable package. In addition, Clair also gives a hint of the specific layer where CVEs are introduced into images.

2.2.3 Dynamic Exploit Detection Approaches

Dynamic runtime detection schemes need to address two key issues: 1) what monitoring data to collect and how to extract proper features from the monitoring data? and 2) what algorithms to use for detecting vulnerability exploits?

Data Collection and Feature Extraction. The behaviors of running containers can manifest in different system metrics (e.g., CPU utilization, memory usage, and network traffic) or system calls. Although system metrics can be collected with low cost, they are heavily affected by dynamic application workloads, which makes them too noisy to be used as reliable data sources for container exploit detection. System calls are the interfaces though which applications access the services of the operating system. We observe that changes in the behaviors of containers from attempted attacks often manifest as variation in system call frequencies. For example, attempted attacks targeted at containers may introduce system calls which rarely appear during the applications' normal executions.

Table 2.2 An example of frequency vectors from a processed system call list.

System call / Timestamp	write	read	futex	epoll_wait
1516544689186	2	4	50	4
1516544689286	9	8	74	8
1516544689386	0	0	9	1

Our container system call logs are collected with a lightweight open source tracing tool called Sysdig Sysdig (2016). Sysdig supports container monitoring with transparent instrumentation, without the agent inside each container, which enables real-time analysis of container activities.

We extract proper features from the raw system call trace within equal sampling intervals. We explored both system call frequency and system call execution time features, which are called system call frequency vectors and system call time vectors, respectively. We formulate a frequency/time vector as $V(t) = [x_1, x_2, ..., x_n]$, where x_i represents the frequency or the execution time of each type of system call in a given sample interval. Table 2.2 gives an example of the extracted frequency vectors from a processed system call list. The first line represents the number of appearances that `sys_write`, `sys_read`, `sys_futex` and `sys_epoll_wait` calls make in the time interval $[t, t+100)$ milliseconds where $t = 1516544689186$.

After extracting proper features, we need to decide what algorithms we should use to detect vulnerability exploits. As mentioned in the introduction, container vulnerability detection needs to meet a set of new challenges. First, the detection algorithm cannot assume a large amount of training data because containers are often short-lived. Second, the detection algorithm cannot assume prior knowledge about either the application behavior or the attack behavior since containers are highly dynamic. Third, the detection algorithm needs to be able to provide real time detection with low overhead. To address these unique challenges of container vulnerability detection, we chose a set of light-weight unsupervised anomaly detection schemes to evaluate.

K Nearest Neighbors (k-NN): The k-nearest neighbors algorithm (k-NN) is used to perform outlier detection. Anomalies are those samples whose average distance to its nearest neighbors fall into the top p percentile. There is a trade-off between true positive rate and false positive rate when we adjust the k and p values. If we lower p, more samples will be identified as anomalous, which might increase both true positive rate and false

positive rate. The value of optimal k requires more sophisticated tuning algorithms. For container vulnerability exploit detection, it is impractical to tune the parameters on-the-fly so they can be empirically decided beforehand. In our experiments, we set k to be five and p to be 10%.

PCA + k-NN: One of the key challenges to achieve high accuracy in the k-NN algorithm lies in the presence of noise in the feature data (hundreds of different types of system calls). We choose Principal Component Analysis (PCA) as our dimension reduction strategy because PCA is fast and incurs low computation cost. In our experiments, we found that the magnitude of the top dimension is larger than that of the fifth dimension by four orders of magnitudes so we set the number of target dimensions to be five.

K-means: K-means is a traditional clustering method and easy to implement. K denotes the number of clusters of feature vectors. We consider clusters with a small number of samples, based on a cluster size threshold, as anomalous. Similar to k-NN, we can only empirically set the value of k to perform container vulnerability exploit detection.

Self-Organizing Map: Self-organizing map (SOM) Kohonen (1998) is a special kind of artificial neural network (ANN) which is able to reduce data dimensions and highlight similarities among data without imposing excessive learning overhead. The SOM algorithm preserves the relative distance between high dimensional data points so that points that are nearby in the input data are mapped to nearby neurons in the SOM.

We conduct training of the SOM network using the algorithm outlined by UBL Dean et al. (2012). A mapped neuron with a large neighborhood area value is far away from others and considered abnormal. The threshold is determined by a certain percentile value p of neighborhood area size. Intuitively, a low p value will make the detection more sensitive and raise more alerts.

2.3 Experimental Evaluation

In this section, we first describe our evaluation setup and then present our evaluation results in detail.

2.3.1 Experiment Setup

We set up victim containers in a virtual machine using Docker v17.05.0 in order to eliminate the interference brought by other activities in the host. The virtual machine is equipped with

2 GB memory and 40 GB disk, running 64 bit Ubuntu v16.04. Each victim container runs a vulnerable application associated with a specific CVE. The static vulnerability scanning is achieved by Clair v2.0.0. The syscall trace is collected using Sysdig v0.19.1.

To evaluate the effectiveness of each detection approach (e.g., real-time) and to restore the container practical usage scenarios (i.e. short-liveness), we collect system calls produced by the victim containers in a short period of time. Specifically, for each vulnerability, we first launch the victim container and start the vulnerable application. We then send workloads from the VM and start the Sysdig tracing module. Sysdig collects the system call traces for about six minutes, including the system calls produced by the application under normal workload and during the attack (i.e, from when the attack is triggered to when the attack succeeds). We then extract the time vectors and frequency vectors from the raw system call traces in samples of 100 milliseconds. We run different dynamic detection algorithms over those feature vectors.

2.3.2 Detection Results

Table 2.3 Detection Result of Clair and Anomaly Detection Approaches.

Threat Impact	CVE ID	CVSS Score	Clair	k-NN		PCA + k-NN		K-means		SOM time		SOM freq	
			Detected	Detected	FPR	Detected	FPR	Detected	FPR	Detected	FPR	Detected	FPR
Return a shell and execute arbitrary code	CVE-2015-8103	7.5	✗	✓	9.97%	✓	9.97%	✓	2.98%	✓	2.47%	✓	0.84%
	CVE-2017-7494	10.0	✗	✓	9.93%	✓	9.96%	✓	4.27%	✓	7.48%	✓	1.10%
	CVE-2016-10033	7.5	✗	✓	9.92%	✗	9.95%	✓	8.78%	✓	0.17%	✓	0.17%
	CVE-2015-2208	7.5	✗	✓	9.91%	✓	9.94%	✗	0.00%	✓	5.26%	✓	3.18%
	CVE-2016-9920	6.0	✗	✗	9.97%	✗	9.97%	✗	0.00%	✓	2.67%	✓	0.48%
	CVE-2015-1427	7.5	✗	✓	9.93%	✓	9.93%	✓	9.14%	✓	0.45%	✓	1.54%
	CVE-2014-3120	6.8	✗	✗	9.92%	✓	9.72%	✓	10.08%	✓	1.46%	✓	1.72%
	CVE-2012-1823	7.5	✗	✗	9.92%	✗	9.92%	✓	2.76%	✓	1.71%	✓	6.50%
	CVE-2017-11610	9.0	✗	✓	9.96%	✗	9.96%	✓	1.13%	✓	0.06%	✓	1.58%
	CVE-2017-8291	6.8	✗	✗	9.94%	✗	9.94%	✓	4.90%	✗	0.14%	✓	1.41%
	CVE-2015-3306	10.0	✗	✗	9.96%	✗	9.96%	✓	2.56%	✓	8.32%	✓	0.95%
	CVE-2017-12615	6.8	✗	✓	9.92%	✗	9.95%	✗	0.00%	✓	1.93%	✓	1.96%
	CVE-2016-3088	7.5	✗	✗	9.92%	✓	9.72%	✓	4.30%	✓	0.63%	✓	3.04%
	CVE-2017-12149	7.5	✗	✗	9.96%	✗	9.96%	✓	3.36%	✓	0.83%	✓	1.72%
	CVE-2015-8562	7.5	✗	✗	9.82%	✗	9.82%	✗	35.27%	✗	0.27%	✓	5.28%
Execute arbitrary code	CVE-2014-6271	10.0	✓	✗	9.97%	✗	9.97%	✗	1.60%	✗	4.64%	✓	0.42%
	CVE-2017-5638	10.0	✗	✗	9.95%	✓	9.65%	✓	4.09%	✓	0.84%	✓	3.17%
	CVE-2017-12794	4.3	✗	✗	9.95%	✗	9.95%	✓	8.90%	✗	0.55%	✗	3.10%
	CVE-2016-3714	10.0	✗	✗	9.97%	✗	9.97%	✓	1.06%	✓	0.36%	✓	0.26%
Disclose credential information	CVE-2017-7529	5.0	✗	✗	9.78%	✗	9.78%	✓	10.40%	✗	1.25%	✗	0.08%
	CVE-2015-5531	5.0	✗	✗	9.95%	✗	9.95%	✓	5.78%	✓	0.72%	✓	1.22%
	CVE-2014-0160	5.0	✓	✗	9.95%	✗	9.95%	✓	5.21%	✓	0.38%	✗	0.96%
	CVE-2017-8917	7.5	✗	✗	9.92%	✓	9.50%	✗	0.25%	✗	0.08%	✗	0.13%
Consume excessive CPU	CVE-2016-6515	7.8	✗	✗	9.97%	✗	9.97%	✓	1.02%	✓	6.73%	✓	3.65%
	CVE-2014-0050	7.5	✗	✗	9.92%	✓	9.72%	✓	6.30%	✓	2.01%	✓	1.97%
Crash the application	CVE-2016-7434	5.0	✗	✓	9.72%	✓	9.72%	✗	36.57%	✗	0.49%	✗	0.00%
	CVE-2015-5477	7.8	✓	✓	9.91%	✗	9.94%	✗	10.22%	✓	0.74%	✗	0.31%
Escalation of privilege	CVE-2017-12635	10.0	✗	✗	9.79%	✗	9.79%	✗	33.88%	✓	3.66%	✓	1.26%
Average Results			10.71%	32.14%	9.92%	35.71%	9.88%	67.86%	7.67%	75.00%	1.88%	78.57%	1.71%

We compare different vulnerability detection schemes using four metrics: 1) detection coverage: whether each approach can detect the vulnerabilities? 2) false positive rate: how accurate each approach can achieve for the detection? and 3) lead time: how quickly each approach can detect the attacks and thus prevent compromise in time?

Detection Coverage

Table 2.3 shows the detection coverage of different anomaly detection approaches. Overall, dynamic approaches achieve better detection coverage than the static approach. Specifically, SOM approaches achieve the highest detection coverage on average, followed by the K-means clustering approach. The k-NN and k-NN combined with PCA approaches achieve the lowest detection coverage among all dynamic approaches. The static approach (i.e., Clair) can only detect three out of 28 CVEs with the average detection coverage of 10.71%. The static approach can be utilized with a dynamic method to achieve the strengths of both techniques. Accordingly, the highest detection coverage results from combining the static and SOM frequency approaches. This pair can detect 24 out of 28 vulnerability cases, giving a detection coverage of 85.71%.

Clair achieves low detection coverage due to the lack of container image features (e.g., installed packages, package versions), or the incomplete remote vulnerability database. For example, Clair fails to detect the CVE-2017-7494 in the vulnerable docker image because vulnerable packages are installed using source code. Without using package managers to install vulnerable packages, e.g., `apt-get install`, Clair cannot extract the image features, thus it fails to detect the vulnerabilities by correlating the indexed features with remote vulnerability database. Another example is the CVE-2016-6515. Clair fails to detect this vulnerability due to the incomplete remote vulnerability database. In fact, Clair has extracted the container image feature (i.e. OpenSSH v1:7.2p2-4ubuntu2.1), but reports an incomplete list of vulnerabilities that threaten this image, e.g., CVE-2016-10009, CVE-2016-10012, CVE-2016-10010, CVE-2016-10011, CVE-2017-15906, and CVE-2016-8858.

The k-NN approach can only detect 32.14% vulnerabilities. It detects 7 out of 15 vulnerabilities that return a shell and execute arbitrary code and both the vulnerabilities that crash the applications, but fails to detect other types of vulnerabilities.

The k-NN combined with PCA approach achieves a slightly better detection coverage than the pure k-NN approach. It detects six out of 15 vulnerabilities that return a shell and execute arbitrary code, and another four vulnerabilities in different categories.

The K-means approach achieves 67.86% detection coverage by detecting 11 out of 15 vulnerabilities that return a shell and execute arbitrary code, 3 out of 4 vulnerabilities that execute arbitrary code, 3 out of 4 credential information disclosure vulnerabilities, two excessive CPU consumption vulnerabilities but it fails to detect any vulnerabilities which could crash the application or cause escalation of privilege.

The SOM approach over system call time vectors (SOM time) achieves the average detection coverage of 75% while the SOM approach over system call frequency vectors (SOM frequency) achieves the average detection coverage of 79%. In particular, they both can detect most or all of the vulnerabilities which would return an interactive shell and enable attackers to execute arbitrary code inside containers. One insight behind this is that system calls generated during the process of exploitation and the arbitrary code execution are distinct from those generated during applications' normal running process. For example, CVE-2014-3120 allows attackers to exploit a remote command execution (RCE) vulnerability in a vulnerable version of ElasticSearch (e.g., v1.1.1). We observed that certain system calls appear more frequently when the vulnerability is exploited (e.g., `sys_lseek`, `sys_mprotect`). We also found that specific system calls only appear after the attack is triggered (e.g., `sys_getuid`).

The K-means, SOM time and SOM frequency approaches achieve 100% detection coverage for the vulnerabilities which can cause performance issues (e.g., consume excessive CPU usage). For example, in CVE-2016-6515, the `auth_password()` function in OpenSSH before version 7.3 does not limit password lengths for password authentication, which allows remote attackers to launch a DoS attack via a long string, causing infinite loops. Another example is CVE-2014-0050, where attackers send a crafted content-type header to a vulnerable version of Apache Tomcat (e.g., v7.0-v7.0.50 and v8.0-v8.0.1), causing the loop index to be always less than or equal to the upper bound, hanging Tomcat endlessly.

False Positive Rate

Table 2.3 also shows the false positive rate of different anomaly detection approaches. Overall, the SOM approaches achieve the lowest false positive rate (1.7% for SOM frequency and 1.9% for SOM time). followed by the K-means clustering approach (7.67%). However, K-means approach has the largest FPR range from 0% to 36.57%. The k-NN and k-NN combined with PCA approaches incur the highest false positive rate (9.92% FPR and 9.88% FPR, respectively). However, these two approaches have the smallest FPR range from 9.5%

Table 2.4 The Lead Time of Anomaly Detection Approaches for CVEs that Return a Shell and Execute Arbitrary Code. "-": the approach does not detect the vulnerability.

Threat Impact	CVE ID	CVSS Score	k-NN (seconds)	PCA + k-NN (seconds)	K-means (seconds)	SOM time (seconds)	SOM freq (seconds)
Return a shell and execute arbitrary code	CVE-2015-8103	7.5	0	0	1	1	28
	CVE-2017-7494	10.0	0	1	1	28	35
	CVE-2016-10033	7.5	0	-	0	67	124
	CVE-2015-2208	7	0	1	-	1	1
	CVE-2016-9920	6.0	-	-	-	121	118
	CVE-2015-1427	7.5	4	4	0	2	7
	CVE-2014-3120	6.8	-	0	1	7	8
	CVE-2012-1823	7.5	-	-	0	44	45
	CVE-2017-11610	9.0	0	-	0	1	1
	CVE-2017-8291	9.0	-	-	0	-	1
	CVE-2015-3306	10.0	-	-	1	1	1
	CVE-2017-12615	6.8	0	-	-	12	5
	CVE-2016-3088	7.5	-	0	0	42	48
	CVE-2017-12149	7.5	-	-	0	8	8
	CVE-2015-8562	7.5	-	-	-	-	1
Average Lead Time			0.57	1.00	0.36	25.77	28.73

to 9.97%.

We omit the false positive rate result of Clair in our evaluation because Clair can report hundreds or thousands of CVEs for each victim container. It is extremely time-consuming to validate all of its detection results manually. It is also wrong to label all the CVEs identified by Clair but not included in our benchmark in Table 2.1 as false positives.

Lead Time

Table 2.4 shows the lead time achieved by different dynamic approaches for the CVEs with the thread impact of returning a shell to the attackers for executing arbitrary code. In those type of CVEs, the attackers require time-consuming operations to exploit the vulnerability such as traversing the vulnerable container to find the path of a specific writable folder (CVE-2017-7494), or creating a backdoor file in the root folder of container-side (CVE-2016-10033).

Overall, the SOM approaches achieve the largest detection lead time (28.7 seconds for SOM frequency and 25.8 seconds for SOM time). However, the other approaches' detection lead time is very low. Specifically, the k-NN combined with PCA approach achieves the average lead time of 1 second. The k-NN approach achieves the average lead time of 0.57 second. The worst case is the K-means approach which achieves a lead time of 0.36 second.

The results show that the SOM approaches are more practical than the other machine learning methods for real-time vulnerability detection. This time window is helpful because effective emergency measures can be taken by administrators to prevent the containers from being totally compromised.

We do not conduct lead time analysis for other CVE impact types such as crash of the application, because these attacks can finish immediately after the exploitation.

2.4 Summary

Emerging container techniques speed up deployments of applications and ease the distribution and delivery of software, but securing containers still has a long way to go toward maturity. In this chapter, we conduct a study to evaluate the effectiveness of different static and dynamic vulnerability exploit detection schemes for container hosted applications. Our initial experiments using 28 real world vulnerabilities discovered in 24 commonly used server applications show that static vulnerability scanning of container images alone is insufficient, which only detects 3 out of 28 vulnerabilities. Dynamic anomaly detection schemes using unsupervised machine learning methods can effectively detect 22 vulnerability exploits with low false positive rates. Combining static and dynamic schemes can further increase the detection coverage to 86% (i.e., 24 out of 28 vulnerabilities). Our experiments are still preliminary. In our future work, we plan to extend our vulnerability cases and further improve the detection accuracy by combining and augmenting our vulnerability detection schemes.

CHAPTER

3

SELF-PATCH: BEYOND PATCH TUESDAY FOR CONTAINERIZED APPLICATIONS

3.1 Introduction

Containers have become increasingly popular in distributed computing environments by providing an efficient and lightweight deployment method for various applications. However, recent studiesDocker Image Vulnerability Research (2017)Shu et al. (2017) have shown that containers are prone to various security attacks, which has become one of the top concerns for users to fully adopt container technology Bettini (2015).

Containerized applications pose a set of new security challenges to distributed computing environments. First, container image repositories are prone to vulnerabilities. Indeed, previous study Shu et al. (2017) reveals an alarming degree of vulnerability exposure and spread in the official Docker Hub container repository. It is complex to maintain a public or private container repository which often consists of a large number of container images and many inheritance layers. If a container is created from a base image, any vulnerability

included in the base image needs to be patched in the containers that are built on top of the base image. Second, containers are often allocated with limited resources because a large number of containers often share the resources of a single physical host. Security patching might cause significant resource increase (e.g., memory bloating) in a patched container, which makes the container unable to run after patching.

Existing security patching schemes in distributed computing environments often follow a periodically scheduled whole upgrade approach, that is, updating all applications as a whole on a certain day (e.g., every Tuesday). The approach works well in stable systems consisting of long running applications. However, containers are often short-lived, which makes periodical patching schemes ineffective if the vulnerable containers miss the pre-scheduled patching day. Moreover, whole software upgrade often significantly increases the memory and storage footprint of the patched containers. As a result, those containers quickly become too heavy to fit in constrained resource allocations.

In this chapter, we present Self-Patch, an intelligent self-triggering security patching framework for containerized applications. Our framework consists of three integrated components: 1) online *attack detection* module which can detect security attacks using low-cost, non-intrusive system call tracing and unsupervised autoencoder neural network models Schmidhuber (2015); 2) *attack classification* module which classifies attack behaviors into specific vulnerability exploits by identifying most frequently appeared system calls during the attack period and 3) *targeted patch execution* module which is responsible for applying proper security patches based on the classification results. Specifically, this chapter makes the following contributions.

- We present a new self-triggering targeted patching framework to achieve effective and efficient attack containment for containerized applications.
- We describe an online attack detection and classification scheme using out-of-box system call tracing and unsupervised machine learning methods.
- We have implemented a prototype of Self-Patch and evaluated it over 31 real world security attacks in 23 commonly used server applications.

Our experimental results show that Self-Patch's attack detection scheme can accurately detect and classify 81% security attacks with 16 seconds lead time on average. In comparison, other commonly used anomaly detection schemes such as k-nearest neighbor (k-NN) and

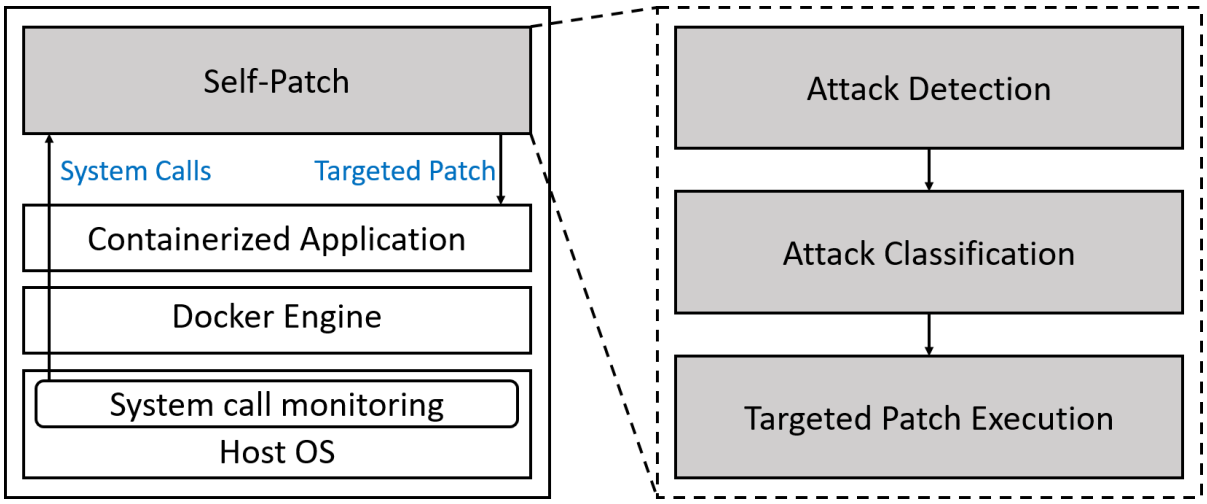

Figure 3.1 System overview of Self-Patch.

k-means clustering algorithm can only detect 6% and 68% exploits, respectively. k-means also produce 7% false alarms while Self-Patch only incurs 0.7% false alarms. We further compare the memory and disk footprint change before and after patching between Self-Patch and the existing whole upgrade approach. Our results show that Self-Patch can reduce the memory footprint increase (caused by the applied patches) by up to 84% and disk size increase by up to 40%.

The rest of the chapter is organized as follows. Section 3.2 describes the system design. Section 3.3 presents our experiment setup and results. Section 3.4 concludes the chapter.

3.2 System Design

This section describes the system design of Self-Patch. We first provide an overview about the system, followed by the design details for each component.

3.2.1 System Overview

Self-Patch aims at providing a self-triggering targeted patching framework for containerized applications, which is illustrated by Figure 3.1. Self-Patch consists of three coordinating components: 1) *attack detection*, 2) *attack classification*, and 3) *targeted patch execution*. The attack detection module monitors container runtime behaviors by analyzing system

Table 3.1 A frequency vector sample for the ActiveMQ application (CVE-2016-3088). An attack is triggered at $t = 1528903079912$.

System call / Timestamp	access	sendto	lseek	fcntl
1528903079812	0	0	0	0
1528903079912	59	4	0	5
1528903080012	299	18	2	0
1528903080112	0	0	0	0

call traces via unsupervised autoencoder neural network learning methods. We pick system call data for our attack detection because many attacks manifest in system call invocations. We decide to use an unsupervised machine learning method in order to achieve online detection for both known and unknown attacks.

Upon detecting anomalies in container runtime behavior, we need to decide what type of vulnerability the detected attack targets on. The attack classification component extracts top frequently used system calls during the attack period. To map to a specific vulnerability, we perform offline profiling to extract the vulnerability signature by extracting the top frequently used system calls after triggering the corresponding attack. Note that we make an assumption here that attacks targeting the same vulnerability exhibit the same behavior in terms of top frequently used system calls. We find our assumption holds in our experiments. We plan to further validate this assumption using more attacks in our future work.

After the vulnerability is identified, the targeted patch execution module is dynamically triggered to contain the attack by patching the victim container to fix the vulnerability targeted by the attack. We first bring the victim container offline and then apply the proper software updates to the container in a quarantined environment. Once the patching is complete, an updated container image is committed to the repository for spawning future containers. We now present the design details of each component in the following subsections.

3.2.2 Attack Detection

Self-Patch performs attack detection by analyzing system call traces invoked by the containerized application. For robustness, we leverage an existing container monitoring tool Sysdig Sysdig (2016) to achieve out-of-box monitoring from the host kernel. We can collect

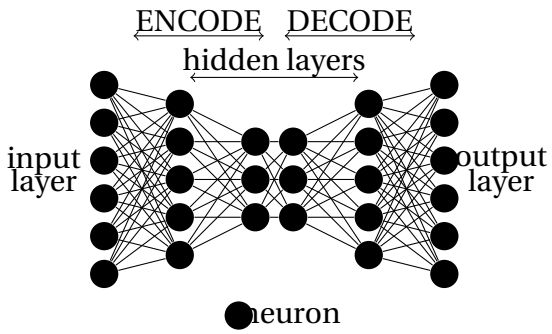

Figure 3.2 Architecture of the autoencoder.

all the system calls invoked by an application running inside the container from outside. Although system call sequences (i.e., n-gram) have been used to identify attacks in intrusion detection systems Forrest et al. (1996), they require a database of recognized sequences for detecting attacks, which cannot handle dynamic workload and mimicry attacks. In this work, we propose to first extract system call features to model runtime application behaviors and then apply unsupervised machine learning methods to detect attack behaviors. Specifically, we compute the frequencies of each system call type per sampling period to form a frequency vector. For example, Table 3.1 shows a frequency vector time series where the `access` system call is invoked 299 times within [1528903080012, 1528903080112) millisecond. Intuitively, when an attack is triggered to exploit a certain vulnerability in the application, certain types of system calls are invoked more frequently than normal. For example, in Table 3.1, we can see an abnormal frequency increase for `access` and `sendto` system calls after the attack is triggered at time 1528903079912.

To achieve online anomaly detection, we leverage unsupervised multi-variate machine learning to detect abnormal system call frequency changes. We choose autoencoder neural network as our detection model because it does not require labelled training data and can achieve good accuracy with a relatively small number of neurons with low training cost. The autoencoder neural network builds a model that learns to reconstruct training samples with minimal error. This is achieved by representing the input data in a lower dimensional space with a small number of neurons, that is called the *encode* step. Thereafter, in the *decode* step, the model attempts to regenerate the data that was compressed by the encode step. Thus, the autoencoder network typically has a symmetric architecture as depicted in Figure 3.2. The figure shows the network configuration of our autoencoder implementation

with four hidden layers between the input and output layers. In the encode region, the network is fully connected from the input layer to the following two hidden layers. The encode region is fully linked to the identical but reversed decode region by their innermost layers of neurons. The length of the vectors is the number of different system call types (e.g., read, write, futex) produced by the application. The number of neurons in the input layer and output layers of the autoencoder is determined by the number of different system call types appeared in the system call training data. The weights between neurons are updated with the sigmoid activation function. Furthermore, the mean squared error (MSE) is the loss function which is minimized via back propagation.

The model classifies test samples with low reconstruction errors as normal and those with high reconstruction errors as abnormal. We determine the error threshold based on the reconstruction errors observed in the training data. Specifically, a certain percentile rank of reconstruction errors from the training set is selected as the threshold. We choose 99 percentile value as our threshold in our experiments. We found the 99 percentile threshold works well in our experiments.

3.2.3 Attack Classification

After detecting an attack, we want to classify the attack to into a specific type which is linked to a vulnerability identifier (e.g. CVE ID). Once this is obtained, we can update the application to the proper version. Similarly, signatures for new attacks would also be generated, which can contribute to the development of new security updates.

The attack classification in our system is guided by the alarms raised by our detection models. Based on the detected attack period, we extract the top ranked system calls with the following algorithm. The rank is calculated by taking an average of the frequency counts for each system call during the interval. The list of system calls and their counts, sorted in descending order, serves as the rank. To extract signature patterns, we first identify the top k ranked system calls (e.g., $k = 5$) and then concatenate the names of those selected system calls into a string. The attack signature is denoted by the hash value from running a Secure Hash Algorithm (SHA) on the string. For example, let us consider the denial of service (DOS) attack to Network Time Protocol (NTP) vulnerability (CVE-2016-7434). The top five frequent system calls are: `rt_sigprocmask`, `gettid`, `write`, `read` and `clock_gettime`. The `rt_sigprocmask` system call checks or modifies the blocked signals of a thread, while `gettid` gets the thread ID. Furthermore, the denial of service attack is

```
#!/bin/bash
# download files
apt-get update
apt-get -y install wget gcc make
wget https://github.com/.../ghostscript-x.xx.tar.gz
tar xvf ghostscript-x.xx.tar.gz

# install files
cd ghostscript-x.xx
./configure
make install

# remove files
apt-get purge -y wget gcc make
apt-get autoremove -y
cd ..
rm -r ghostscript-x.xx.tar.gz ghostscript-x.xx
```

Figure 3.3 A targeted patching example for Ghostscript.

caused by sending an extremely long character to the NTP service over a socket connection. Thus, the application would need to make read and write calls to service this request. The *clock_gettime* call retrieves the time of a requested clock.

The signature is then mapped to a corresponding existing CVE ID that is collected by an offline profiling process using the same signature extraction algorithm. However, if we fail to map the signature with any existing CVE, we mark this attack as an unknown attack which requires further investigation.

3.2.4 Targeted Patch Execution

After a specific attack is detected and classified, Self-Patch triggers the targeted patching module over the victim container to contain the attack. The targeted patch execution module focuses on installing only the specific libraries needed to address the identified vulnerability. Our patching process consists of three steps: downloading new software packages, installing new software packages, and removing unnecessary files.

Obtaining source files involves using tools such as *wget* or *git* or APT, depending on where the files are located. Wget is useful for downloading files provided by a URL (Uniform

Resource Locator), `git clone` for GitHub repositories and `apt-get update` for retrieving packages provided by APT.

Installation may require other tools like *make* or *pip* to compile and install the application. Applications downloaded from source with a Makefile are typically installed with `./configure` to prepare a Makefile, followed by `make` to compile source code and finally a `make install` to move the compiled files to proper locations. Those applications configured with APT can leverage `apt-get install -only-upgrade` commands, whereas those with pip can use `pip install` which handles both the download and install steps. Applications that escape the above efforts, may be be supplemented with files from third party entities such as Personal Package Archives (PPA) supervised by Ubuntu.

Finally, the installation is cleaned up. Downloaded archive source files and folders extracted from them as well as their outdated counterparts can be removed with basic Linux commands. APT can handle this process with `apt-get purge` and `apt-get autoremove` commands.

Figure 3.3 presents a basic targeted patch example for three Ghostscript vulnerabilities (CVE-2018-16509, CVE-2018-19475 and CVE-2019-6116). In the download files section, dependent *wget*, *gcc* and *make* libraries are retrieved with APT to execute the rest of the installation procedure. Whereas, *wget* downloads the tar archive that contains the new application version source files. Notice that after the file is installed, these files and packages are removed.

The difficulty in the patch execution lies in the installation differences among applications. Discovering these libraries and installation steps involves extensive searches over security databases, application sites and manuals. We compare results of this targeted method of applying updates to the alternate periodical update approach in Section 3.3.2.

Targeted patching is applied to the container in a quarantined environment isolated from other normal applications. Meanwhile, various security countermeasures can be applied. For instance, further requests from the compromised container can be dropped while new trusted containers are spawned to replace compromised ones. After a successful update, an image is saved from the container with a `docker commit`. The resulting image is then used to deploy new containers.

Table 3.2 List of explored real-world vulnerabilities.

Threat Impact	CVE ID	CVSS Score	Application	Attack Duration (seconds)
Return a shell and execute arbitrary code	CVE-2012-1823	7.5	PHP	1
	CVE-2014-3120	6.8	Elasticsearch	9
	CVE-2015-1427	7.5	Elasticsearch	60
	CVE-2015-2208	7.5	phpMoAdmin	2
	CVE-2015-3306	10	ProFTPd	4
	CVE-2015-8103	7.5	JBoss	30
	CVE-2016-10033	7.5	PHPMailer	125
	CVE-2016-3088	7.5	Apache ActiveMQ	49
	CVE-2016-9920	6	Roundcube	121
	CVE-2017-11610	9	Supervisor	2
	CVE-2017-12615	6.8	Apache Tomcat	13
	CVE-2017-7494	10	Samba	36
	CVE-2017-8291	6.8	Ghostscript	1
Execute arbitrary code	CVE-2014-6271	10	Bash	2
	CVE-2015-8562	7.5	Joomla	1
	CVE-2016-3714	10	ImageMagick	4
	CVE-2017-12794	4.3	Django	1
	CVE-2017-5638	10	Struts	29
	CVE-2018-16509	9.3	Ghostscript	2
	CVE-2018-19475	6.8	Ghostscript	2
	CVE-2019-6116	6.8	Ghostscript	2
Disclose credential information	CVE-2014-0160	5	OpenSSL	14
	CVE-2015-5531	5	Elasticsearch	2
	CVE-2017-7529	5	Nginx	1
	CVE-2017-8917	7.5	Joomla	1
	CVE-2018-15473	5	OpenSSH	2
Consume excessive CPU	CVE-2014-0050	7.5	Apache Tomcat	45
	CVE-2016-6515	7.8	OpenSSH	20
Crash the application	CVE-2015-5477	7.8	BIND	6
	CVE-2016-7434	5	NTP	1
Escalate privilege level	CVE-2017-12635	10	CouchDB	1

3.3 Experimental Evaluation

In this section, we present our evaluation methodology and experimental results. We implement a prototype of Self-Patch and conduct the experiments using machines running on 64-bit Ubuntu 16.04. Each machine is equipped with a dual-core 2.6 GHz CPU along with 4 GB memory.

3.3.1 Evaluation Methodology

Real world vulnerabilities

We evaluate Self-Patch using 31 real world vulnerabilities discovered in 23 commonly used server applications, highlighted in Table 3.2. We especially focus on vulnerabilities of medium to high severity reported in the last five years. These applications include widespread web services (e.g. Apache Tomcat, Nginx, Elasticsearch) and database services (e.g. CouchDB), which are recently popular Carter (2018), Datadog (2018). Attacks to application vulnerabilities result in threat impacts that fall into six categories classified by a recent study Shu et al. (2017): 1) return a shell and execute arbitrary code; 2) execute arbitrary code; 3) disclose credential information; 4) consume excessive CPU; 5) crash the application and 6) escalate privilege level.

Experiment setup

We run workload generated with Apache JMeter[1] on the container of each target vulnerability, to approach real world system operation. Specifically, JMeter quickly delivers appropriate requests to the applications. The supplied request rate increases to the maximum value that the application can accommodate. After running the container for a period of normal operation, enough to train the detection model, an attack is triggered to exploit the security vulnerability. The attack then executes for a subsequent period until no further attack activity is made. Meanwhile, we use Sysdig Sysdig (2016) to record the system calls invoked by the running containerized applications. We separate the entire system call trace into two halves. We use the first half of the data to train the detection model as it consists of enough samples of normal operation. However, we use the whole trace to extract detection and attack signature results.

To evaluate the results of targeted patching, we repeat the above process. Immediately after the vulnerability is triggered, we execute the targeted patching. At the same time, we monitor the memory utilization and disk size. In particular, we track the memory usage by leveraging the APIs exposed by cAdvisor[2]. Meanwhile, the container disk size is collected with native Docker commands. The sizes of the read-only image layers and writable container layers are summed and given by the `docker ps -s` command.

[1]Apache Jmeter can be found at `https://jmeter.apache.org/`
[2]cadvisor available at `https://github.com/google/cadvisor`

Alternative Schemes

To evaluate Self-Patch's performance, we compare it with several baseline methods. Self-Patch consists of three phases, i.e., attack detection, attack classification, and targeted patch execution. For the attack detection phase, we compare Self-Patch against k-nearest neighbors (k-NN) Altman (1992) and k-means Kanungo et al. (2002) techniques. For the targeted patch execution phase, we evaluate our approach against the *whole upgrade* method. We describe each alternative method in detail below.

k-NN for anomaly detection: We take the system call frequency vectors as input and return the outliers at their corresponding timestamps. The k-NN algorithm typically involves assigning a label to a data point based on the majority vote from its k closest neighbors. Abnormal samples are those too far away from their neighbors. We calculate the average distance of each point to its nearest neighbors and determine the anomalous ones with larger distances. In our experiment, we empirically select k as 5. In addition, we choose the samples with the top 10% largest average neighbor distance as the outliers.

k-means for anomaly detection: We customize the k-means algorithm for the anomaly detection phase in a similar way to the k-NN algorithm. To be specific, data samples are distributed to one of the k randomly initialized cluster centers. Thereafter, the cluster centers are recalculated based on the average position of its members and then cluster memberships are reassigned. This process is performed iteratively until no more change occurs. Here, the algorithm identifies abnormal samples as those that belong to isolated clusters with a little membership. The tuning process is similar to that of k-NN. We observe the detection and false positive rate while varying k and keeping the cluster size threshold constant and vice versa. The resulting number of clusters is equal to ten ($k = 10$). Meanwhile, the cluster threshold for determining anomalous clusters is set to 100, which is chosen to achieve a good tradeoff between true positive rate and false positive rate in our experiments.

Whole upgrade for patch execution: This approach refers to the conventional manner in which security updates are performed in Debian-based Linux systems that containers run on. Old versions of all packages found by the package manager are updated to their newest versions. In our study, this is accomplished by an `apt-get update` followed by an `apt-get upgrade` of the APT package manager. The update command refreshes the package source lists to find the latest available packages while the latter installs the newly found software versions. We also employ the corresponding commands for containers based on Alpine Linux (i.e. replacing `apt` with `apk`).

3.3.2 Results and Analysis

In this subsection, we discuss our experimental results. Firstly, we discuss the results of detecting attacks. Then we analyze the results of signatures extracted from the system. Lastly we compare the cost of targeted patching by Self-Patch with that of whole upgrade.

Attack Detection Results

We present the detection results of Self-Patch over three evaluation metrics, i.e., true positive rate, false positive rate (FPR) and lead time.

For detection coverage, we measure whether the attack is detected by checking whether the alarm is raised after the attack is triggered and *before* the attack is successful. The detection coverage is also referred as true positive rate (TPR) in this chapter calculated by the following standard true positive rate equation, where TP is number of attacks that are detected and FN is the number of attacks that are undetected.

$$TPR = \frac{TP}{TP + FN} \tag{3.1}$$

Next, we use the standard false positive rate FPR as the second evaluation metric. FP represents the number of false alarms and TN represents the number of normal data samples that Self-Patch correctly does not generate alarms on.

$$FPR = \frac{FP}{FP + TN} \tag{3.2}$$

Lastly, we assess detection performance using lead time as the third metric. Lead time is defined to be the amount of time between the first alert from the detector after the malicious command is executed and completed. This represents the amount of flexibility the system has to initiate security countermeasures before the container is fully compromised.

Table 3.3 shows the detection results for each detection approach (i.e. k-NN, k-means and Self-Patch). The detection coverage results show that k-NN performs much more poorly than k-means and Self-Patch. k-NN detects 2 of 31 attacks (6.45%), whereas, k-Means and Self-Patch detection recognize 21 (67.74%) and 25 (80.65%), respectively. Self-Patch also demonstrates superior performance with a lower average FPR of 0.72% than the 7.16% k-Means result. The average lead time of Self-Patch is the longest (16.38 seconds), compared with both k-means (13.53 seconds) and k-NN (0.15 seconds). Although the attacks have

Table 3.3 Detection result of Self-Patch and alternative approaches.

Threat Impact	CVE ID	CVSS Score	k-NN	k-Means	Self-Patch
Return a shell and execute arbitrary code	CVE-2012-1823	7.5	✗	✗	✗
	CVE-2014-3120	6.8	✗	✓	✓
	CVE-2015-1427	7.5	✓	✓	✓
	CVE-2015-2208	7.5	✗	✗	✓
	CVE-2015-3306	10	✗	✓	✓
	CVE-2015-8103	7.5	✗	✓	✓
	CVE-2016-10033	7.5	✗	✓	✓
	CVE-2016-3088	7.5	✗	✓	✓
	CVE-2016-9920	6	✗	✗	✓
	CVE-2017-11610	9	✗	✓	✓
	CVE-2017-12615	6.8	✗	✗	✓
	CVE-2017-7494	10	✗	✓	✓
	CVE-2017-8291	6.8	✗	✗	✓
Execute arbitrary code	CVE-2014-6271	10	✗	✗	✓
	CVE-2015-8562	7.5	✗	✓	✗
	CVE-2016-3714	10	✗	✓	✓
	CVE-2017-12794	4.3	✗	✓	✗
	CVE-2017-5638	10	✗	✓	✓
	CVE-2018-16509	9.3	✗	✗	✓
	CVE-2018-19475	6.8	✗	✗	✓
	CVE-2019-6116	6.8	✗	✗	✓
Disclose credential information	CVE-2014-0160	5	✗	✓	✓
	CVE-2015-5531	5	✗	✓	✓
	CVE-2017-7529	5	✗	✓	✗
	CVE-2017-8917	7.5	✗	✓	✓
	CVE-2018-15473	5	✗	✓	✓
Consume excessive CPU	CVE-2014-0050	7.5	✗	✓	✓
	CVE-2016-6515	7.8	✗	✓	✓
Crash the application	CVE-2015-5477	7.8	✗	✗	✗
	CVE-2016-7434	5	✓	✓	✗
Escalate privilege level	CVE-2017-12635	10	✗	✓	✓
Average Results			6.45%	67.74%	80.65%

varied attack periods, noted in Table 3.2, Self-Patch more consistently yields higher lead time

We express the detection coverage, FPR and lead time of each method over the attacks in each threat impact category in Figure 3.4, 3.5 and 3.6, respectively. Figure 3.4 shows that Self-Patch achieves the highest detection coverage in all but two categories: *disclose credential information* and *crash the application*. Although k-means outperforms Self-Patch in these areas, it suffers from a much higher false positive rate. Furthermore, Self-Patch as well as the other detection approaches struggle with attacks that crash the application. This is likely because the crash causes the container end abruptly and lose data before an alarm is confidently raised. We plan to improve the accuracy of Self-Patch in future work with strategies that leverage system call arguments.

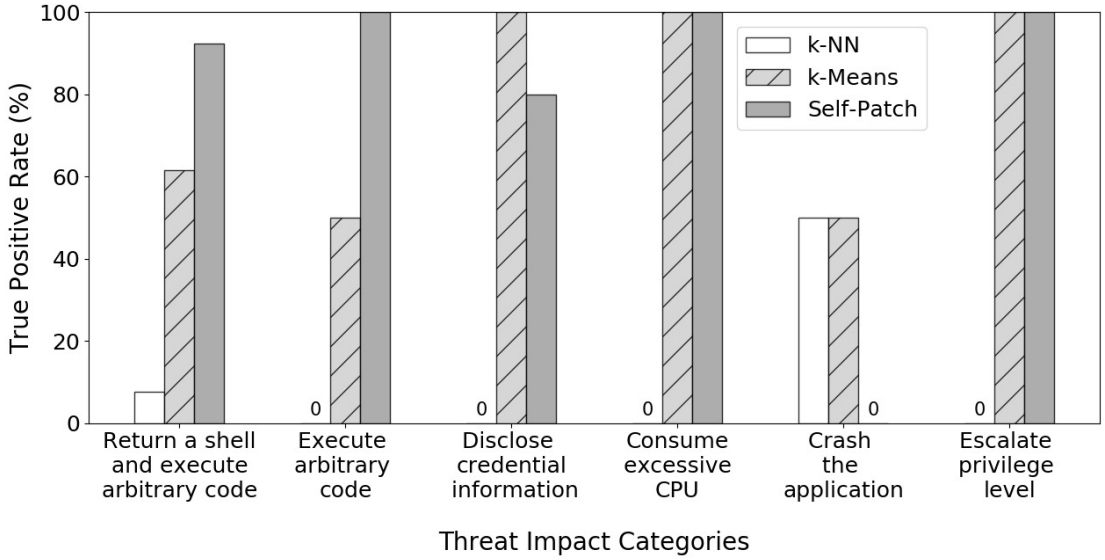

Figure 3.4 True positive rate result of anomaly detection approaches.

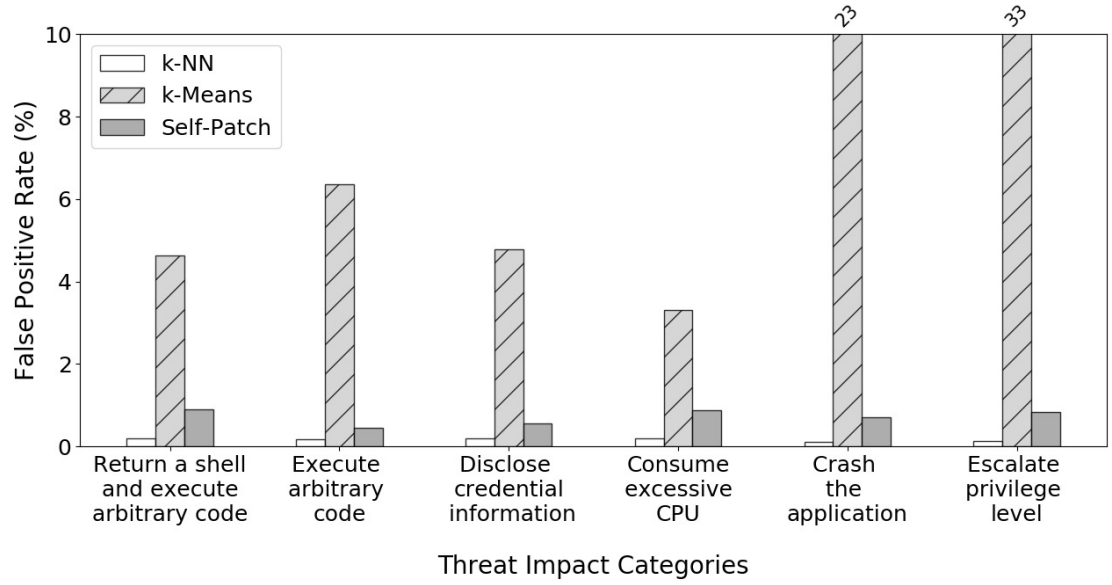

Figure 3.5 False positive rate result of anomaly detection approaches.

Attack Classification Results

Table 3.4 summarizes the patterns that correspond to attacks on each vulnerability, including the underlying top system calls for each pattern entry. We observe that Self-Patch

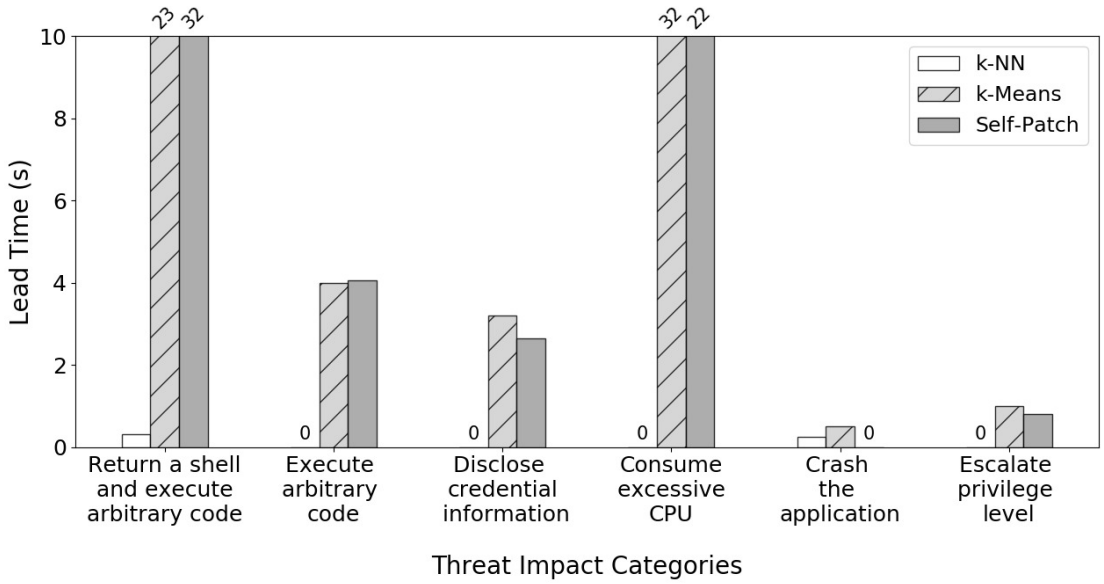

Figure 3.6 Lead time result of anomaly detection approaches.

produces unique patterns for 29 of the 31 CVEs. In particular, the duplicates are observed among three of four containers of the GhostScript application used for image processing. However, other applications with multiple containers of distinct vulnerabilities do not yield identical signatures. The GhostScripts attacks exploiting CVE-2018-16509, CVE-2018-19475 and CVE-2019-6116 are of a similar fashion. They involve uploading vulnerable image files embedded with malicious content to bypass the GhostScript security sandbox and execute commands. Thus, one can expect similar behavior from these attacks. Regardless, the GhostScript vulnerabilities can all be addressed by the same targeted patch.

Patching Results

We discuss the patching results, including success status and patching costs, i.e., memory and disk costs. The patching results are shown in Table 3.5, Figure 3.7 and Figure 3.8.

First, we describe how we determine whether patching is successful or not. After the completion of each patching experiment, we save the image of the container for future testing. We then start a new container using the image just created and then execute the attack commands. If the commands continue to work as before, we mark this patching experiment as unsuccessful, otherwise, we mark it as successful. Targeted patching by Self-Patch achieves 80.65% success rate. Note that this is a 100% of the cases where the attack

Table 3.4 Top system call composition of generated patterns.

CVE ID	Pattern	Top System Calls				
CVE-2012-1823	24355	futex	switch	epoll_wait	write	read
CVE-2014-3120	770ab	stat	lstat	setitimer	switch	read
CVE-2015-1427	81b6c	futex	switch	stat	read	lseek
CVE-2015-2208	ffffe	switch	select	nanosleep	read	stat
CVE-2015-3306	e0eaa	futex	switch	stat	sched_yield	close
CVE-2015-8103	81dbf	read	mmap	close	writev	lstat
CVE-2016-10033	a8c13	switch	futex	epoll_wait	read	stat
CVE-2016-3088	e01e9	write	fcntl	geteuid	getegid	switch
CVE-2016-9920	a53d4	read	switch	stat	close	poll
CVE-2017-11610	f58fe	stat	lstat	read	close	switch
CVE-2017-12615	a0ea6	futex	switch	stat	close	epoll_wait
CVE-2017-7494	f23a4	futex	switch	epoll_wait	read	sched_yield
CVE-2017-8291	634c5	mmap	close	open	fstat	switch
CVE-2014-6271	10b79	lstat	fstat	access	close	open
CVE-2015-8562	5f2a6	switch	read	stat	close	mmap
CVE-2016-3714	139c2	read	lseek	open	switch	futex
CVE-2017-12794	15862	stat	switch	poll	sendto	futex
CVE-2017-5638	26a4a	lstat	stat	setitimer	fcntl	read
CVE-2018-16509	a060e	fcntl	setitimer	stat	chdir	sendto
CVE-2018-19475	a060e	fcntl	setitimer	stat	chdir	sendto
CVE-2019-6116	a060e	fcntl	setitimer	stat	chdir	sendto
CVE-2014-0160	22184	epoll_wait	switch	close	writev	write
CVE-2015-5531	0c129	switch	futex	epoll_wait	read	mprotect
CVE-2017-7529	7ea25	switch	poll	stat	writev	read
CVE-2017-8917	22c99	fstat	lstat	access	close	open
CVE-2018-15473	2182c	read	lseek	open	switch	stat
CVE-2014-0050	6052e	xtensa	ioctl_console	ioctl_iflags	ioctl_getfsmap	ioctl_fideduperang
CVE-2016-6515	9cbbe	close	mmap	read	open	fstat
CVE-2015-5477	b4736	epoll_wait	sched_yield	close	switch	futex
CVE-2016-7434	4daa5	rt_sigprocmask	gettid	write	read	clock_gettime
CVE-2017-12635	ad14b	read	close	fstat	mmap	open

Table 3.5 Overall comparison result of different patching approaches.

Patching Approach	Success Rate	Memory Cost	Disk Cost
Whole Upgrade	6.45%	10.13x	1.49x
Self-Patch	80.65%	4.79x	1.16x

was detected by Self-Patch. However, only 6.45% of whole upgrade trials are successful. This demonstrates the superiority of targeted patching over periodic updates.

The reason why whole upgrade achieves such low success rate is because many containers are not configured with package managers. On one hand, there are programs whose developers have not prepared the files that will be handled by the manager (e.g., debian files for APT). Users of the software have no choice but to install via other provided means. Some of such applications or libraries we encountered include JBoss, Joomla, Roundcube and phpMoAdmin. On the other hand, when developers provide diverse ways of installa-

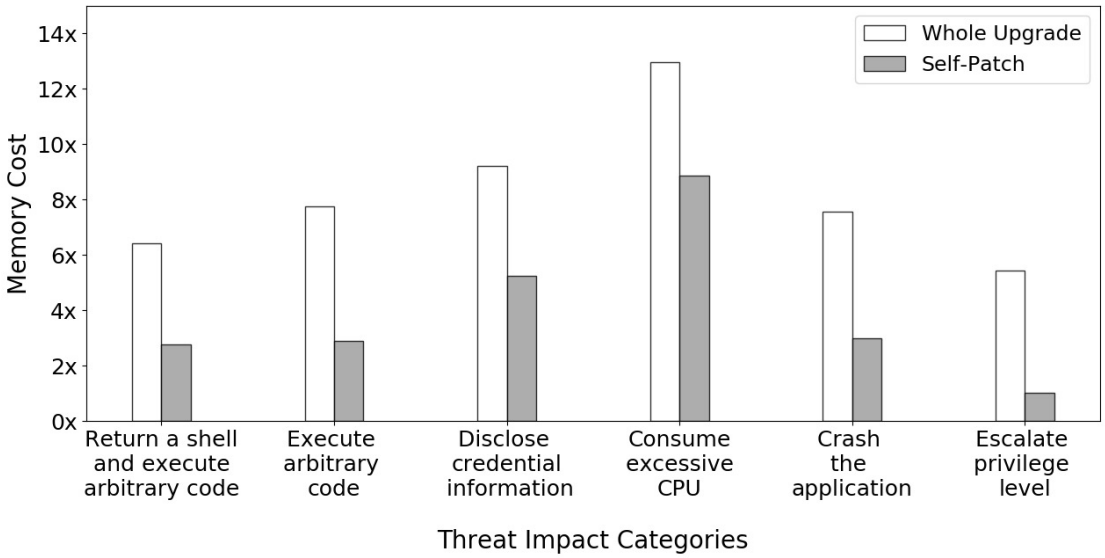

Figure 3.7 Container memory cost of patching.

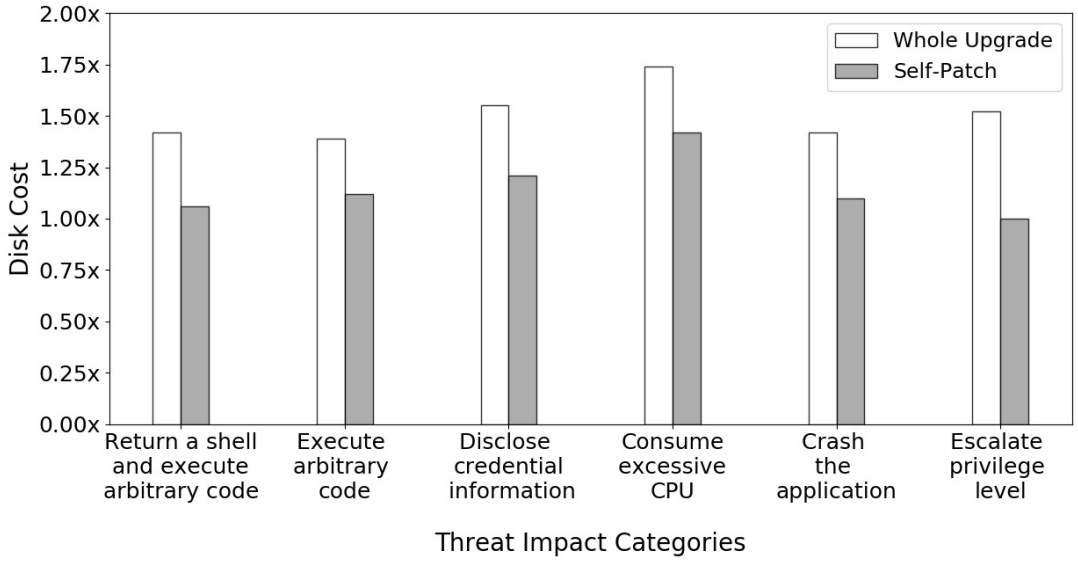

Figure 3.8 Container disk cost of patching.

tion (e.g. from source, binaries, etc), then users can choose to install based on preference. Software development and IT teams, drawn to containers for the fundamental guarantee of a consistent application environment and behavior may easily package their applications from source files to avoid accidental upgrades. Applications built for archival purposes

such as for showcasing vulnerabilities may also elect to install via downloaded source files. Therefore, for applications not managed by APT, the periodic update process will upgrade libraries other than those needed to address the vulnerability of the containerized application in question.

In addition, the patching cost results show lower memory and disk size footprint when performing targeted patching rather than the whole upgrade approach. On average, the size grew to 4.79 times its original size with targeted patching versus 10.13 with whole upgrade in memory size. Similarly, targeted patching multiplied the disk size by a factor of 1.16, whereas whole upgrade increased the size by a factor of 1.49. Nevertheless, the following other cases are encountered. Slightly higher memory size resulted after targeted patching for one vulnerable application, OpenSSH (CVE-2018-15473). It is important to recall that the whole upgrade approach did not successfully update the target application for most entries, including OpenSSH. Finally, the whole upgrade memory size is slightly higher than that of the targeted patch in Bash (CVE-2014-6271), although the disk size is the same. This is attributed to both methods installing the same single Bash library for the update.

3.4 Summary

In this paper, we have presented Self-Patch, a new self-triggering targeted patching framework for container-based distributed computing environments. Self-Patch aims at providing effective and efficient solutions to protect containerized applications from security attacks. To achieve this goal, the Self-Patch framework consists of three coordinating components: 1) an online attack detection module which can dynamically detect abnormal attack activities by extracting feature vectors from system call traces and applying unsupervised machine learning methods over the extracted features; 2) an attack classification scheme which classifies a detected attack into a specific type linked to a certain CVE; and 3) a targeted patch execution module which can install proper software patches to fix the vulnerability. We have implemented a prototype of Self-Patch and evaluated it over 31 real-world vulnerabilities discovered in 23 common server applications. Our initial experimental results are promising, which shows we can increase detection rate to over 80% and reduce false alarm rate to 0.7%. In contrast, traditional schemes can either only detect 6% attacks or incur more than 20% false alarms. Compared to the whole software upgrade approach, Self-Patch can reduce the memory overhead by up to 84% and disk overhead by up to 40%.

CHAPTER

4

UNDERSTANDING SECURITY VULNERABILITIES IN CLOUD SERVER SYSTEMS

4.1 Introduction

Cloud servers provide a cost-effective platform for deploying software applications in a pay-as-you-go fashion. However, due to its multi-tenant sharing nature, the cloud environment is especially vulnerable to security attacks. Due to its widespread deployment, any security vulnerability in cloud server systems can cause extensive impact on the end users Shu et al. (2017). For instance, vendors of the popular Java logging library, Apache Log4j, reported a serious vulnerability on December 9, 2021, affecting industries worldwide Wetter and Ringland (2021); Korn (2021). The vulnerability, named Log4Shell, allowed attackers to execute any commands in cloud systems that contained the library, resulting in about 200,000 global attacks within one day of the disclosure Ltd (2021). The open source insights

Figure 4.1 The Apache Log4j CVE-2021-44228 bug (CVSSv3: 10.0, CVSSv2: 9.3). The vulnerable function *lookup* does not restrict the lookup of JNDI URIs before instantiating the requested class with the security-sensitive *getObjectFactoryFromReference* function. This 'improper execution restrictions' bug has the 'execute arbitrary code' impact.

team from Google Cloud estimates that Log4Shell affected 8% of all artifacts in the Maven Central repository, which is four times the average vulnerability impact Wetter and Ringland (2021).

Cloud security has become increasingly important for many real world critical applications. In response to security risks, previous work proposes various intrusion detection systems to meet the resource constraints and dynamic workload challenges in cloud environments Yen et al. (2013); Du et al. (2017); Lin et al. (2020). These approaches inspect system telemetry data such as system metrics or system calls to identify abnormal attack behavior. However, those approaches are *reactive* in nature, which cannot prevent those security vulnerabilities from affecting many cloud users. Moreover, previous intrusion detection schemes do not provide information about the underlying software defects for the developer to fix the security vulnerabilities. To mitigate those vulnerabilities, developers have to manually analyze massive code bases to figure out the underlying root causes. In this paper, we make the first step to understand the software vulnerabilities called *security*

bugs in 13 commonly used cloud server systems, which provides foundations for proactively detecting software vulnerabilities before they get released to production cloud systems.

4.1.1 Motivating Example

We illustrate security bugs affecting cloud server systems using the Apache Log4j CVE-2021-44228 vulnerability. The bug occurs because Log4j retrieves data from any external Java Naming Directory Interface (JNDI) server without restriction. Accordingly, attackers can submit a request to lookup a class from the attacker's JNDI server. The exploit example in Figure 4.1 is an HTTP request with a path that contains a JNDI request enclosed in the '${' and '}' substitution characters. The JNDI request contains the vulnerable *LDAP* protocol, the attack server *attackhostname.com*, and the attack class *AttackClass*. Log4j resolves the request with the lookup function on line 171. The vulnerable version of the function only contains line 172, which starts a series of invocations to retrieve the requested class using the LDAP context. However, *lookup* does not validate *name* before this line. Eight hops along the call path, the *getObjectFactoryFromReference* method of the *javax.naming.spi* library loads the *AttackClass* from the external *attackhostname.com* and creates an instance using the *java.lang.Class newInstance* method. Finally, the application invokes the new *AttackClass* instance, executing its malicious commands.

The developers patch this bug by using allowlists to restrict each component of the JNDI lookup requests, namely the protocol, the hostname, and the class. Developers have to spend a long time analyzing applications in detail to identify vulnerabilities and provide appropriate fixes. Furthermore, the analysis can be challenging because the vulnerable functions often reside at a different location from where the symptoms, such as the results of the executed commands, occur. Understanding the security bug root cause informs the automatic detection and patching tools to efficiently locate the vulnerable function before the production system is affected.

4.1.2 Contribution

In this paper, we investigate 110 recent security bugs selected from over 300 CVEs in the past five years in 13 popular cloud server systems. We categorize the vulnerabilities by answering the following questions: 1) what are the causes of the security bugs? 2) what threat impact does the vulnerable code have? 3) how do developers patch the vulnerable

code?

Specifically, this paper makes the following contributions:

- We identify five common vulnerable code patterns by systematically analyzing 110 security bugs: 1) *improper execution restrictions*, 2) *improper permission checks*, 3) *improper resource path-name checks*, 4) *improper sensitive data handling*, 5) *improper synchronization handling*,
- Our study shows that the leading causes of the security bugs are improper execution restrictions (37%), improper permission checks (25%), and improper resource path-name checks (24%). The remaining bugs are due to improper sensitive data handling (7%) and improper synchronization handling (7%).
- We describe a set of vulnerable code patterns in order to catch vulnerable code before security bugs impact production cloud systems.

The rest of the paper is organized as follows. Section 4.2 describes our methodology for security bug collection and categorization. Section 4.3 presents the details of our security bug categorization. Section 4.4 concludes the paper.

4.2 Methodology

In this section, we present our methodology. We provide details about the examined security bugs, our bug discovery process, and our vulnerable code categorization.

4.2.1 Real-world security bug discovery

We examine 110 real-world security vulnerabilities in 13 popular Java cloud server applications: Apache ActiveMQ, Apache Log4j, Apache Solr, Apache Struts 2, Apache Tomcat, Apache Unomi, Elasticsearch, GlassFish, JBoss, Jenkins, Jetty, Undertow, and WildFly (previously JBoss). The vulnerabilities reside in the core application or in Java libraries used by these programs. To comprehensively study the current state of security vulnerabilities in server systems, we primarily study security-focused bugs over the past 5 years (2017 to 2021) with available open-source code. After inspecting the source code of the bugs, we exclude those that are in tiny vulnerable function and threat impact categories. Accordingly, we examine over 300 security bugs to arrive at 110 studied bugs.

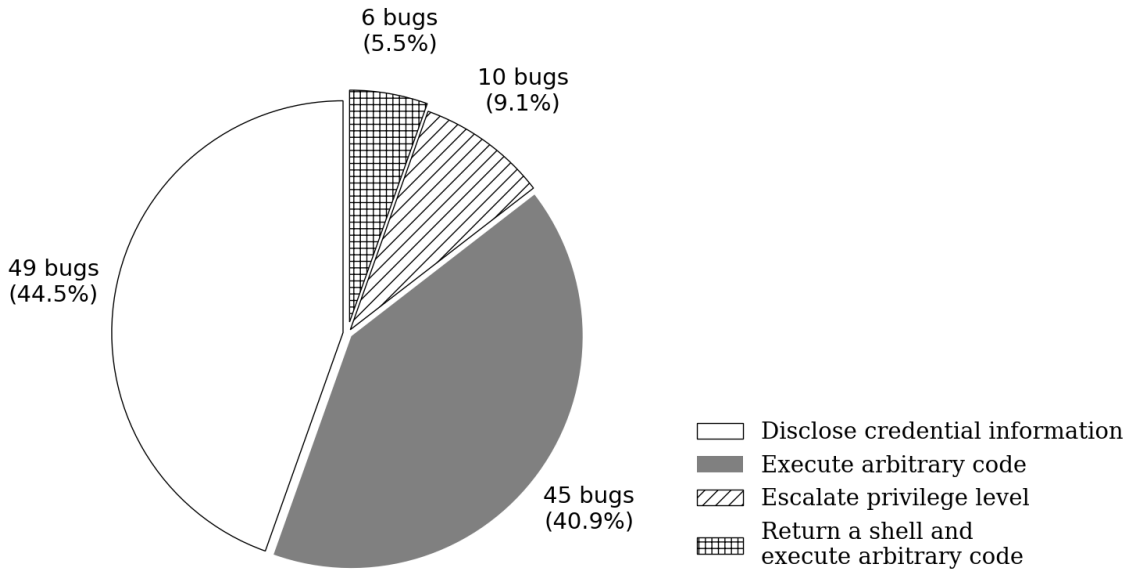

Figure 4.2 Distribution of security bugs by threat impact.

We primarily search for recent CVEs listed for each application in the national vulnerability database (NVD) NIST (2023). The database provides bug descriptions, vulnerable and fixed versions, and other references. We also explore vulnerability databases such as Red Hat Bugzilla Red Hat (2021), Veracode Veracode (2021), and CVEDetails CVE Details (2023) that often include relevant references to vulnerability details. We manually examine the appropriate application versions hosted on repositories like GitHub and Apache Subversion (SVN). We compare differences in application versions, and search through developer correspondence like commits.It is challenging and extremely time-consuming to track and analyze vulnerable code and attack steps as many vulnerability reports do not give detailed exploit descriptions.

Vulnerability databases often record the impact of vulnerability exploits to the system. Figure 4.2 presents the threat impact distribution of the bugs, which is composed of bugs that: 1) disclose credential information, 2) execute arbitrary code, 3) escalate privilege level, and 4) return a shell and execute arbitrary code. We observe that the leading security threats to the cloud server systems are attacks that *disclose credential information* and *execute arbitrary code*, which account for 45% and 41% of the studied bugs, respectively. In contrast, the *escalate privilege level* and return a shell and *execute arbitrary code* represent 9% and 5% of the bugs, respectively.

Table 4.1 List of studied vulnerabilities by root cause.

Root Cause	Description	CVE ID	Threat Impact	Count
Improper execution restrictions	Inadequate or missing restrictions to functions that can execute commands	CVE-2015-8103, CVE-2017-12611, CVE-2017-12629, CVE-2017-5638, CVE-2017-7504, CVE-2017-9791, CVE-2017-9805, CVE-2018-11776, CVE-2019-0192, CVE-2019-0193, CVE-2019-0230, CVE-2019-0232, CVE-2019-1003031, CVE-2019-10104, CVE-2019-10241, CVE-2019-10355, CVE-2019-10431, CVE-2019-14379, CVE-2019-14439, CVE-2019-16538, CVE-2019-17558, CVE-2019-17632, CVE-2020-10740, CVE-2020-11975, CVE-2020-13942, CVE-2020-13957, CVE-2020-17530, CVE-2020-26217, CVE-2020-26258, CVE-2020-26259, CVE-2020-5245, CVE-2020-9484, CVE-2021-21350, CVE-2021-21351, CVE-2021-29505, CVE-2021-39139	Execute arbitrary code	36
		CVE-2014-3120, CVE-2015-1427, CVE-2017-12149, CVE-2019-0221	Return a shell and execute arbitrary code	4
Improper permissions checks	Insufficient or missing checks for security-sensitive parameters used in privileged functions	CVE-2017-7674, CVE-2017-9803, CVE-2018-1000169, CVE-2018-11775, CVE-2018-1305, CVE-2018-3831, CVE-2018-8014, CVE-2018-8034, CVE-2019-12418, CVE-2019-3894, CVE-2019-7611, CVE-2020-1745, CVE-2020-1938, CVE-2020-27216, CVE-2020-7009, CVE-2021-20250, CVE-2021-21605, CVE-2021-29262	Disclose credential information	18
		CVE-2018-14627, CVE-2020-13920, CVE-2020-13941, CVE-2020-1719, CVE-2020-7020, CVE-2021-26117, CVE-2021-29943	Escalate privilege level	7
		CVE-2017-10391, CVE-2018-1000192	Execute arbitrary code	2
Improper resource pathname checks	Incomplete or missing checks to filter requested resource paths and filenames	CVE-2015-5531, CVE-2016-4800, CVE-2017-12196, CVE-2017-12616, CVE-2017-7658, CVE-2017-7675, CVE-2018-1000817, CVE-2018-1047, CVE-2018-10862, CVE-2018-11784, CVE-2018-1304, CVE-2020-1757, CVE-2020-2160, CVE-2021-24122, CVE-2021-28163, CVE-2021-28164, CVE-2021-28169, CVE-2021-34429	Disclose credential information	18
		CVE-2016-3088, CVE-2017-12615, CVE-2017-12617, CVE-2017-2666, CVE-2017-3163, CVE-2019-10184	Execute arbitrary code	6
		CVE-2015-4165, CVE-2017-1000029	Return a shell and execute arbitrary code	2
Improper sensitive data handling	Improper protection of sensitive data that become exposed in program output	CVE-2017-1000030, CVE-2018-1000176, CVE-2018-12536, CVE-2019-10212, CVE-2019-10246, CVE-2019-10247, CVE-2019-3888, CVE-2020-25640	Disclose credential information	8
Improper synchronization handling	Issues in code that handles many concurrent requests	CVE-2018-14642, CVE-2018-17244, CVE-2018-8037, CVE-2020-1732, CVE-2021-25122	Disclose credential information	5
		CVE-2018-12538, CVE-2019-17638, CVE-2019-3894, CVE-2019-7614	Escalate privilege level	3

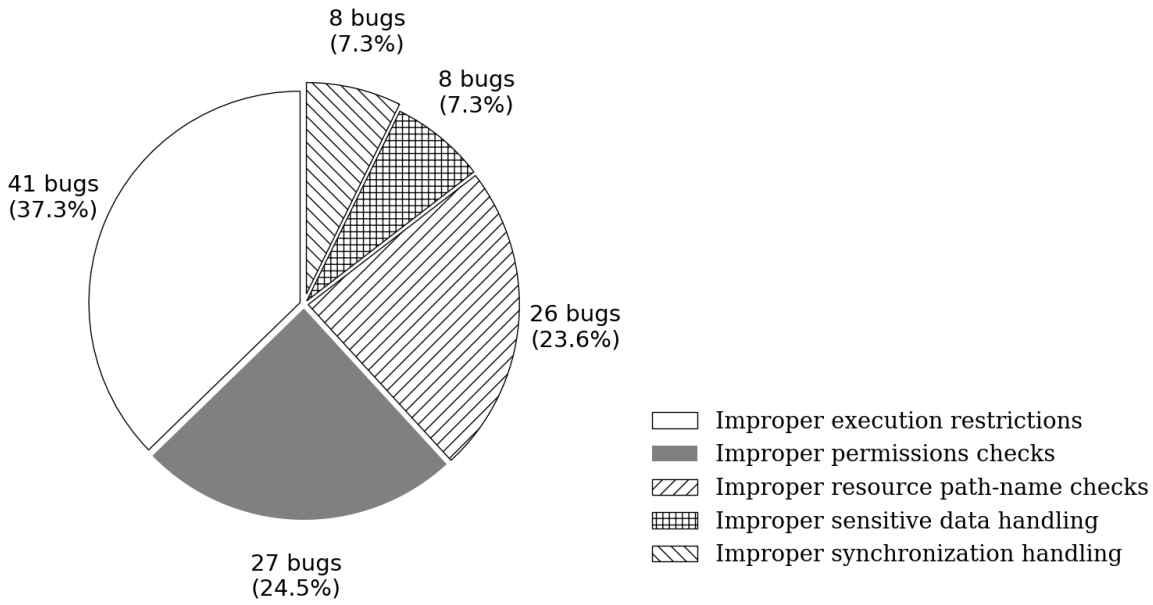

Figure 4.3 Distribution of security bugs by vulnerable code category.

4.2.2 Vulnerable Code Categories

Prior security system work often detects attacks to server software through system events like system call activity, instead of finding the underlying vulnerability patterns in the code. Such an approach learns a model of normal system behavior to identify significant deviations as attack activity. However, in previous work, we see that security attacks do not always differ from the expected system behavior, which leads to missed detection. In addition, unexpected activity can occur under normal operation, which results in false alarms Tunde-Onadele et al. (2020); Lin et al. (2020). Security personnel need not only the detection alarms that the prior systems provide but also an explanation of the attack cause to confidently defend against attacks. Thus, we study vulnerable code to understand how to locate the cause of software defects.

We start by investigating the vulnerabilities in similar threat impact groups because we notice that the security attacks use similar commands to exploit them. For instance, many attacks call the *exec* method of the Java *Runtime* class to execute arbitrary commands. Nevertheless, attackers can modify their requests to use *ProcessBuilder* objects instead so we pay attention to the code features of the vulnerable application. Specifically, we identify the vulnerable function and variables that explain why the bug occurs at the vulnerable code location. We examine how the attack exploitation commands move through the application code until they succeed at a vulnerable code location, using debugging tools when possible. Finally, we repeat our analysis for all the vulnerabilities and group similar causes to extract common patterns.

Figure 4.3 outlines the distribution of the studied security bug categories, defined below[1].

1. **Improper execution restrictions** characterize inadequate or missing restrictions to functions that can execute malicious commands.

2. **Improper permission checks** characterize insufficient or missing checks for security-sensitive parameters used in privileged functions.

3. **Improper resource path-name checks** characterize incomplete or missing checks to filter requested resource paths and filenames.

[1]We publish a dataset repository of our studied bugs resources at www.github.com/NCSU-DANCE-Research-Group/understanding-sec-vuln.

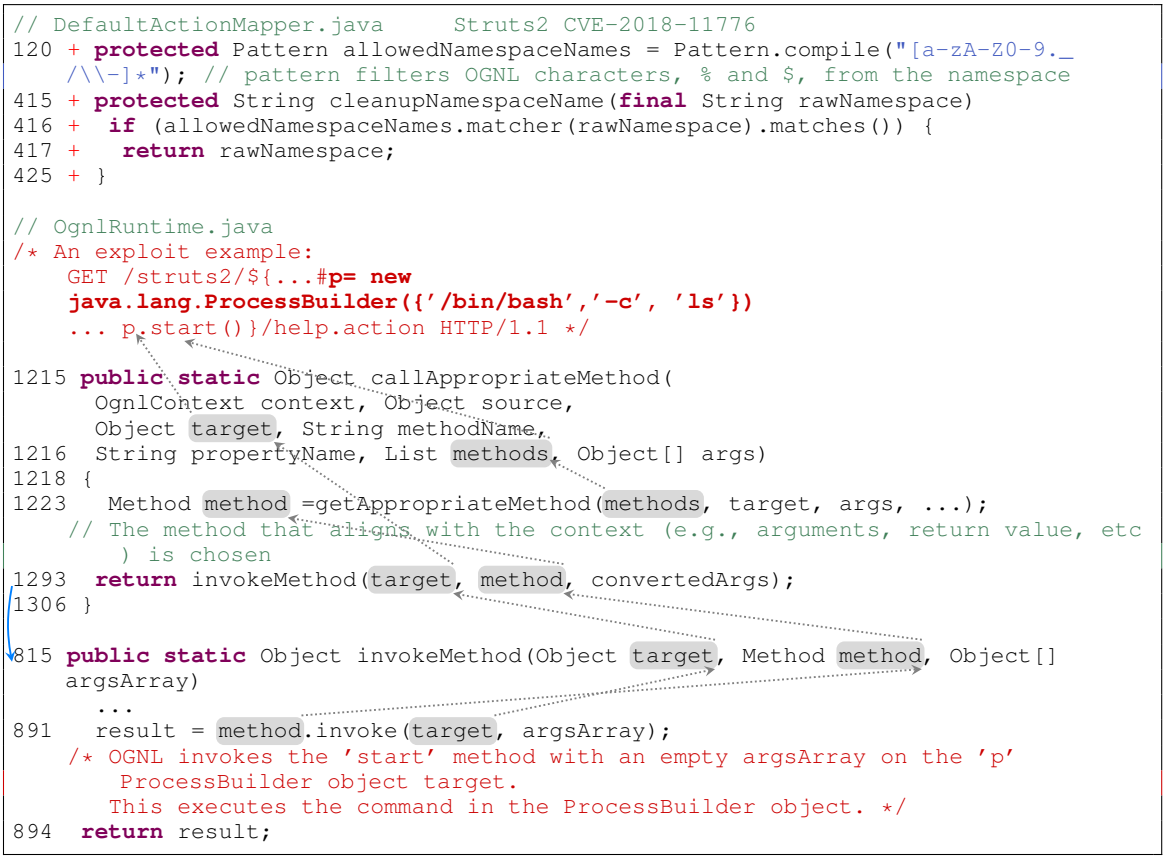

Figure 4.4 The Apache Struts 2 CVE-2018-11776 bug (CVSSv3: 8.1, CVSSv2: 9.3). The vulnerable function *invokeMethod* calls the security-sensitive Java *invoke* method to execute commands inserted into the namespace section of a URI without restriction. This 'improper execution restrictions' bug has the 'execute arbitrary code' impact.

4. **Improper sensitive data handling** characterize improper protection of sensitive data that become exposed in program output.

5. **Improper synchronization handling** characterize issues in code that handles many concurrent requests.

We find that three prominent categories, *improper execution restrictions*, *improper permission checks*, and *improper resource path-name checks* span 37%, 25%, and 24% of the bugs, respectively. The remaining categories, *improper sensitive data handling* and *improper synchronization handling*, each includes 7% of the bugs.

4.3 Security Bug Characteristics

In this section, we explain the characteristics of our studied security bugs with representative examples. For each category, we describe its vulnerable function patterns, patching strategies, and analysis summary.

4.3.1 Improper execution restrictions

Vulnerable function patterns: Web server applications provide features that attackers target for code execution. Many servers offer scripting capabilities to help users automate tasks. Moreover, applications accept structured input such as extensible markup language (XML) files via a stream of bytes and then *deserialize* the bytes into objects and data structures. However, malicious users can manipulate the application to run unsafe classes during deserialization. Thus, developers need to restrict these powerful features so that malicious users do not compromise the server. The vulnerabilities in this category are due to inadequate or missing restrictions to functions that can execute commands. For instance, code that maintains a blocklist of unsafe classes may be incomplete. Therefore, attackers can use unanticipated classes to call functions like *java.lang.Runtime#exec* or *java.lang.ProcessBuilder#start* that create processes to execute commands.

We illustrate this category with the Apache Struts 2 vulnerability, CVE-2018-11776. The vulnerability allows execution of expressions included in a user-requested uniform resource identifier (URI). If the namespace section of the URI contains an object-graph navigation language (OGNL) expression marked by '%{}' or '${},' Struts 2 prepares it for evaluation. The vulnerability resides in the *OgnlRuntime* class of the OGNL package. Figure 4.4 shows an example exploit request above the *OgnlRuntime* class (line 1215). The GET request includes an OGNL expression within '${' and '}'. First, the expression defines a *p* variable, which refers to a Java ProcessBuilder object constructed to execute the bash command, *ls*. Next, the expression calls the *start* method against the *p* object to create the bash process. To invoke the *start* method in the expression, execution eventually reaches the vulnerable function, *invokeMethod* on line 815. Specifically, on line 891, the Java *invoke* function is called to invoke the *start* method against the *p* ProcessBuilder object given by the method and target variables, respectively. We highlight how these vulnerable variables connect to the exploit request with data dependency flow arrows. Accordingly, a new process starts to run the *ls* command. However, the application can execute any malicious code injected in

the crafted URI.

Patching strategies: We notice four main patching approaches. First, developers implement input checks to avoid improper use of an execution function. For example, the manual patch of CVE-2018-11776 is applied in the *DefaultActionMapper* class, where the namespace section of the URI is parsed. Specifically, ActiveMQ uses the regex pattern on line 120 in a *cleanupNamespaceName* method to prevent the use of OGNL characters by excluding them from the set of allowed characters. As another example, JBoss (CVE-2020-5245) includes checks to exclude commands within scripting characters, '${}.' Second, developers eliminate unsafe classes using a blocklist, or define allowed ones in an allowlist. For instance, the jackson-databind library of Apache Struts 2 (CVE-2019-14379) prevents the invocation of an *ehcache* class that is used to load other unsafe classes. Third, developers introduce security variables to control the callers of execution functions. For example, the patch for Elasticsearch (CVE-2014-3120) only allows registered plugins to call its execution function. Finally, developers may disable a feature that allows command execution by default or altogether if it is not a core application function. For example, the Apache Solr CVE-2017-12629 patch stops parsing external entities of XML files.

Observations: Improper execution restrictions frequently occur because no restrictions are present. In 22 out of 41 bugs (54%), the applications do not have checks to filter unsafe inputs or prevent unsafe classes from evaluating inputs. We also find that 16 out of 41 bugs (39%) are due to improper restrictions during deserialization. Developers often address deserialization with blocklists. However, applications that filter classes with blocklists (22% of cases) tend to be vulnerable because the lists need to be comprehensive. Attackers only need to find a new unsafe class to defeat this measure. For instance, applications that use XStream for XML processing encounter at least five recent CVEs related to deserialization. Thus, the fix for the most recent of them, CVE-2021-39139, modifies the application to use an allowlist by default.

4.3.2 Improper permission checks

Vulnerable function patterns: Web server applications use permissions to control different levels of access to its resources. Applications use permissions for privileged server functions such as actions related to connections, file access, and security policies. In addition, the libraries they leverage offer parameters to control specific security-relevant functions. Furthermore, the application considers internet protocol properties such as hypertext transfer

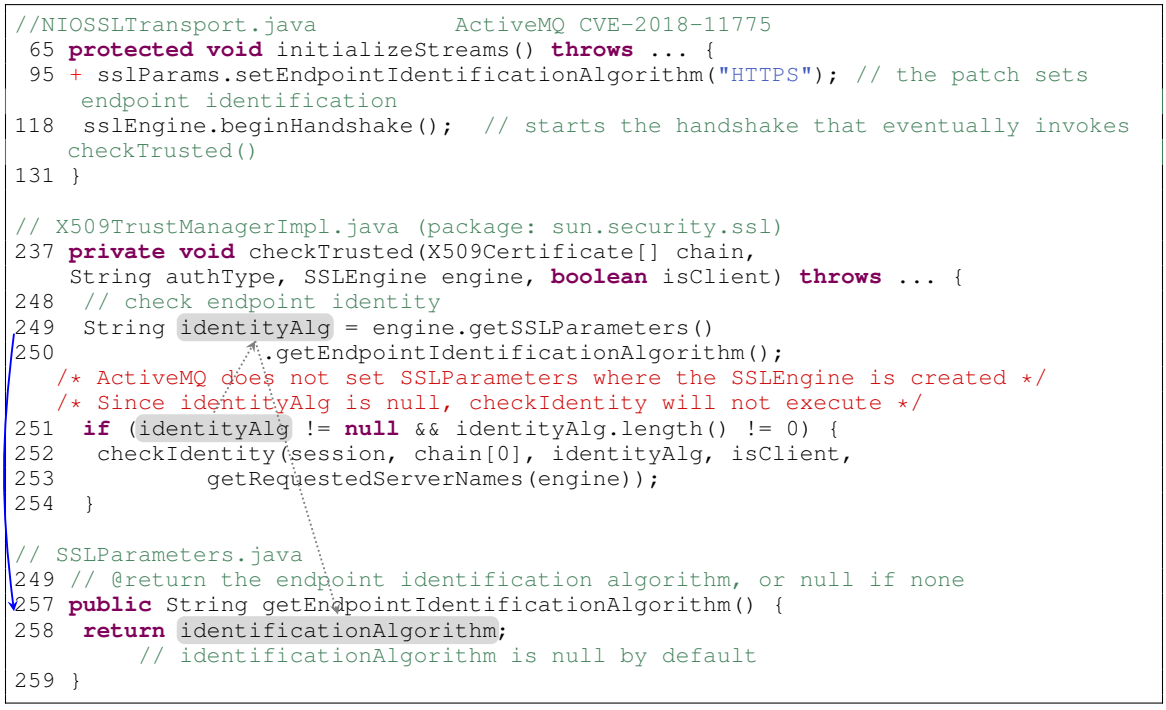

Figure 4.5 The ActiveMQ CVE-2018-11775 bug (CVSSv3: 7.4, CVSSv2: 5.8). The function *initializeStreams* never sets an identification algorithm so the variable *identityAlg* does not meet the condition to call the security-sensitive *checkIdentity* function to better secure TLS connections. This 'improper permission checks' bug has the 'disclose credential information' impact.

protocol (HTTP) header attributes that impact security. Accordingly, managing multifaceted permissions becomes complex and error-prone. The vulnerabilities in this category do not have sufficient checks for security-sensitive parameters. Such parameters may be variables related to configuration, security context, or keys used in sensitive functions.

For example, Apache ActiveMQ versions before 5.15.6 establish socket connections without the use of transport layer security (TLS)/secure socket layer (SSL) parameters to verify server hostname identity. This CVE-2018-11775 vulnerability exposes the application to man-in-the-middle (MITM) attacks. During a TLS handshake, a server confirms its authenticity with a certificate, which the client verifies against that from an official certificate authority (CA). If endpoint identification is not configured for the communication, the client does not confirm that the entity presenting the certificate is the intended hostname. Consider the code excerpt in Figure 4.5. As the server certificate is processed, execution reaches the Java *checkTrusted* function to confirm server identity. On line 249, *checkTrusted* gets the endpoint identification algorithm set by SSL parameters and saves the result to

identityAlg. However, since the identification algorithm is not set by ActiveMQ, *identityAlg* is null because *getEndpointIdentificationAlgorithm* returns null by default (line 258). Thus, the *checkIdentity* function, which performs the hostname check, is not called on line 252. Consequently, an attacker can intercept connections between the client and the expected server.

Patching strategies: Improper permission checks happen when applications 1) miss security-sensitive parameters in libraries, 2) miss checks for privileged application functions, or 3) have logical errors in the implementation of permissions. Accordingly, the patches generally take three approaches. First, developers set missing configuration parameters often provided by libraries. For example, the patch of CVE-2018-11775 introduces Java *SSLParameters* on lines 93-97 to configure the HTTPS endpoint identification when initializing TLS connections. Apache Tomcat (CVE-2018-8034) also adds the TLS parameters needed to verify the identity of a client hostname against the certificate it presents. Next, the patch introduces checks for permissions of core application classes. The change typically modifies supporting classes and function arguments to include the permission variable. Apache ActiveMQ patches CVE-2020-13920 with a class that checks user access permissions before modifying its remote method invocation (RMI) server. Finally, the patch may adjust inaccurate logic of existing permission checks. For instance, Apache Tomcat (CVE-2018-1305) moves permission instructions outside a check for a specific authentication level so that the instruction applies for all levels.

Observations: We observe that 8 out of the 27 bugs (30%) occur when security configurations are missing. Developers need specific knowledge of numerous properties to address these vulnerabilities, which is challenging. We also find that 33% of the bugs are due to logical errors in the implementation of permissions. Vulnerabilities occur when developers apply security instructions at the wrong time or create conflicting permission variables. For example, Apache Tomcat (CVE-2018-1305) wrongly applies security parameters at the start of the application before the target servlet loads. Finally, we note that developers sometimes neglect to consider error messages as privileged actions since the message can allow a user to infer the existence of a resource.

4.3.3 Improper resource path-name checks

Vulnerable function patterns: The vulnerabilities in this category are due to incomplete or missing checks for proper resource path-names. For instance, the code may miss a check

```
// FileDirContext.java        Tomcat CVE-2017-12615
/* An exploit example:
   PUT /aaa.jsp/ HTTP/1.1
      ...Process p = Runtime.getRuntime().exec
      (request.getParameter(cmd)).... */
780 - protected File file(String name) {
    + protected File file(String name, boolean mustExist)
782    File file = new File(base, name);
            // indirectly invokes the normalize function
    +   if (name.endsWith("/") && file.isFile()) {
    +     return null;
    +   }
826 }

// WinNTFileSystem.java
102 private String normalize(String path, int len, int off){
107  StringBuffer sb = new StringBuffer(len);
115   sb.append(path.substring(0, off)); /* off is one less than the actual path
    length so the path is returned without the last character
     (e.g., 'aaa.jsp/' becomes 'aaa.jsp') */
     ...
159  String rv = sb.toString();
160  return rv; /* rv == 'aaa.jsp' is the filename to be created (with malicious
    content) */
161 }
```

Figure 4.6 The Apache Tomcat CVE-2017-12615 bug (CVSSv3: 8.1, CVSSv2: 6.8). The vulnerable function *file* does not verify safe file extensions after calling the security-sensitive *normalize* function that can modify the file extension. This 'improper resource path-name checks' bug has the 'execute arbitrary code' impact.

for a specific slash characters to ensure that a user-provided path does not go outside the web directory.

We highlight this bug category with the Apache Tomcat CVE-2017-12615 vulnerability. The exposure allows a user to upload and execute jakarta server pages (jsp) by bypassing jsp restrictions. Tomcat typically processes and restricts files ending with ".jsp" using its JSPServlet class. However, when an attacker appends an extra "/" character to the file extension, the application does not recognize it as jsp and processes it with the DefaultServlet class instead. Figure 4.6 shows an example exploit request. The PUT request specifies the crafted file name, 'aaa.jsp/', to be created, followed by some malicious content. The content includes the *java.lang.Runtime#exec* function so that the jsp file can execute commands. Before creating files, Tomcat filters out trailing slashes not allowed in Windows filenames using the Java *normalize* function from the WinNTFileSystem class. In our example, at line 102, normalize will be called with the *path* variable as 'aaa.jsp/' and *off* as $len-1$, the index of '/'. On line 159, the *path* is truncated by removing the trailing slash. Thus, 'aaa.jsp' is saved to *sb*, which is passed to *rv* and returned. Thus, the attacker is able to successfully create a

file with the intended jsp filename extension. After 'aaa.jsp' is created, it is converted to a java file 'aaa_jsp.java' to service additional command requests. The attacker can now send an additional GET request with a command as an HTTP parameter. The request reaches the *_jspService* function of the 'aaa_jsp.java' file, which calls the inserted *exec* method to execute the requested command. Thus, the Tomcat server can be used to execute arbitrary code input by the attacker.

Patching strategies: Developers patch improper path-name checks with four main methods. First, developers add checks to filter characters such as directory traversal characters like '../' that attackers use to access unintended parent directories or other characters that are disallowed from filenames. These patches add new character cases to check for or use regular expressions to allow or disallow certain characters like the Apache Struts 2 (CVE-2018-11776) fix. Second, developers check for special characters that follow filenames and extensions. Attackers insert special characters after unsafe paths, knowing that the checks would resolve the path-name silently, as in Apache Tomcat (CVE-2017-12615) shown in Figure 4.6. The developer patch adds filename checks primarily in the *file* function of *FileDirContext* where files are created. After line 782, if the path-name has a trailing slash, null is returned so that the file is not created. Third, developers use consistent path-names to identify resources. Applications such as Jetty (CVE-2021-28163) have checks for accepting resources into sensitive directories that expect absolute paths instead of other aliases. These patches often extract absolute paths before applying checks to avoid errors. Finally, developers fix logical errors that prevent path-names from reaching expected checks. For example, Apache Tomcat (CVE-2017-7675) corrects the object type of a path variable to satisfy the branch conditions for performing path-name checks as intended.

Developers usually implement the above checks against malicious path-names with a specialized function named as *normalize*. These normalization functions include checks that filter directory traversal characters, remove other special disallowed characters, or extract absolute paths. Normalization functions resemble the *normalize* and *resolve* functions of *java.nio.file.Path* but include more comprehensive or application-specific checks.

Observations: Developers need to protect applications against unsafe input paths and filenames. In 12 out of 26 (46%) cases, necessary checks are missing in the appropriate classes. In complex application codebases, it is challenging to know where to place checks. We observe that 17 (65%) out of the 26 vulnerabilities are in existing normalization functions.

```
//                                WildFly CVE-2020-25640
// JmsConnectionFailedException.java
  40 private static String extractMessage(IOException cause) {
  41  String m = cause.getMessage();
  42  if (m == null || m.length() == 0) {
  43   m = cause.toString();
  44  }
  45  return m;

// JmsManagedConnection.java
1009 public String toString() {
1010  return "JmsManagedConnection{"
1011  + "mcf=" + mcf
1012  + ", info=" + info
1013  + ", user=" + user
1014 - + ", pwd=" + pwd
         /* pwd is the password printed in plain text */
1014 + + ", pwd=****"
         // the patch excludes the password
1025  + '}';
1026 }
```

Figure 4.7 The WildFly CVE-2020-25640 bug (CVSSv3: 5.3, CVSSv2: 3.5). The vulnerable function *toString* outputs the security-sensitive *pwd* password variable. This 'improper sensitive data handling' bug has the 'disclose credential information' impact.

4.3.4 Improper sensitive data handling

Vulnerable function patterns: These vulnerabilities occur when applications do not properly handle sensitive data such as credentials or full document paths, so that the data is exposed to users in some program output. Applications can reveal plaintext passwords and full file base names in error messages and web pages (CVE-2020-25640 and CVE-2019-10247). Otherwise, the applications may display the information upon requests for certain expected files. For instance, passwords can be found in a JVM report (CVE-2017-1000030), keys in a configuration file (CVE-2018-1000176) or full filenames on a directory listing web page (CVE-2019-10246).

In Figure 4.7, Wildly CVE-2020-25640 prints all field variables including password in plain-text in exception messages. Thus, an attacker can induce an error to access exposed credentials. WildFly has exception classes to extract and output specific exception messages. For Java Message Service (JMS) connections, the *extractMessage* function of the *JmsConnectionFailedException* class is invoked. If a message is not found for the exception via *getMessage*, the *toString* method is called on line 43. This invokes the vulnerable *toString* function of *JmsManagedConnection*, which prints the password *pwd* on line 1014.

Patching strategies: The patches often remove sensitive objects or parameters from

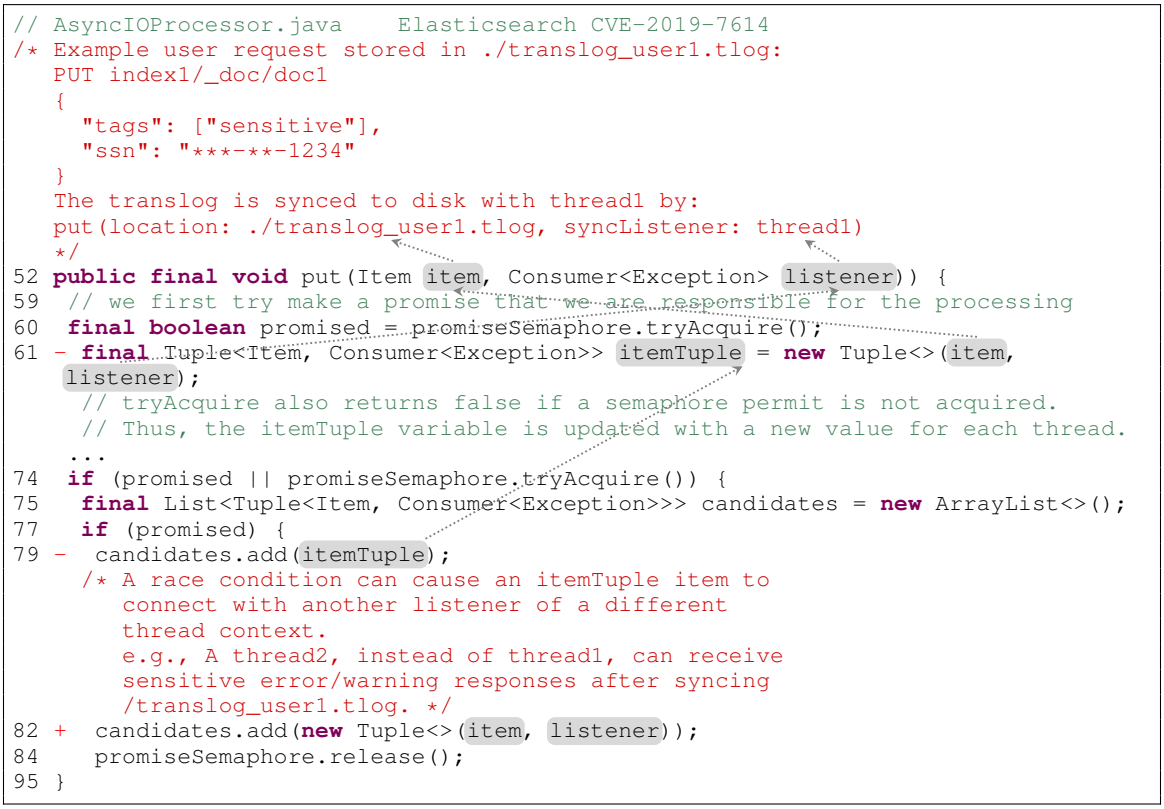

Figure 4.8 The Elasticsearch CVE-2019-7614 bug (CVSSv3: 5.9, CVSSv2: 4.3). The vulnerable function *put* modifies the *itemTuple* variable outside its critical section, which leads to a race condition that can mix user data. This 'improper synchronization handling' bug has the 'disclose credential information' impact.

print and log statements. For instance, in Figure 4.7, the patch uses '***' on line 1014 to protect the password. Jetty (CVE-2019-10247) removes header objects that contain credentials from the *toString* function of its *HttpServerExchange* class. In addition, the patches trim variables that expose full paths. Finally, developers introduce permissions for getters used to retrieve sensitive objects. Jenkins (CVE-2018-1000176) encapsulates relevant functions in a new class that checks user accounts.

Observations: We observe that improper sensitive data handling occurs when developers do not filter sensitive object variables and full file paths from being displayed in messages and publicly accessible files. Four of eight bugs (50%) are exposed by the *toString* function of sensitive objects. The remaining bugs call log statements (25%) or getters of sensitive objects (25%) (such as *getSmtpAuthPassword*, or *getFileName*) before the objects are output.

4.3.5 Improper synchronization handling

Vulnerable function patterns: These security vulnerabilities are caused by issues in code that handles many concurrent requests. Improper synchronization can allow threads to access the content of variables from other thread contexts.

The Elasticsearch CVE-2019-7614 exposure, shown in Figure 4.8, can allow sensitive responses to be delivered to the wrong user. An example PUT request is made to include sensitive data in document *doc1*. Elasticsearch temporarily stores request data and statistics in its transaction log (translog) before writing to the underlying disk. To sync the translog, Elasticsearch invokes the *put* function of the AsyncIOProcessor class with the translog location as the *item* and a listener thread as the *listener*. The vulnerable function is the *put* function due to the *itemTuple* variable. The AsyncIOProcessor class uses semaphores to process multiple IO operations in batches. On line 60, the semaphore *tryAcquire* function returns false if a semaphore permit is not acquired. The next statement then initializes the *itemTuple* variable. However, on line 79, *itemTuple* is also used within the critical section from line 77 to 84. Thus, a race condition can result when *itemTuple* is updated by a thread on line 61 as it is added to the *candidates* list by another thread on line 79. When the write operation is complete, the listener is notified of errors or warnings such as deprecation warnings, which can contain sensitive details from the translog. Consequently, such messages can be returned to the wrong listener for the request.

Patching strategies: In four (50%) out of eight cases, the patches provide threads with variables and classes to preserve their context. Elasticsearch (CVE-2018-17244) adds a metadata field to its AuthenticationResult class to hold security token data. Context may be added in combination with other changes. Jetty (CVE-2018-12538) employs the thread-safe *ConcurrentHashMap* to track user sessions. In Figure 4.8, the patch of Elasticsearch (CVE-2019-7614) removes line 61 from the function and replaces line 79 with line 82. It not only updates shared variables inside the critical section to avoid race conditions but also adds context to threads before placing them in a waitlist. In two cases, developers fix how buffers that contain user data are managed. The Jetty CVE-2019-17638 patch adds checks to appropriately clear the buffer before an exception occurs to prevent double release, while the Apache Tomcat CVE-2021-25122 patch clears buffer for HTTP header content before handling an upgrade request. Otherwise, the patch fixes improper data types and structures used for synchronization.

Observations: Improper synchronization handling results in mix-up of user data. To

overcome the problems, developers can implement thread-safe variables and data structures, check that structures only contain single user information, and ensure that variables are not updated outside their critical sections. In addition, developers may provide thread classes with context about request data to resolve the vulnerabilities.

4.4 Summary

In this paper, we have presented a comprehensive study over 110 recent real world security bugs in 13 popular cloud server systems. Our study first identifies five common vulnerability categories among those 110 studied security bugs: 1) *improper execution restrictions*, 2) *improper permission checks*, 3) *improper resource path-name checks*, 4) *improper sensitive data handling*, and 5) *improper synchronization handling*. Furthermore, we extract key software code patterns in each category. We believe that our work makes the first step toward proactively protecting cloud server systems from security bugs.

CHAPTER

5

XSCOPE: DETECTING CODE EXECUTION VULNERABILITIES IN CLOUD SERVER SYSTEMS

5.1 Introduction

Cloud systems provide on-demand elastic computing resources and services over the Internet, without requiring users to manage or maintain their own physical devices or infrastructure. Cloud systems allow developers to quickly deploy and update their code, without having to wait for hardware provisioning, installation, or configuration. Hence, cloud computing enables developers to deliver new features and fixes to their users much faster and more frequently than ever before SalesForce (2023) Cloud (2023). Code can be deployed and executed on a cloud system, and interact with the cloud system through various methods, such as APIs, event streaming, message brokers, or containers Baudoin et al. (2014) Nayyar and Kasthuri (2021). Today, cloud systems often leverage open source

code to benefit from many useful features from logging frameworks, continuous integration, continuous deployment (CI/CD), web servers, and many other open source software projects.

However, software bugs that execute arbitrary malicious code are notoriously bad for cloud systems. The U.S. Cybersecurity and Infrastructure Security Agency (CISA) released its latest list of the top 12 routinely exploited vulnerabilities in 2022 Cybersecurity and Agency (2023). Over half of them were code execution attacks, including the notorious Log4j bug. A recent study finds the predominant code vulnerability category, representing 37% of the examined vulnerabilities, are improper execution restrictions Tunde-Onadele et al. (2022). Java security tools dedicate a significant of their detection solutions to address code execution bugs. For instance, 45% of the FindSecBugs OWASP (2023) bug patterns target code execution issues. The vast MITRE ATT&CK knowledge base of observed attack tactics also highlights an execution category and execution techniques in other categories Corporation (2023).

Existing work has been focusing on detecting security attacks using intrusion detection systems (IDS) Shen et al. (2018); Lin et al. (2022, 2020); Tunde-Onadele et al. (2020); Yen et al. (2013); Dash et al. (2016). IDS approaches are reactive in nature, which can only detect a security attack after the attack happens. Moreover, they often cannot reveal how an attack has been triggered or pinpoint the vulnerable code that is the culprit of the security attack. Other work has been done to detect code vulnerabilities with static program analysis Thomé et al. (2017b); Livshits and Lam (2005); Enck et al. (2014); Zheng and Zhang (2013) or symbolic execution Thomé et al. (2017a); Fratantonio et al. (2016). However, existing vulnerability detection solutions often suffer from either high false positives or high false negatives due to too general or too narrow rule-based approaches.

5.1.1 Motivating Example

Execution attacks have made a critical impact on server systems on a global scale in recent years. On December 9, 2021, the Apache Software Foundation publicly disclosed the Log4j vulnerability (CVE-2021-44228), Log4Shell, allowing remote code execution in dependent Java cloud software. Five days after the disclosure, the Financial Times reported over 1.2 million Log4Shell attacks with researchers observing up to about 100 attacks per minute Murphy (2021). The bug also affected 8% of all the Java Maven repository artifacts, compared to the average vulnerability impact of 2%, according to the open source insights team from

```
1  /*
2   * An attack example:
3   * > curl 'http://vulnerablename.com/...?action=
4   *         ${jndi:ldap://attackerhostname.com/
5   *           MaliciousClass}' ...
6   */
7   // Class: JndiManager
8   public synchronized <T> T lookup(final String name)... {
9
10      return (T) this.context.lookup(name);
11      /* no security checks on the "name"
12      argument containing user input */
13  }
```

Figure 5.1 A real-world execution vulnerability in Apache Log4j (CVE-2021-44228). The function does not perform security checks on the `name` variable during a JNDI lookup. This allows malicious user inputs in `name` to eventually execute.

Google Cloud Wetter and Ringland (2021). Furthermore, developers found it challenging to locate software that depends on the Log4j package, especially over many dependency levels. The vulnerability, undetected in the Log4j package for eight years, demonstrates the need for vulnerability detection in cloud software.

Figure 5.1 shows the vulnerability details. The comments from line 2 to line 5 give an example attack vector that a malicious actor can send to a vulnerable system (e.g. *vulnerablename.com*), using a simple curl HTTP request. This malicious request tells the vulnerable application to make a Java Naming and Directory Interface (JNDI) lookup for the attacker's server at *attackerhostname.com* to retrieve a *MaliciousClass*. The embedded command would be resolved by calling the vulnerable function cause, `JndiManager.lookup()`, on lines 8 to 13. In particular, the part of the command highlighted in bold enters the *name* argument of the function. The inner `lookup` function in line 10 indirectly calls a key system-level execution function in the Java libraries to execute the value in *name*. So the security vulnerability is caused by missing security checks on the *name* argument before calling the system-level execution function `JndiManager.lookup()` that accepts and deserializes remote objects.

5.1.2 Contribution

In this chapter, we present XScope, an automatic pattern-driven execution vulnerability checker for proactively identifying vulnerable program functions for cloud server systems. In contrast to previous work, XScope does not rely on pre-defined rules to identify code vulnerabilities. Instead, XScope proposes a drill-down protocol to narrow down the vulnerable code segments. XScope also leverages insights about the vulnerable functions and the security patches from recent code execution vulnerabilities to optimize code vulnerability detection accuracy.

The key observation behind XScope is that the security attack exploiting an improper code execution restriction vulnerability often takes advantage of missing important security checks before invoking certain system-level execution library calls such as `java.lang.Runtime.exec()`. Thus, XScope first starts from those key system-level execution library functions to identify potential risky functions that invoke those execution library functions using call graph analysis. However, not all of those candidate functions are vulnerable. Next, XScope leverages data-flow analysis to filter out those safe functions which include proper security checks when the functions perform any execution decision based on external program inputs or derived data from external inputs. Those security checks are derived from the security patches to past code execution vulnerabilities. For those candidate vulnerable functions, XScope performs ranking among them based on the number of unprotected data-flow paths leading to them. In this chapter, we focus on Java programs which are commonly used in cloud server systems. However, our approach can be applied to any programming language with proper code analysis tools. Specifically, this chapter makes the following contributions.

- We present a new pattern-driven vulnerability detection framework that can proactively detect code execution vulnerabilities before they are released into production environments.
- We introduce a drill-down code analysis protocol that starts by identifying candidate vulnerable functions and then performing a security check based filtering scheme to narrow down truly vulnerable functions.
- We have implemented a prototype of XScope and evaluated it over real-world vulnerabilities in six commonly used cloud server systems.

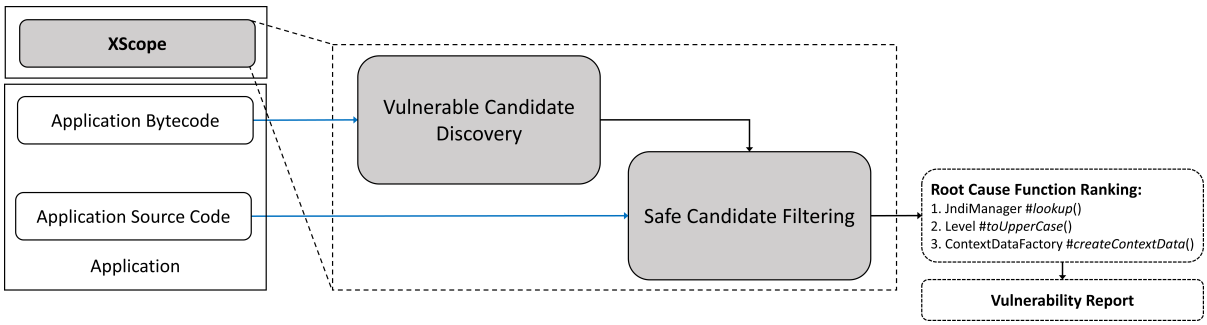

Figure 5.2 System overview of XScope.

Our results show that XScope can localize the vulnerable functions in all the tested execution vulnerabilities with 166.7% higher detection rate, compared to an existing popular security code checking tool, Find Security Bugs (FindSecBugs), provided by the Open Worldwide Application Security Project (OWASP) OWASP (2023). Furthermore, in the cases where both methods detect the bug, XScope can lower the false alarm by 53%.

The rest of the chapter is organized as follows. section 5.2 presents the details of our system design. section 5.4 describes our experimental evaluation over real-world improper execution restriction bugs. section 5.5 concludes the chapter.

5.2 System Design

In this section, we describe the system design of Xscope. We provide an overview, explain our vulnerable code execution patterns, and discuss each component in detail.

5.2.1 Overview

XScope provides a hybrid code analysis framework to detect remote code execution vulnerabilities that have improper execution restrictions. It combines static analysis techniques that examine application code to pinpoint vulnerable function causes using vulnerable code execution patterns. XScope, illustrated in Figure 5.2, consists of two primary components, namely i) vulnerable candidate discovery, and ii) safe candidate filtering.

When XScope is triggered for a codebase, the *vulnerable candidate discovery* component analyzes an intermediate representation (IR) of the application bytecode to identify a set of candidates that can make sensitive system-level execution library function calls

Table 5.1 Call stack snippet during an Apache Log4j (CVE-2021-44228) exploit.

Package	Class.Method
org...log4j	LoggerConfig.log()
...	
org...log4j	StrSubstitutor.resolveVariable()
...	
org...log4j	Interpolator.lookup()
org...log4j	JndiLookup.lookup()
org...log4j	JndiManager.lookup() [**vulnerable function**]
javax	InitialContext.lookup()
com.sun	ldapURLContext.lookup()
...	
javax	DirectoryManager.getObjectInstance()
javax	NamingManager.getObjectFactoryFromReference()
	[**key Java execution library function**]

(subsection 5.2.3). In this work, these key system-level execution library functions are those Java functions that can execute arbitrary code (e.g. `java.lang.Runtime.exec()`). Using the static analysis tool, Soot Vallée-Rai et al. (2010), we trace the application call graph backward from the key system-level execution library functions to find reachable vulnerable function candidates within the application. Within each vulnerable candidate function, we can locate the specific vulnerable call on this sensitive call path, which we refer to as the candidate sink.

Next, the *safe candidate filtering* module evaluates how malicious data can flow from user-controlled application input sources to the previously obtained vulnerable candidate sink (subsection 5.2.4). Using the CodeQL GitHub, Inc. (2021) code analysis tool, we review the security-related operations in checks along the data-flow paths and apply filtering techniques to remove protected functions from the candidate list, narrowing down the actual vulnerable functions. Finally, this module tracks the remaining vulnerable data-flow paths to output a ranked list of vulnerable functions along with a report of the identified vulnerable paths.

```
+ protected String cleanupNamespaceName(final String
   rawNamespace) {
+     if(allowedNamespaceNames.matcher(rawNamespace).matches()
+         return rawNamespace;
+     } else {
+     LOG.warn(
+         "{} did not match allowed..."
+     );
+     return defaultNamespaceName;
+ }
```

Figure 5.3 An example input validation (with regex) patch for Apache Struts CVE-2018-11776

5.2.2 Vulnerable Code Execution Patterns

We introduce two major code patterns that XScope examines for automatically finding execution vulnerabilities. XScope first identifies vulnerable function patterns with call graph techniques and then security check patterns with data-flow mechanisms.

Vulnerable Function Patterns

Table 5.1 shows an execution trace excerpt of Apache Log4j during a CVE-2021-44228 exploit. The program execution advances from top to bottom. As discussed in subsection 5.1.1, an attacker exploits the vulnerability by sending the vulnerable server a malicious request containing an embedded JNDI lookup command to the attacker's server. After receiving the user input, Log4j invokes the `log` function on line 1 to process the log. As the application processes the log, it finds shell characters that represent an unresolved expression so it tries to substitute the expression for its value. Eventually, Log4j processes the JNDI lookup request, calling the `JndiManager.lookup()` method. This lookup method sits at the boundary between the application library and Java library functions. The execution proceeds from lookup through the subsequent Java library functions before reaching the `getObjectFactoryFromReference` method, where the referenced attack class would be retrieved from the attacker's server and then executed.

We refer to functions like getObjectFactoryFromReference that are responsible for command execution as *key system-level execution library functions*. Our pattern-driven checker tracks suspicious application functions based on the following key system-level

```
+ public synchronized <T> T lookup(final String name)... {
+     URI uri = new URI(name);
+     if (!allowedProtocols.contains(uri...)) {
+         LOGGER.warn("Log4j JNDI does not...");
+         return null;
+     }
```

Figure 5.4 An example allowlist patch for Apache Log4j CVE-2021-44228

```
+ if(!"true".equalsIgnoreCase(unsafeSerializableProperty) {
+     throw new UnsupportedOperationException(
+         "...");
+ }
```

Figure 5.5 An example security variable patch for Apache Commons CVE-2017-7504

execution functions found in our study.

1. `java.lang.reflect.Method.invoke()`
2. `java.lang.Runtime.exec()`
3. `javax.naming.spi.NamingManager.getObjectFactoryFromReference()`
4. `java.lang.Class.newInstance()`

However, we omit system-level execution library functions that have now been deprecated from the Java library, namely `invokeFunction` and `invokeMethod` from the `jdk.nashorn.api.scripting.NashornScriptEngine` class.

Security Check Patterns

Unsafe code execution can occur wherever malicious inputs reach the functions that lead to key system-level execution functions without any protective checks. We investigate security check code-patterns to understand the feasibility of execution attacks in potentially vulnerable functions.

We examine the patches of our studied vulnerabilities and observe the following three security-related check types. We show patch examples for each type in Figure 5.3, Figure 5.4, and Figure 5.5.

- *Regular expression (regex) matching*: These checks ensure that input strings conform to expected values and do not contain sensitive characters that induce code execution. Instances of these checks call the `matches` function of the Java Pattern class.

- *Block-list/Allow-list verification*: These checks determine whether a variable value is among an approved values given in an allow-list. They call the `contains` method to verify the variable value (e.g. `List.contains(String)`).

- *Security variable comparison*: These checks use functions like `equalsIgnoreCase` and `equals` to ascertain the values of security properties that lead to execution.

We use the above security operations list as our model of security-related checks for filtering.

5.2.3 Vulnerable Candidate Discovery

We drill down potentially vulnerable functions in the application that can lead to code execution in Java library functions.

To accomplish this, we trace the call graph backward from the key system-level execution functions to extract a set of initial candidate application functions that are reachable. We describe the procedure with the Apache Log4j (CVE-2021-44228) example shown in Figure 5.6. The call graph includes edges from caller function nodes to their callees. We search the graph from the key system-level execution functions to discover vulnerable function candidates in the application. For instance, on the path from `javax...NamingManager.getObjectFactoryFromReference()`, we find candidates like `ReflectionUtil.instantiate()` and the vulnerable function, `JndiManager.lookup()`.

Using the static analysis tool Soot Vallée-Rai et al. (2010), we construct the call graph with the Spark framework Lhoták and Hendren (2003) with the key system-level execution library functions as entry points to explore reachable methods. For each entry point, we traverse the call graph with a conservative approach that includes considering any possible callers of the `newInstance` method of the *Class* class, for example, since we cannot statically determine dynamically loaded object types. In our implementation, we obtain the candidates with a breadth-first search (BFS) from each key system-level execution function until a maximum search depth. We empirically obtain the maximum search depth based on the vulnerabilities in our study. The implementation details are described in section 5.4.

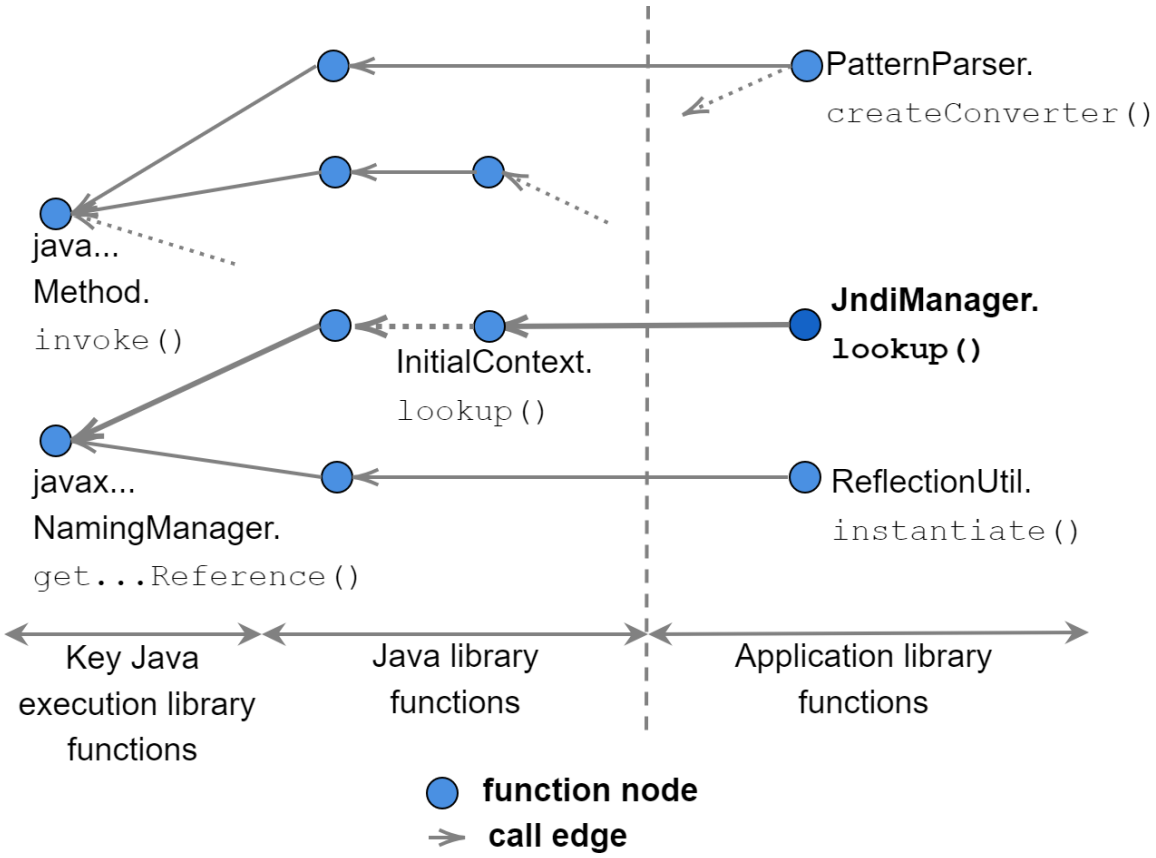

Figure 5.6 An example call graph illustration of Apache Log4j (CVE-2021-44228), representing function calls from caller to callee. We trace the call graph backward from the key system-level execution functions to find vulnerable function candidates in the application libraries.

Finally, our search yields a list of functions, each with their next hop call in the call chain to a key system-level execution library function. We pass these function and next call pairs as candidates to the safe candidate filtering phase.

5.2.4 Safe Candidate Filtering

We filter out safe functions from the vulnerable candidate discovery module, leaving the vulnerable ones that program inputs can flow to without protection.

We implement data-flow analysis to trace user inputs to candidate functions via the open source code analysis engine, CodeQL GitHub, Inc. (2021). CodeQL builds an application source code into a database that one can probe using its QL-based query language.

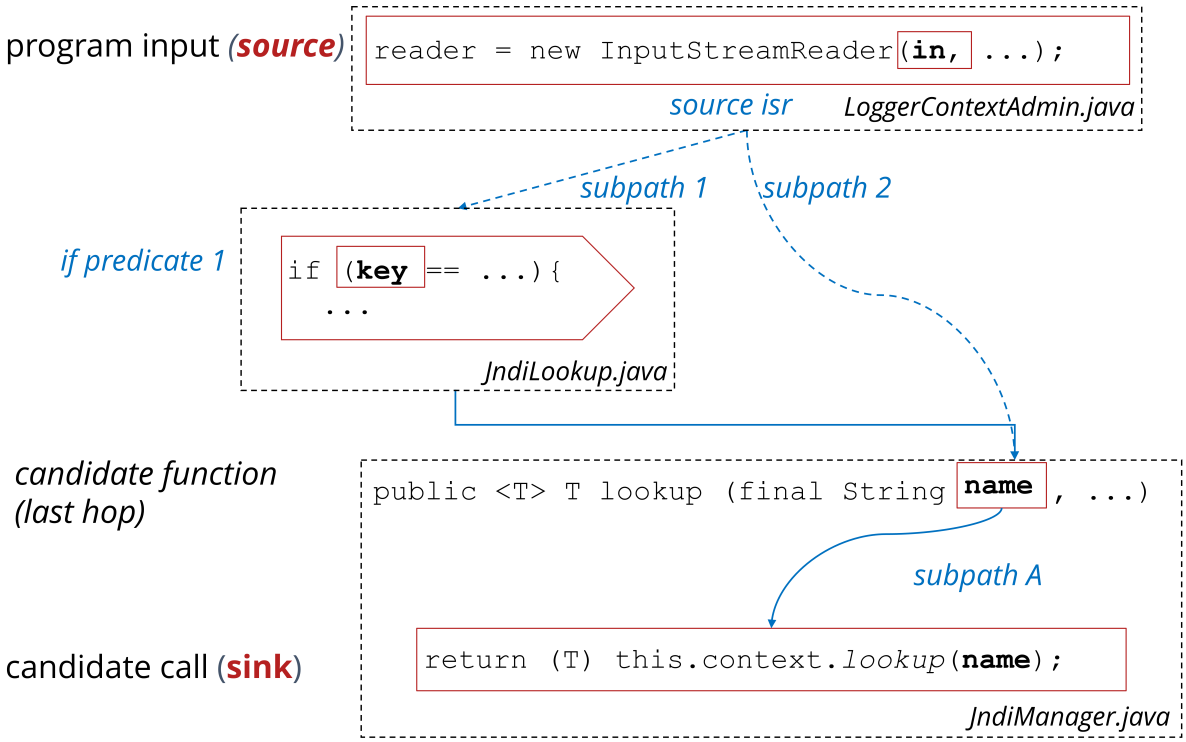

Figure 5.7 Data-flow illustration using Apache Log4j (CVE-2021-44228). A candidate is considered vulnerable if there are unfiltered data-flow paths from a program input source to the candidate such as the path: *source isr* → *subpath 2* → *subpath A*.

In the CodeQL Semmle library, the *UserInput* class refers to both local and remote user inputs. Local inputs include sources that can originate from files (e.g. *FileInputStream* objects), or environment variables, etc. Whereas, remote inputs include sources from relevant Java API like the *Network* class, which has functions to collect user-controlled data (e.g. `getHostName`). We augment the CodeQL-provided flow sources to include *ServletRequestParameterSource* and *AutoCloseable* sources, which track *ServletRequest* parameters from the Java Servlets API (`javax.servlet`), and request classes that extend the AutoCloseable resource class (`java.lang.Autocloseable`), respectively. For instance, Apache Tomcat accepts client inputs with Java *ServletRequest* objects.

We trace data flow from the aforementioned sources to the candidate with various schemes that reason about the conditional constructs in the program (i.e. if checks). In our descriptions, we refer to the conditional expression in an if check as the *if predicate* while we refer to both the conditional expression and then-else statements as the *if body*. Our

analysis considers a candidate unsafe if there still exist unprotected paths to the candidate after considering the scheme. For example, Figure 5.7 illustrates a data-flow example with Log4j (CVE-2021-44228). The diagram shows data-flow paths from the program input *source isr* to the candidate function, `lookup`, through intermediate if checks such as *if predicate 1* in functions outside the method. The data-flow path then enters `lookup` via the method argument, *name*, and ends up at the sink call. Note that following the call chain from this sink would lead to a key system-level execution library function (i.e. `getObjectFactoryFromReference`) as determined by the vulnerable candidate discovery analysis, referenced in Table 5.1. Nevertheless, one also can observe a complete path without any if checks from *source isr* to `lookup` via *subpath 2*, and then from `lookup` to the sink through *subpath B*. Since there is at least one unfiltered path from *source isr* to the sink, XScope would mark `lookup` as vulnerable.

Furthermore, XScope uses filtering strategies that reason about if predicates to deal with false positives and false negatives.

Filtering Strategies

Overall, XScope filters a candidate if there only exist data-flow paths through *security-related if-predicates* outside the candidate function or *non-trivial if-predicates* in the candidate function when flowing from a user input source to the candidate sink. We explain the details of the filtering strategies as follows.

False Alarm Filtering Many false alarms can cause developers to become frustrated so our scheme algorithm takes steps to filter them out. However, the challenge is finding code patterns that achieve a good trade-off between filtering the false positives instead of the true positives. We primarily consider security-related checks that have certain sensitive operations along the data-flow paths, outside the last hop. If the candidate is protected from vulnerable data-flow paths by security-related checks then it should actually be safe so we prune such paths.

The Figure 5.7 diagram shows two subpaths, *subpath 1* and *subpath 2*, outside the candidate method, `lookup`. If *if predicate 1* on *subpath 1* contains the security sensitive operations like `matches`, then XScope would filter the *subpath 1* as that path performs a security-related check. XScope would consider `lookup` as unprotected not because of *subpath 1* but due to *subpath 2* followed by *subpath A*.

Although the operations may be used in normal non-security scenarios, we find that in-

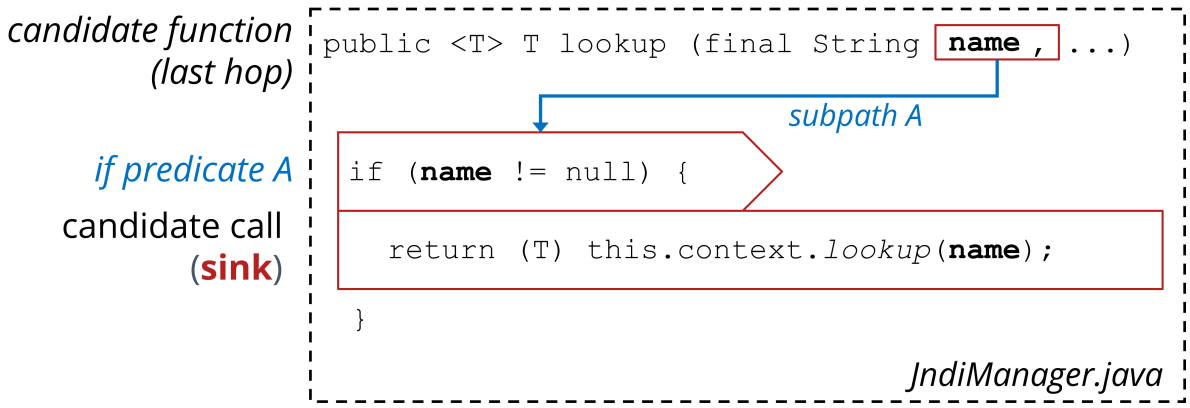

Figure 5.8 Data-flow scenario with a trivial check in the candidate function. XScope filters out paths with non-trivial if predicates to avoid missing true positives.

corporating them on the vulnerable paths of sensitive sources and candidates reduces false positive paths without missing true vulnerabilities. The results are discussed in section 5.4.

False Negative Filtering XScope also requires an if check in the final candidate hop for the candidate to be considered safe so it filters out data-flow paths through checks within the candidate hop. However, this would mean that any trivial checks within the candidate that certainly do not have a security purpose can inaccurately filter the remaining sensitive data-flow paths and cause the vulnerable function to not be detected. To avoid false negatives, we omit trivial checks from filtering data-flow paths.

For instance, suppose the candidate in the data-flow example (depicted in Figure 5.8) contained a non-trivial check encapsulating the sink as in Figure 5.8. All data-flow paths have to flow through the *if predicate A* to reach the sink so they would all be filtered out. This results in a false negative, although the predicate is not performing security checks.

In this work, we filter predicates that make a simple null comparison as they do not play a security role in preventing improper code execution vulnerabilities. Instead, these checks ensure that a function parameter is not null so that the function can execute without failure.

From data-flow analysis, we extract the number of unrestricted data-flow paths for each vulnerable function candidate as it measures the degree of data-flow exposure. We rank the candidates by this number of unrestricted data-flow paths in descending order.

Table 5.2 List of security sensitive API.

CVE ID	Application	Vulnerable Function, and Key System-Level Execution Library Function
CVE-2021-44228	Apache Log4j	`org.apache.logging.log4j.core.net.JndiManager.lookup()` `javax.naming.spi.NamingManager.getObjectFactoryFromReference()`
CVE-2017-7504	JBoss (Commons)	`org.apache.commons.collections.InvokerTransform.transform()` `java.lang.reflect.Method.invoke()`
CVE-2019-0232	Apache Tomcat	`org.apache.catalina.servlets.CGIRunner.run()` `java.lang.Runtime.exec()`
CVE-2017-7525	Jackson-databind	`com.fasterxml.jackson.databind.deser.impl.SetterlessProperty.deserializeAndSet()` `java.lang.reflect.Method.invoke()`
CVE-2020-8840	Jackson-databind	`com.fasterxml.jackson.databind.deser.impl.MethodProperty.deserializeAndSet()` `java.lang.reflect.Method.invoke()`
CVE-2019-0221	Apache Tomcat	`org.apache.catalina.ssi.SSIExec.process()` `java.lang.Runtime.exec()`
CVE-2019-1003000	Jenkins (Groovy)	`org.codehaus.groovy.reflection.CachedMethod.invoke()` `java.lang.reflect.Method.invoke()`
CVE-2020-26258	XStream	`com.thoughtworks.xstream.converters.collections.MapConverter.putCurrentEntryIntoMap()` `java.lang.reflect.Method.invoke()`

Table 5.3 List of explored real-world vulnerabilities with unique root cause functions.

CVE ID	Application	CVSS v3 Score
CVE-2021-44228	Apache Log4j	10.0
CVE-2017-7504	JBoss (Commons)	9.8
CVE-2019-0232	Apache Tomcat	8.1
CVE-2020-8840	Jackson-databind	9.8
CVE-2017-7525	Jackson-databind	9.8
CVE-2019-0221	Apache Tomcat	6.1
CVE-2019-1003000	Jenkins (Groovy)	8.8
CVE-2020-26258	XStream	7.7
Count	8	

5.3 Evaluation

5.4 Experimental Evaluation

In this section, we present our evaluation methodology, assess XScope against alternative schemes, and discuss the case studies.

5.4.1 Evaluation Methodology

Vulnerabilities studied

We study eight real-world vulnerabilities in six popular Java applications used in cloud server systems. The applications consist of various software types including web application servers like Apache Tomcat, data processing libraries for various object formats

like FasterXML Jackson-databind and XStream, and logging frameworks like Apache Log4j, etc. Table 5.3 lists the studied vulnerabilities recorded by the common vulnerabilities and exposures (CVE) database. The table includes the CVE ID, the application name, and the common vulnerability scoring system (CVSS) version 3.0 score. For certain vulnerabilities, the vulnerable function lies in dependent libraries so we highlight the libraries in parentheses under the application column.

We begin by examining the bug collection highlighted by the Tunde-Onadele et al. (2022) study. The collection lists Java code execution vulnerabilities in a category known as *improper execution restrictions* with CVE information and related resources such as links to patches and exploits when available. We refer to the national vulnerability database (NVD) NIST (2023) as well as other databases like Bugzilla Red Hat (2021) and Veracode Veracode (2021). We select CVEs that have analysis information or available bug exploits so that we can understand the vulnerable function causes. In Table 5.2, we present the vulnerable functions for our studied bugs along with the key system-level execution library functions they invoke. We observe that the majority (62.5%) of the bugs primarily execute by calling the `invoke` key system-level execution function, followed by `exec` (25%), and then `getObjectFactoryFromReference` (12.5%).

Experiment Setup

Our experiments are conducted on machines running Ubuntu 20.04 with four 3.40GHz cores and 16GB memory. We use the following tools:

Call graph analysis with Soot Vallée-Rai et al. (2010) During vulnerable candidate discovery, we implement the call graph analysis by performing a whole program analysis with Soot (version 2.5). Soot transforms the Java program bytecode to Jimple intermediate representation (IR) after constructing the call graph via its whole-jimple transformation pack (wjtp) analysis. We specify the *all-reachable* parameter to consider all application class methods as reachable so they exist in the generated call graph we analyze. In addition, we use the *safe-newinstance* option to conservatively find constructors that can be called via Java's `Class.newInstance()` method. To identify vulnerable function candidates, we search for functions outside of the standard Java libraries. Thus, we exclude functions with library prefixes like the following: `java`, `sun`, `com.ibm`, etc. We empirically find that the vulnerable functions are discovered in the candidate output list with a maximum search depth of seven.

Table 5.4 XScope detection result comparison with vulnerable candidate discovery (VCD) only and related work, FindSecBugs.

CVE ID	Application	Detected			False Positive Rate		
		VCD	FindSecBugs	**XScope**	VCD	FindSecBugs	**XScope**
CVE-2021-44228	Apache Log4j	✓	✓	✓	5.95%	1.81%	0.00%
CVE-2017-7504	JBoss (Commons)	✓	✗	✓	5.10%	-	0.67%
CVE-2019-0232	Apache Tomcat	✓	✓	✓	6.83%	1.64%	0.94%
CVE-2017-7525	Jackson-databind	✓	✗	✓	8.89%	-	1.43%
CVE-2020-8840	Jackson-databind	✓	✗	✓	8.89%	-	1.43%
CVE-2019-0221	Apache Tomcat	✓	✓	✓	11.10%	1.64%	1.44%
CVE-2019-1003000	Jenkins (Groovy)	✓	✗	✓	7.19%	-	1.74%
CVE-2020-26258	XStream	✓	✗	✓	13.31%	-	2.85%
Detection Rate / Average False Positive Rate		100%	37.5%	100%	8.41%	1.70%	1.31%

Table 5.5 Summary of the alternative filtering schemes compared to XScope.

Scheme	Vulnerable Candidate Discovery	Filtering outside the last hop	Filtering inside the last hop
VCD	✓	N/A	N/A
SCF-Naive	✓	None	None
SCF-Last Hop	✓	None	Non-trivial if predicate
SCF-Full Path	✓	Security-related if predicate	Non-trivial if body
XScope	✓	Security-related if predicate	Non-trivial if predicate

Data-flow analysis with CodeQL GitHub, Inc. (2021) In the safe candidate filtering module, we perform data-flow analysis with the code analysis engine, CodeQL (version 2.8.2). CodeQL first builds the program source code into a CodeQL database to allow one to explore the codebase via queries written in the QL query language. We specify the kind property of our queries as *path-problem* to be able to examine the resulting data-flow paths. As discussed in 5.2.4, we cover *ReadObjectMethod* and *UserInput* flow sources. However, we modify the *UserInput* class defined in `semmle.code.java.dataflow.FlowSources` to include useful *ServletRequest* and *Autocloseable* resource objects. Finally, the security-related functions we filter along the data-flow paths are: `equals`, `equalsIgnoreCase`, `contains`, and `matches`.

Alternative approaches

We evaluate our approach against alternative static analysis security solutions that examine Java applications.

Table 5.6 Detection result comparison across the alternative schemes.

CVE ID	Detected				False Positive Rate			
	SCF-Naive	SCF-LH	SCF-FP	**XScope**	SCF-Naive	SCF-LH	SCF-FP	**XScope**
CVE-2021-44228	✓	✓	✓	✓	0.00%	0.00%	0.22%	0.00%
CVE-2017-7504	✓	✓	✓	✓	1.71%	0.73%	0.67%	0.67%
CVE-2019-0232	✓	✓	✗	✓	1.44%	1.05%	-	0.94%
CVE-2017-7525	✓	✓	✓	✓	1.80%	1.46%	1.06%	1.43%
CVE-2020-8840	✓	✓	✓	✓	1.80%	1.46%	1.06%	1.43%
CVE-2019-0221	✓	✓	✗	✓	1.88%	1.44%	-	1.44%
CVE-2019-1003000	✓	✓	✓	✓	2.42%	1.85%	1.51%	1.74%
CVE-2020-26258	✓	✓	✓	✓	3.60%	2.99%	2.24%	2.85%
Detection Rate / Average False Positive Rate	100%	100%	75%	100%	1.83%	1.37%	1.13%	1.31%

FindSecBugs (Spotbugs security plugin) OWASP (2023); SpotBugs (2023). Spotbugs is the successor of the Findbugs work Hovemeyer and Pugh (2004), a Java tool for detecting bug patterns that can incorporate data-flow and control-flow techniques. Although it has over 400 bug patterns, they do not properly address security issues out-of-the-box. Thus, Find Security Bugs (FindSecBugs) is the plugin designed to focus on security bugs with 141 patterns. For a fair comparison, we filter the patterns to the 64 that are relevant to improper execution restrictions vulnerabilities. FindSecBugs leverages techniques like taint analysis to perform its detection. We run our tests on Spotbugs version 3.1.5 available on the Eclipse (2020-06) marketplace, and FindSecBugs version 1.12.0.

Alternative Filtering Schemes We define alternative filtering schemes as follows and outline their characteristics in Table 5.5. In our descriptions, we refer to the conditional expression in an if check as the *if predicate* while we refer to both the conditional expression and then-else statements as the *if body*.

- *Vulnerable Candidate Discovery (VCD) only:* This scheme represents the candidates returned from the vulnerable candidate discovery stage alone, which performs the call graph based analysis.

- *Safe Candidate Filtering - Naive (SCF-Naive):* In the safe candidate filtering component, this scheme filters a candidate only if no data-flow paths exist from a user input source to the candidate sink. SCF-Naive performs no further filtering based on predicates.

- *Safe Candidate Filtering - Last Hop (SCF-LH):* This scheme filters a candidate if there only exist data-flow paths through a non-trivial if predicate in the candidate function when flowing from a user input source to the candidate sink. In other words, SCF-LH

filters the paths flowing through non-trivial if predicates in the candidate function (last hop).

- *Safe Candidate Filtering - Full Path (SCF-FP):* This scheme filters a candidate if there only exist data-flow paths through security-related if-predicates outside the candidate function or non-trivial if-body in the candidate function when flowing from a user input source to the candidate sink. SCF-FP filters the paths that flow through security-related if predicates outside the last hop and non-trivial if bodies in the last hop.

Evaluation Metrics

We calculate the percentage of CVEs where the vulnerable function is detected out of all tested CVEs as the *detection rate*. In addition, we record the number of false alarms above and equal to the rank of the vulnerable function to demonstrate its rank. For each CVE, we calculate the percentage of these false alarms compared to all the methods present in the examined source code as the *false positive rate*.

The detection rate and false alarm rate are given by equations (5.1) and (5.2), respectively.

$$Detection\,Rate = \frac{\text{detected CVEs}}{\text{detected CVEs} + \text{missed CVEs}} \tag{5.1}$$

$$False\,Positive\,Rate = \frac{\text{number of false alarm methods}}{\text{number of codebase methods}} \tag{5.2}$$

5.4.2 Results Analysis

In this section, we present the analysis of the detection results. We compare the XScope detection results to those of the alternative approaches and discuss our detection challenges.

Detection Results

First, we discuss the XScope results compared to the FindSecBugs work and the vulnerable candidate discovery (VCD) only scheme, summarized in Table 5.4. The table shows the detection status and the false positive rate for each tool over the CVEs.

VCD represents the result of the call graph based search from the key system-level functions that can execute commands. Although VCD includes the vulnerable function

in the candidate list, it has many false alarms as it conservatively finds potential methods that lead to key system-level code execution functions. FindSecBugs has a 79.7% lower average false positive rate than VCD but only detects 3 out of 8 CVEs (37.5%). Whereas, the results show that XScope is able to achieve 100% detection rate with the lowest number of false alarms. XScope reduces the false positive rate of VCD and FindSecBugs by 84.4% and 22.9% on average. In the cases where both XScope and Findbugs methods detect the CVE, XScope attains 53.3% lower average false positive rate than Findbugs. FindSecBugs examines applications with static methods including taint analysis, a special case of data-flow analysis. However, XScope is able to outperform such analysis with efficient vulnerable code execution patterns and filtering techniques.

We notice that XScope performs well for cases like Apache Log4j (CVE-2021-44228), where it accurately ranks the vulnerable function first. For CVEs like Tomcat CVE-2019-0221 with large codebases of over 19800 methods, although XScope can reduce the false alarms to 1.44% of the codebase, there are still 285 false alarm methods. In the best performing cases, VCD is able to initially drill down the analyzed codebase methods to about 5% before applying the safe candidate filtering phase on those candidates.

The Jackson-databind CVEs (CVE-2017-7525, CVE-2020-8840) have similar detection results. Although their vulnerable functions are different, they reside in the same repository location (the `com.fasterxml.jackson.databind.deser.impl` package) and make similar calls. Thus, XScope discovers them at the same level during VCD and finds similar data-flow paths during SCF.

Next, we present the results for the alternative detection schemes including their detection and the false alarm rates in Table 5.6. The alternative detection schemes juxtaposed are the safe candidate filtering schemes that drill-down the vulnerable function candidates from VCD. Recall from Table 5.5 that SCF-Naive and SCF-LH do not filter data-flow paths through any predicates outside the candidate, while XScope and SCF-FP filter paths through security-related predicates. In addition, SCF-LH and XScope filter paths through non-trivial if predicates in the last hop, while SCF-FP filters through non-trivial if bodies.

In general, as the filtering level increases, the number of false positives reduces while maintaining vulnerable function detection. However, the SCF-FP filtering starts to become excessive as it results in some missed detection instances (false negatives). Thus, filtering more strictly than SCP-FP would cause more false negatives. XScope reduces the false positive rate of SCF-Naive and SCF-LH by 28.4% and 4.37%, respectively. Although SCF-FP

Table 5.7 Runtime measurements of the XScope components

XScope Module	Execution Time (per function)
Vulnerable Candidate Discovery	73.25 ± 0.31 ms
Safe Candidate Filtering	21.64 ± 0.23 s

has 13.7% lower false alarm than XScope, XScope has 33.3% higher detection rate. The results aligns with the observed patches, discussed in section 5.2, which often perform filtering in the if predicates. We observe that the statements in the if body may perform some corresponding action but do not execute the logic for security filtering.

Challenges

Overall, security vulnerability detection is a challenging endeavor, especially over mature codebases. To determine the function causes of execution vulnerabilities, one needs to understand how the complex application logic processes malicious inputs. Moreover, it is difficult to validate all false positive cases without exploits for each candidate. This means that unknown functions regarded as false positives could be vulnerabilities in practice.

In addition, although data-flow analysis tools are powerful, they have implementation challenges. During an attack, inputs may flow through multiple libraries before the data reaches the vulnerable application that contains the vulnerable function. The engine only analyzes a single codebase at a time which limits our analysis to supporting single libraries. For example, many Apache Struts 2 application versions have exposures such as CVE-2018-11776 that are exploited with Object-Graph Navigation Language (OGNL) payloads. The attack request flows through Struts libraries like the `com.opensymphony.xwork2` package before reaching OGNL library functions that can contain the vulnerable function. Analyzing the Struts application or OGNL library alone would fail to detect the vulnerability. We plan to tackle such issues in future work.

5.4.3 Run-time Measurements

Table 5.7 summarizes the run-time for XScope and its components. The run-time depends on the codebase methods and vulnerable candidates methods analyzed so we present the average results for each function. We calculate the vulnerable candidate discovery run-time per function in the code repository while we calculate the safe candidate filtering run-time

per candidate function.

XScope can be run offline before server applications are released to production environments to avoid strict resource constraints. Nevertheless, users may find it useful to adjust the configuration options of the call graph analysis done during vulnerable candidate discovery. In addition, we currently use the CodeQL data-flow tool out of the box, which may be optimized. Teams may alleviate the data-flow run-time by running XScope on smaller parts of the codebase as the repository grows following prior testing.

5.4.4 Case Study

In this subsection, we primarily discuss the XScope detection details for the specific vulnerabilities.

CVE-2021-44228 Apache Log4j: The Log4j logging library does not properly restrict program inputs in log messages from evaluation. Thus, attackers can inject JNDI commands into a vulnerable server to cause it to receive and execute malicious classes from their external servers.

The vulnerable function for this CVE is located in `JndiLookup.lookup()`. During vulnerable candidate discovery, XScope locates the vulnerable function seven hops from `getObjectFactoryFromReference` and also later finds it when tracing from `newInstance`. Finally, after safe candidate filtering, XScope detects `lookup` as the highest ranked vulnerable function candidate with 75 vulnerable paths.

FindSecBugs finds `lookup` under the `LDAP_INJECTION` pattern in the 1.12.0 version was released after the public disclosure of the bug.

CVE-2017-7504 JBoss (Commons): The Jboss implementation of the Java Message Service (JMS) has an HTTP Invocation layer that allows deserialization of inputs without restriction. Thus, the application can read malicious objects an attacker inserts through the dependent Apache Commons library. We inspect Jboss under attack to verify that the vulnerable function is `org.apache.commons.collections.InvokerTransform-.transform()`.

The function directly calls the `invoke` Key execution function, so XScope finds `transform` as a candidate in the call graph at a depth of one. We observe that during the attack, Jboss calls the `readObject` method of `java.io.ObjectInputStream`, which aligns with the semmle *ReadObjectMethod* source of CodeQL. XScope determines the `transform` function as vulnerable with 21 unprotected paths.

FindSecBugs highlights the entire *InvokerTransformer* class for potential deserialization but does not narrow down the vulnerable function(s).

CVE-2017-7525 Jackson-databind: Jackson-databind is vulnerable to RCE attacks when using the deserialization features of the `readValue` method of the *ObjectMapper* class. During an exploit, we observe the vulnerable function `deserializeAndSet` of the *SetterlessProperty* class calling `invoke` as the next hop. XScope finds the vulnerable function `deserializeAndSet` as a direct caller of `invoke`. Nevertheless, XScope also traces the function from `getObjectFactoryFromReference` and `exec` at further depths. XScope labels the function as vulnerable with over 280 unfiltered paths from a variety of data-flow sources including *AutoCloseable* resources.

Spotbugs equipped with FindSecBugs highlight the *SetterlessProperty* class for non-security reasons like inheritance style issues but not for security reasons.

CVE-2020-8840 Jackson-databind: Jackson-databind is also affected by CVE-2020-8840. Its block-list mechanism is missing restrictions for certain JNDI-related objects during deserialization which leads to arbitrary code execution. Similar to the CVE-2017-7525 case, the vulnerable function named `deserializeAndSet` calls `invoke` but resides in a different class, *MethodProperty*.

XScope discovers `deserializeAndSet` one function call away from the Java `invoke` method. XScope detects the vulnerable function with the same number of unfiltered paths rank as that of Jackson-databind CVE-2017-7525.

FindSecBugs also does not find security issues related to the vulnerable class for this Jackson-databind CVE.

CVE-2019-0232 Apache Tomcat: When the Tomcat Common Gateway Interface (CGI) protocol is enabled with the *enableCmdLineArguments* parameter, servers can execute scripts via URL parameters. However, the vulnerability allows attackers to inject arbitrary commands as the JRE passes on command line arguments in Windows.

Since the vulnerable function, `CGIServlet$CGIRunner. run()`, immediately calls `exec`, XScope discovers a depth of one from a key system-level execution library function. XScope traces data-flow paths especially from client request input parameters of the Java Servlets API (*ServletRequest* in the semmle library) to find more than 50 unfiltered paths.

FindSecBugs finds the vulnerable CGIServlet$CGIRunner function under the `COMMAND_INJECTION` pattern.

CVE-2019-0221 Apache Tomcat: The vulnerable Apache Tomcat version with server

side includes (SSI) and the printenv configuration directive allows input commands to print and execute by default. Without properly sanitizing inputs, the application is exposed to exploits like cross-site scripting (XSS) attacks.

XScope finds the vulnerable function, `SSIExec.process()`, via the `exec` key system-level execution function at a depth of three. XScope also identifies data-flow paths from *ServletRequest* input parameters. Thus, XScope detects the vulnerable function with over 50 unprotected paths.

FindSecBugs finds the vulnerable function, process, under its `COMMAND_INJECTION` pattern.

CVE-2020-1003000 Jenkins (Groovy): Jenkins allows authorized users to execute pipeline build scripts written in Groovy to support application development, However, this Jenkins exposure allows attackers to bypass the sandbox protection and execute scripts even on master nodes. Attackers use Groovy abstract syntax tree (AST) transformations to override blacklisted packages.

During vulnerable execution, the call chain the `invoke` function of the Groovy *Cached-Method* class. The vulnerable function is discovered one hop away from the Java execution function, `invoke`. Thereafter, XScope detects the function with 51 vulnerable data-flow paths.

FindSecBugs does not raise alerts for this vulnerable class and function.

CVE-2020-26258 XStream: The XStream deserialization library uses a class blocklist when converting XML data to Java objects. However, attackers can use unrestricted classes to prompt the server to request unintended server-side resources.

The vulnerable function `putCurrentEntryIntoMap`, from the *MapConverter* class, starts the Java invocations that eventually reach the `invoke` key execution function. XScope finds the vulnerable function at a depth of five. The XScope call graph report also includes unrestricted classes like `jdk.nashorn.internal.objects.NativeString` on the vulnerable call path, which can inform developer patches. After safe candidate filtering, XScope reports `putCurrentEntryIntoMap` as vulnerable with 72 unrestricted data-flow paths.

However, FindSecBugs does not recognize any functions in *MapConverter* or in related classes.

5.5 Summary

In this chapter, we present XScope, a new pattern-driven fine-grained vulnerability detection framework for proactively protecting cloud server systems from security bugs due to improper code execution restrictions. XScope can not only detect code execution vulnerabilities but also localize the vulnerable functions in complex large-scale cloud server programs consisting of tens of thousands of functions. XScope leverages general patterns extracted from known code execution vulnerabilities and patches to achieve a higher detection rate than simple rule-based detection approaches. Furthermore, XScope combines call graph analysis and data-flow analysis to minimize false positive rates while maintaining a high detection rate. We have implemented a prototype of XScope and tested it using real world vulnerabilities including the high impact Log4j vulnerability on six commonly used cloud server systems. Our experimental results show that XScope can achieve a 100% detection rate while existing security checking tools like FindSecBugs can only detect 38% of those CVEs. Moreover, XScope can reduce the false positive rate by 53% for those CVEs that can be detected by both XScope and FindSecBugs.

CHAPTER

6

RELATED WORK

The research is related to the following areas: intrusion detection, vulnerability categorization, and vulnerability detection.

6.1 Intrusion Detection

IDS in the literature detect security issues based on system activity such as system metrics. We focus on comparing our work with the most related IDS work.

Shen et al. propose Tiresias using RNNs to predict security events including RCE exploit steps Shen et al. (2018). However, it requires a large amount of labeled training data to detect similar attacks. Lin et al. propose SHIL Lin et al. (2022) and CDL Lin et al. (2020) to perform security attack detection in containerized environments with the goal of lowering false alarms. Beehive leverages mining techniques over security logs from heterogeneous security products to produce consistent incident reports Yen et al. (2013). Beehive focuses on identifying malware infections and suspicious enterprise activities. Droidscribe Dash et al. (2016) monitors system call run-time information to detect Android malware (including

file access and execution malware) using a support vector machine (SVM)-based multi-class classifier. In contrast, we identify code patterns before the application is released. In comparison, XScope takes a proactive approach to detecting security vulnerabilities before attacks can occur.

6.2 Vulnerability Categorization

Zhou et al. study malware samples in Android applications and reveal shortcomings of antivirus tools against major categories of malware Zhou and Jiang (2012). In contrast, we focus on the underlying vulnerabilities at the code level. Tsipenyuk et al. focus on classifying coding problems called phylla into seven plus one kingdoms to help developers avoid such error themes Tsipenyuk et al. (2005). For example, specific missing buffer checks under the buffer overflow phyllum is classified under the input validation kingdom. In addition, Xu et al. find five security vulnerability patterns in C code patches Xu et al. (2017). Chen et al. study security bugs in the Linux kernel, written in C Chen et al. (2011). Compared to these papers, we focus on vulnerabilities in Java-based cloud systems. We also present code patterns for each category. Meng et al. investigate posts from the popular StackOverflow forum platform to understand coding issues with Java security libraries Meng et al. (2018). This work focuses on the proper use of security APIs, which may address some *improper permission checks* bugs but does not cover the other vulnerability categories we consider.

The common weakness enumeration (CWE) list is an extensive manual community-based effort to document software and hardware errors Corporation (2021). It provides a taxonomy of errors, sometimes with broad code examples, potential mitigation options, and detection approaches. The CWE system presents a database of the weaknesses but does not give the precise root causes for specific CVE-identified vulnerabilities. We locate the vulnerable code for each CVE to find patterns (e.g. relevant Java functions) that help one locate and understand the vulnerability. In contrast, our work characterizes software security bugs with vulnerable code patterns to efficiently detect the vulnerabilities that fall under each category. The code patterns emphasize core functions to help locate the vulnerable functions. For example, according to the NVD NIST (2023), the Log4j vulnerability (CVE-2021-44228) has three weaknesses: deserialization of untrusted data (CWE-502), uncontrolled resource consumption (CWE-400), and improper input validation (CWE-20). We can focus on the vulnerable function pattern of the *improper execution restrictions*

category to detect this command execution bug. Although in the example NVD assigns three weaknesses, the weaknesses do not give further information about locating the issue in the code. Moreover, one weakness 'uncontrolled resource consumption' is not involved in fixing the core command execution issue, while the location and interaction of the other two ('improper input validation' and 'deserialization of untrusted data') still need to be connected to understand the issue.

6.3 Vulnerability Detection

Data-flow Analysis. Livshits et al. target vulnerabilities in Java web applications that lead to control-hijacking attacks like SQL injections and cross-site scripting Livshits and Lam (2005). They improve taint analysis precision by implementing an object naming scheme that avoids unnecessary tainting of strings. Their work only targets unchecked inputs but does not analyze the predicates so they would miss errors in existing checks. The authors focus on control-hijacking attacks and not all of the *improper execution restrictions* vulnerabilities. Taintdroid leverages taint analysis tracking to trace how privacy-sensitive data leaves Android applications Enck et al. (2014). However, they are only concerned with APIs that transmit sensitive sensor data over networks not execution functions. Taintdroid only tracks explicit data-flow on Dalvik bytecode at the instruction-level. It does not support comparing instructions to reason about data-flow if-checks. Zheng et al. Zheng and Zhang (2013) tackle remote code execution attacks that execute input scripts in PHP web applications. The authors analyze the behavior of string and non-string inputs separately across multiple requests and sessions that incorporate a string solver and a satisfiability modulo theories (SMT) solver to derive taint analysis solutions. Their taint analysis traces client inputs to file writes and dynamic script evaluations. However, they do not consider other sensitive sinks like object deserialization functions in Log4j.

Vanguard examines C and C++ code for general missing check vulnerabilities Situ et al. (2018). They perform taint analysis to find how outside sources reach sensitive operations, followed by data-flow to find missing checks. The authors evaluate Vanguard using known vulnerable functions in the application. Yamaguchi et al. propose Chucky to improve manual code auditing of input validation vulnerabilities by finding abnormal checks that deviate from regular checks around sensitive functions Yamaguchi et al. (2013). Chucky uses static taint analysis to extract features from conditional expressions that machine

learning techniques can use to perform anomaly detection.

TAJ is a taint analysis security solution targeted to large-scale industry applications Tripp et al. (2009). TAJ performs taint tracking on a hybrid system dependence graph (HSDG), where nodes are load, store, or call statements, and edges are added using dependence context from a call graph. The call graph construction prioritizes parts of the web application that are close to the taint source and methods that are assigned high priorities. Balzarotti et al. propose Saner which performs static taint analysis with data-flow information to detect incorrect input sanitization. Balzarotti et al. (2008). They combine the static analysis with the dynamic approach of testing a suite of cross-site scripting and SQL injection attacks to reduce false positives. Balzarotti et al. model strings with finite automata that indicate a taint status for each character. Any paths with suspicious string values are marked for further analysis. The FindSecBugs OWASP (2023) security package improves upon this as a plugin of the industry-based Findbugs successor called Spotbugs SpotBugs (2023). FindSecBugs includes taint analysis tactics to detect security bugs.

In comparison, XScope leverages both control-flow and data-flow analysis to overcome the limited scope of rule-based detection schemes and high false positives of data-flow analysis-only approaches.

Control-flow Analysis Thomé et al. propose a symbolic execution approach called ACO-Solver to solve complex string operations involved in input sanitization conditions of Java web applications Thomé et al. (2017a). ACO-Solver uses symbolic execution to search for code that matches the specified expert-provided attack constraints. ACO-Solver leverages existing state-of-the-art constraint-solvers to reduce the search space. If the solvers fail, it then runs an automata-based string constraint solver guided by an ant-colony optimization heuristic. ACO-Solver is designed for injection and XSS vulnerabilities. In contrast, XSCope can detect a broader range of *improper execution restrictions* vulnerabilities.

Hybrid approaches. JoanAudit Thomé et al. (2017b) automates the software auditing process by locating and inserting missing security checks between input program sources and sinks. It constructs and prunes a system dependency graph (SDG) with data-flow, control-flow, and call dependencies from Java bytecode, using a predefined list of input sources, sanitation procedures, non-security related functions, and sensitive sink functions. Fratantonio et al. propose TriggerScope that finds Android malware called logic bombs that trigger under very specific check conditions Fratantonio et al. (2016). TriggerScope combines taint analysis and symbolic execution by building a super control flow graph (sCFG) consisting of both interprocedural and intraprocedural CFG, which they annotate

with symbolic execution to trace how inputs flow to sensitive Android operations. TriggerScope focuses on Android API relevant to mobile device privacy like permissions or file-system operations related to time, location, and SMS messages. In comparison, XScope aims at detecting generic code execution vulnerabilities with a fine-grained pattern-driven approaches.

Related work in vulnerability detection use static and symbolic execution program analysis techniques. Balzarotti propose Saner which performs static taint analysis with data-flow information to detect incorrect input sanitization. Balzarotti et al. (2008). They combine the static analysis with the dynamic approach of testing a suite of cross-site scripting and SQL injection attacks to reduce false positives. Frantantonio et al build a super control flow graph (sCFG) consisting of both interprocedural and intraprocedural CFG, which they annotate with symbolic execution to trace how inputs flow to sensitive Android operations Fratantonio et al. (2016). Yamaguchi et al. propose Chucky to improve manual code auditing of input validation vulnerabilities by finding abnormal checks that deviate from regular checks around sensitive functions Yamaguchi et al. (2013). Chucky uses static taint analysis to extract features from conditional expressions that machine learning techniques can use to perform anomaly detection.

Xu et al. perform a semantic analysis of application binaries using control flow graph techniques Xu et al. (2017). Their work finds five security vulnerability patterns in C code used to uncover vulnerabilities in other applications. Livshits et al. use taint tracking to find violations given by process query language (PQL) specifications that correspond to attacks due to unchecked inputs in Java applications Livshits and Lam (2005). Enck et al. also leverage taint analysis tracking to reveal how privacy-sensitive data leave Android applications with practical tradeoff between performance overhead and precision Enck et al. (2014). Thomé et al. use symbolic execution to improve the efficiency of solving complex string operations involved in matching code against expert-provided specifications for injection and cross site scripting (XSS) vulnerabilities Thomé et al. (2017a). Shen et al. propose VulnLoc which assigns probabilities to instructions to determine vulnerable function locations for fixing vulnerabilities. They perform fuzzing using a single exploit, to generate other paths that help VulnLoc avoid overfitting their models. Some works combine program analysis with machine learning to detect vulnerabilities. Pang et al. detect vulnerable classes with supervised models, namely support vector machines (SVM) Pang et al. (2015). They use frequency vectors of n-grams of Java keywords and remove features that are not significantly different between vulnerable and non-vulnerable classes.

Some work use static analysis to detect other non-security issues like API misuse. Wan et al. build static checkers that mainly use interprocedural data-flow analysis to analyze how variables of misused ML API calls flow across functions Wan et al. (2021). Wang et al. Wang (2021) study error-prone early exit (EE) paths in detecting memory leaks. They highlight new EE scenarios to consider when checking for missing memory deallocation. DScope Dai et al. (2018) performs intraprocedural analysis to identify loop conditions related to Java I/O APIs that cause the system to hang. Hangfix He et al. (2020) performs intraprocedural dataflow analysis to detect and fix certain root cause function patterns of software hang bugs. XRay combines information flow analysis with record and replay to locate performance bugs caused by misconfiguration. XRay finds time points where system metric performance costs are high to evaluate the likelihood that certain inputs and functions cause the bugs. ESG uses static analysis and symbolic execution to find program constraints that reproduce concurrency bugs. ESG relies on information like problematic program counter values provided in bug reports. Compared to these work, our study focuses on understanding security bugs.

CHAPTER

7

CONCLUSION

In this dissertation, we develop frameworks for automatically diagnosing and fixing security bugs supported by comprehensive studies. We contribute a self-triggering targeted patching framework for containerized systems with insights from our vulnerability exploit detection study, and a pattern-driven fine-grained vulnerability detection framework for cloud server systems with our understanding of security vulnerability patterns. We make the following specific contributions:

- We present a comparative study on the effectiveness of various vulnerability detection schemes for containers. Specifically, we evaluate a set of static and dynamic detection schemes using 28 real world vulnerabilities that widely exist in docker images. Our results show that the static vulnerability scanning scheme only detects 3 outs of 28 tested vulnerabilities and the dynamic anomaly detection schemes detect 22 vulnerability exploits. Combining static and dynamic schemes can further improve the detection rate to 86% (i.e., 24 out of 28 exploits). We also observe that the dynamic anomaly detection scheme can achieve more than 20 seconds lead time (i.e., a time window before attacks succeed) for a group of commonly seen attacks in containers

that try to gain a shell and execute arbitrary code.

- We present Self-Patch, a new self-triggering targeted patching framework. To achieve this goal, the Self-Patch framework consists of three coordinating components: 1) an online attack detection module which can dynamically detect abnormal attack activities by extracting feature vectors from system call traces and applying unsupervised machine learning methods over the features; 2) an attack classification scheme which classifies a detected attack into a specific type linked to a certain CVE; and 3) a targeted patch execution module which can install proper software patches to fix the vulnerability. We have implemented a prototype of Self-Patch and evaluated it over 31 real-world vulnerabilities discovered in 23 common server applications. Our experimental results show we can increase detection rate to over 80% and reduce false alarm rate to 0.7%. In contrast, traditional whole software upgrade schemes can either only detect 6% attacks or incur more than 20% false alarms. Self-Patch can also reduce the memory overhead by up to 84% and disk overhead by up to 40%.

- We present a comprehensive study over 110 recent real world security bugs in 13 popular cloud server systems. Our study first identifies five common vulnerability categories among those 110 studied security bugs: 1) *improper execution restrictions*, 2) *improper permission checks*, 3) *improper resource path-name checks*, 4) *improper sensitive data handling*, and 5) *improper synchronization handling*. Furthermore, we extract key software code patterns in each category and describe a set of pattern-driven strategies for detecting those security bugs before they are released to production cloud environments.

- We present XScope, a new pattern-driven fine-grained vulnerability detection framework for proactively detecting security bugs due to improper code execution restrictions. XScope not only detects code execution vulnerabilities but also localizes the vulnerable functions in complex large-scale cloud server programs consisting of tens of thousands of functions. Furthermore, XScope combines call graph analysis and data-flow analysis to minimize false positive rates while maintaining a high detection rate. We have implemented a prototype of XScope and tested it using real world vulnerabilities including the high impact Log4j vulnerability on six commonly used cloud server systems. Our experimental results show that XScope can achieve a 100% detection rate while existing security checking tools like FindSecBugs can only detect

38% of those CVEs. Moreover, XScope can reduce the false positive rate by 53% for those CVEs that can be detected by both XScope and FindSecBugs.

REFERENCES

Altman, N. S. (1992). An introduction to kernel and nearest-neighbor nonparametric regression. *The American Statistician*, 46(3):175–185.

Apache JMeter (2018). https://jmeter.apache.org/.

Balzarotti, D., Cova, M., Felmetsger, V., Jovanovic, N., Kirda, E., Kruegel, C., and Vigna, G. (2008). Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 387–401. IEEE.

Banyan Collector (2018). https://github.com/banyanops/collector.

Baudoin, C., Dekel, E., and Edwards, M. (2014). Interoperability and portability for cloud computing: a guide. *Cloud Standards Customer Council*, 1(1):1–31.

Bettini, A. (2015). Vulnerability exploitation in Docker container environments. https://www.blackhat.com/docs/eu-15/materials/eu-15-Bettini-Vulnerability-Exploitation-In-Docker-Container-Environments-wp.pdf.

Burp Suite Scanner (2018). https://portswigger.net/burp.

Carter, E. (2018). 2018 docker usage report. https://sysdig.com/blog/2018-docker-usage-report.

Chen, H., Mao, Y., Wang, X., Zhou, D., Zeldovich, N., and Kaashoek, M. F. (2011). Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, pages 1–5.

Clair (2017). https://github.com/coreos/clair.

Cloud, G. (2023). Advantages of cloud computing | google cloud. https://cloud.google.com/learn/advantages-of-cloud-computing.

Corporation, T. M. (2021). Common weakness enumeration. https://cwe.mitre.org.

Corporation, T. M. (2023). Mitre att&ck®. https://attack.mitre.org/.

CVE Details (2023). https://www.cvedetails.com.

Cybersecurity and Agency, I. S. (2023). 2022 top routinely exploited vulnerabilities - CISA. https://www.cisa.gov/sites/default/files/2023-08/aa23-215a_joint_csa_2022_top_routinely_exploited_vulnerabilities.pdf.

Dai, T., He, J., Gu, X., Lu, S., and Wang, P. (2018). Dscope: Detecting real-world data corruption hang bugs in cloud server systems. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 313–325.

Dash, S. K., Suarez-Tangil, G., Khan, S., Tam, K., Ahmadi, M., Kinder, J., and Cavallaro, L. (2016). Droidscribe: Classifying android malware based on runtime behavior. In *Security and Privacy Workshops (SPW), 2016 IEEE*, pages 252–261. IEEE.

Datadog (2018). 8 surprising facts about real docker adoption. https://datadoghq.com/docker-adoption.

Dean, D. J., Nguyen, H., and Gu, X. (2012). Ubl: Unsupervised behavior learning for predicting performance anomalies in virtualized cloud systems. In *Proceedings of the 9th international conference on Autonomic computing*, pages 191–200. ACM.

Docker Image Vulnerability Research (2017). https://www.federacy.com/docker_image_vulnerabilities.

Dockscan (2018). https://github.com/kost/dockscan.

Du, M., Li, F., Zheng, G., and Srikumar, V. (2017). Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1285–1298. ACM.

Enck, W., Gilbert, P., Han, S., Tendulkar, V., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. (2014). Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):1–29.

Forrest, S., Hofmeyr, S. A., Somayaji, A., and Longstaff, T. A. (1996). A sense of self for unix processes. In *Security and Privacy, 1996. Proceedings., 1996 IEEE Symposium on*, pages 120–128. IEEE.

Fratantonio, Y., Bianchi, A., Robertson, W., Kirda, E., Kruegel, C., and Vigna, G. (2016). Triggerscope: Towards detecting logic bombs in android applications. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 377–396. IEEE.

GitHub, Inc. (2021). Codeql. https://codeql.github.com/.

Gummaraju, J., Desikan, T., and Turner, Y. (2015). Over 30% of official images in docker hub contain high priority security vulnerabilities. Technical report, Technical report, BanyanOps.

He, J., Dai, T., Gu, X., and Jin, G. (2020). Hangfix: automatically fixing software hang bugs for production cloud systems. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 344–357.

Hovemeyer, D. and Pugh, W. (2004). Finding bugs is easy. *Acm sigplan notices*, 39(12):92–106.

JexBoss (2018). https://github.com/joaomatosf/jexboss.

Kanungo, T., Mount, D. M., Netanyahu, N. S., Piatko, C. D., Silverman, R., and Wu, A. Y. (2002). An efficient k-means clustering algorithm: Analysis and implementation. *IEEE Transactions on Pattern Analysis & Machine Intelligence*, (7):881–892.

Kohonen, T. (1998). The self-organizing map. *Neurocomputing*, 21(1-3):1–6.

Korn, J. (2021). The log4j security flaw could impact the entire internet. Here's what you should know. https://www.cnn.com/2021/12/15/tech/log4j-vulnerability/index.html.

Lhoták, O. and Hendren, L. (2003). Scaling java points-to analysis using s park. In *International Conference on Compiler Construction*, pages 153–169. Springer.

Lin, Y., Tunde-Onadele, O., and Gu, X. (2020). Cdl: Classified distributed learning for detecting security attacks in containerized applications. In *Annual Computer Security Applications Conference*, ACSAC '20, page 179–188, New York, NY, USA. Association for Computing Machinery.

Lin, Y., Tunde-Onadele, O., Gu, X., He, J., and Latapie, H. (2022). Shil: Self-supervised hybrid learning for security attack detection in containerized applications. In *2022 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, pages 41–50. IEEE.

Livshits, V. B. and Lam, M. S. (2005). Finding security vulnerabilities in java applications with static analysis. In *USENIX Security Symposium*, volume 14, pages 18–18.

Ltd, C. P. S. T. (2021). https://blog.checkpoint.com/2021/12/13/the-numbers-behind-a-cyber-pandemic-detailed-dive/.

Meng, N., Nagy, S., Yao, D., Zhuang, W., and Argoty, G. A. (2018). Secure coding practices in java: Challenges and vulnerabilities. In *Proceedings of the 40th International Conference on Software Engineering*, pages 372–383.

Metasploit penetration testing framework (2018). https://www.metasploit.com/.

Murphy, H. (2021). Hackers launch more than 1.2m attacks through Log4J flaw. https://www.ft.com/content/d3c244f2-eaba-4c46-9a51-b28fc13d9551.

Nayyar, A. and Kasthuri, M. (2021). How open source tools help to manage cloud infrastructure. https://www.opensourceforu.com/2021/12/how-open-source-tools-help-to-manage-cloud-infrastructure/.

NeuVector (2018). https://neuvector.com/.

NIST (2023). National vulnerability database. https://nvd.nist.gov.

Offensive Security's Exploit Database Archive (2018). https://www.exploit-db.com/.

OpenSCAP Container Compliance (2016). https://github.com/OpenSCAP/container-compliance.

OWASP (2023). Find security bugs. https://find-sec-bugs.github.io/.

Pang, Y., Xue, X., and Namin, A. S. (2015). Predicting vulnerable software components through n-gram analysis and statistical feature selection. In *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*, pages 543–548. IEEE.

Red Hat (2021). Red Hat Bugzilla. https://bugzilla.redhat.com.

SalesForce (2023). 12 benefits of cloud computing and its advantages - salesforce.com. https://www.salesforce.com/products/platform/best-practices/benefits-of-cloud-computing/.

Schmidhuber, J. (2015). Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117.

Shen, Y., Mariconti, E., Vervier, P. A., and Stringhini, G. (2018). Tiresias: Predicting security events through deep learning. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, pages 592–605. ACM.

Shu, R., Gu, X., and Enck, W. (2017). A Study of Security Vulnerabilities on Docker Hub. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pages 269–280. ACM.

Situ, L., Wang, L., Liu, Y., Mao, B., and Li, X. (2018). Vanguard: Detecting missing checks for prognosing potential vulnerabilities. In *Proceedings of the 10th Asia-Pacific Symposium on Internetware*, pages 1–10.

SpotBugs (2023). https://spotbugs.github.io/.

Sqlmap (2018). sqlmap.org/.

Sysdig (2016). http://www.sysdig.org/.

Sysdig Falco (2018). https://www.sysdig.org/falco/.

Thomé, J., Shar, L. K., Bianculli, D., and Briand, L. (2017a). Search-driven string constraint solving for vulnerability detection. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 198–208. IEEE.

Thomé, J., Shar, L. K., Bianculli, D., and Briand, L. C. (2017b). Joanaudit: A tool for auditing common injection vulnerabilities. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 1004–1008.

Tripp, O., Pistoia, M., Fink, S. J., Sridharan, M., and Weisman, O. (2009). Taj: effective taint analysis of web applications. *ACM Sigplan Notices*, 44(6):87–97.

Tsipenyuk, K., Chess, B., and McGraw, G. (2005). Seven pernicious kingdoms: A taxonomy of software security errors. *IEEE Security & Privacy*, 3(6):81–84.

Tunde-Onadele, O., Lin, Y., Gu, X., and He, J. (2022). Understanding software security vulnerabilities in cloud server systems. In *2022 IEEE International Conference on Cloud Engineering (IC2E)*, pages 245–252. IEEE.

Tunde-Onadele, O., Lin, Y., He, J., and Gu, X. (2020). Self-patch: Beyond patch tuesday for containerized applications. In *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, pages 21–27. IEEE.

Twistlock (2017). The Road to Twistlock 2.0: Runtime Radar for Runtime Defense. https://www.twistlock.com/2017/04/11/road-twistlock-2-0-runtime-radar/.

Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., and Sundaresan, V. (2010). Soot: A java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214–224.

Veracode (2021). Veracode. https://sca.veracode.com/vulnerability-database.

Vulhub: Docker-Compose File for Vulnerability Environment (2018). http://vulhub.org/.

Wan, C., Liu, S., Hoffmann, H., Maire, M., and Lu, S. (2021). Are machine learning cloud apis used correctly? In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 125–137. IEEE.

Wang, W. (2021). Mlee: Effective detection of memory leaks on early-exit paths in os kernels. In *2021 {USENIX} Annual Technical Conference ({USENIX}{ATC} 21)*, pages 31–45.

Wetter, J. and Ringland, N. (2021). Understanding the impact of apache log4j vulnerability. https://security.googleblog.com/2021/12/understanding-impact-of-apache-log4j.html.

Xu, Z., Chen, B., Chandramohan, M., Liu, Y., and Song, F. (2017). Spain: security patch analysis for binaries towards understanding the pain and pills. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 462–472. IEEE.

Yamaguchi, F., Wressnegger, C., Gascon, H., and Rieck, K. (2013). Chucky: Exposing missing checks in source code for vulnerability discovery. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 499–510.

Yen, T.-F., Oprea, A., Onarlioglu, K., Leetham, T., Robertson, W., Juels, A., and Kirda, E. (2013). Beehive: Large-scale log analysis for detecting suspicious activity in enterprise networks. In *Proceedings of the 29th Annual Computer Security Applications Conference*, pages 199–208. ACM.

Zheng, Y. and Zhang, X. (2013). Path sensitive static analysis of web applications for remote code execution vulnerability detection. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 652–661. IEEE.

Zhou, Y. and Jiang, X. (2012). Dissecting android malware: Characterization and evolution. In *2012 IEEE symposium on security and privacy*, pages 95–109. IEEE.