The Thesis Committee for Naveena Sankaranarayanan
Certifies that this is the approved version of the following thesis:

SELinC: Security Evaluation of Linux Containers

APPROVED BY

SUPERVISING COMMITTEE:

Mohit Tiwari, Supervisor

Sarfraz Khurshid

SELinC: Security Evaluation of Linux Containers

by

Naveena Sankaranarayanan

Thesis

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

May 2018

Abstract

SELinC: Security Evaluation of Linux Containers

Naveena Sankaranarayanan, M.S.E.
The University of Texas at Austin, 2018

Supervisor: Mohit Tiwari

Operating System containers are key to the success of modern datacenters. Containers provide an easy way to encapsulate an application with its operating environment and deliver it seamlessly from development to test to production. Containers ensure consistency in loading the application across a variety of environments including physical servers, virtual machines, private or public clouds. Despite these benefits, containers often invite security concerns. Since containers share the Linux kernel on the host machine, a bug or a vulnerability in the kernel can compromise all the containers on the host machine. This thesis presents a novel model-driven fuzzing technique that can automatically analyze the kernel sources of containers. The proposed technique exposed two unknown bugs in the kernel implementation.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Operating-system-level virtualization refers to an operating system feature in which the kernel allows the existence of multiple isolated user-space instances called **containers** [1]. Linux kernel features, namely "namespaces" and "cgroups" help in isolating the processes from each other. The namespaces provide a layer of isolation. Each aspect of a container runs in a separate namespace and its access is limited to that namespace. The control groups (cgroups) limit an application to a specific set of resources. They allow shared available hardware resources to containers and optionally enforce limits and constraints.

Unlike in hardware virtualized environments, where a hypervisor serves as a point of control, in containers any user or service with access to the kernel root account is able to view and access all the containers sharing the Linux kernel. Security of containers rely on proven approaches to harden the kernel. Our threat model includes malicious users or processes that run within containers and try to exploit vulnerabilities within the kernel (Figure 1.1). On successful exploitation, the attacker can get control of the host machine or launch DoS attacks.

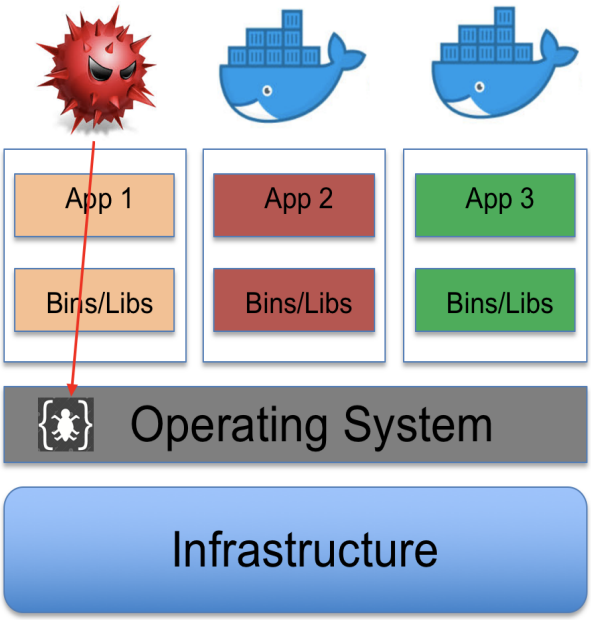

Figure 1.1: Threat Model : A malicious user on one container exploits a bug in the kernel and compromises all the containers on the machine

Fuzzing is a well-known technique for program testing by generating crafted malformed/semi-malformed random inputs to programs. Fuzzing Operating System interfaces or system calls is a hard problem because the kernel structures include numerous function pointers making it difficult to trace the control flow of the execution. A sample code snippet from kernel implementation of namespaces is depicted in Figure 1.2. The data structure *proc_ns_operations* has five function pointers pointing to different functions in *ipc* and uts namespace implementation.

```
/* ns_proc.h - Declaration of the struct proc_ns_operations */
struct proc_ns_operations {
        const char *name;
        const char *real_ns_name;
        int type;
        struct ns_common *(*get)(struct task_struct *task);
        void (*put)(struct ns_common *ns);
        int (*install)(struct nsproxy *nsproxy, struct ns_common *ns);
        struct user_namespace *(*owner)(struct ns_common *ns);
        struct ns_common *(*get_parent)(struct ns_common *ns);
} __randomize_layout;

/* ipc/namespace.c - Usage of the struct proc_ns_operations */
const struct proc_ns_operations ipcns_operations = {
        .name           = "ipc",
        .type           = CLONE_NEWIPC,
        .get            = ipcns_get,
        .put            = ipcns_put,
        .install        = ipcns_install,
        .owner          = ipcns_owner,
};

/* kernel/utsname.c - Usage of the struct proc_ns_operations */
const struct proc_ns_operations utsns_operations = {
        .name           = "uts",
        .type           = CLONE_NEWUTS,
        .get            = utsns_get,
        .put            = utsns_put,
        .install        = utsns_install,
        .owner          = utsns_owner,
};
```

Figure 1.2: Kernel code snippet depicting the struct members as function pointers

In this thesis, we propose a novel and automated technique to identify the system calls that access the namespaces and cgroups implementation in the kernel through a model generator and effectively fuzz the corresponding system calls through a state-of-the-art Linux system call fuzzer, **Syzkaller**[2]. We built a validator module that checks for any access control violations after the fuzzer executes every system call on the target machine. Our solution can thus detect vulnerabilities causing Denial of Service(DoS) and Access Control Violations(ACV). The quality of fuzzing quantized by the coverage metric and the new exploits found are presented.

In summary, we make the following contributions:

- **Model-driven fuzzing.** We design a novel technique to facilitate the fuzzing of system calls that can exploit the vulnerabilities in namespaces and cgroups.
- **Access Control Validation.** Traditionally, fuzzers are capable of monitoring the target system for crashes, failing built-in code assertions or memory leaks. To the best of our knowledge, our system is the first practical attempt to integrate access control monitoring capability to kernel fuzzers.
- We evaluate our technique on five Linux kernel versions, and found two new and several reported kernel vulnerabilities consisting of crash, service unavailabilty and access control violation bugs.

This thesis is structured as follows. Chapter 2 provides a brief background on containers, security implications of containers and fuzzing in addition to a motivating example. Chapter 3 presents the design overview of the proposed technique. Chapter 4 presents our evaluation and results. Chapter 5 discusses the related work based on fuzzing and other techniques. Chapter 6 concludes the thesis with the limitations of the current design and potential improvements.

Chapter 2

Background

2.1 Containers

Containerization is an OS-level virtualization technique for deploying and running distributed applications without launching an entire Virtual Machine (VM) for each application. Alternative to Virtual Machines, multiple isolated systems called containers are run on a single host and access a single kernel. Since containers share the same OS kernel, containers are faster and consume fewer resources than VMs, which require separate OS instances.

Containers can run seamlessly on bare-metal systems, cloud instances and virtual machines, across Linux and select Windows and Mac OS. Containers hold all the necessary components to run an application, such as application code base, language runtime, libraries and environment variables. The complete bundle of information to execute in a container is called the image. The container engine running at the user space deploys the image on to the hosts. To update an application, a developer just has to make changes to the code in the container image, then redeploy that image to run on the host OS. The most popular container engines are Docker [3], LXC [4] and rkt [5] (CoreOs).

2.1.1 Containers Vs Virtual Machines

Virtual machines (VMs) are an abstraction of hardware virtualization [6]. The hypervisor resides between the hardware and the operating system. The hypervisor allows multiple VMs to run on a single machine. Each VM includes a full copy of an operating system. Although this allows different VMs on the same host to use different OS versions, resource consumption is more and the boot up is slow.

Containers are an abstraction at the Operating System layer that packages code and dependencies together. Several containers can run on the same machine (VMs or bare metal) and share the OS kernel with other containers, each running as isolated processes in the user space. Containers require less space than VMs (container images are typically tens of MBs in size), and boot almost instantly. Containers provide isolation similar to virtual machines, but they virtualize the operating system instead of the hardware. Figure 2.1 depicts the organizational difference between VMs and containers.

2.2 Security concerns in containers

A potential shortcoming of containerization is lack of isolation from the host OS. Since containers share a host OS, security exploits have easier access to the entire system when compared to hardware virtualization.

Thus, the Operating System that manages the container environment is the vital layer of the stack to secure, because an exploit that compromises

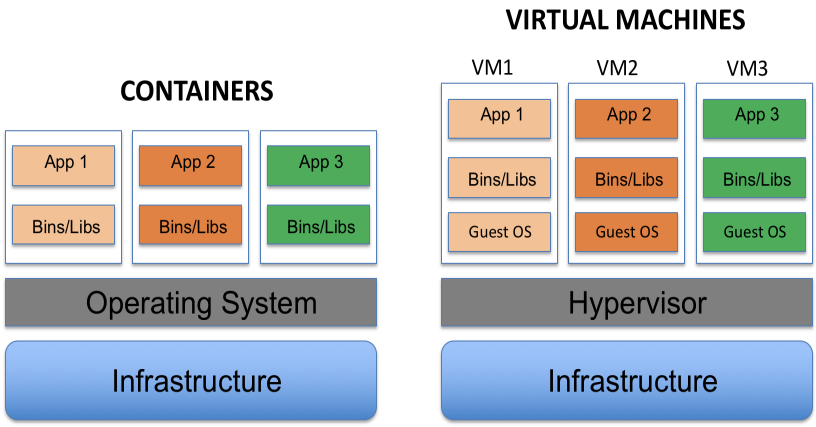

Figure 2.1: Containers Versus Virtual Machines

the host environment can potentially provide exploiters access to everything else in the stack.

Unfortunately, ensuring the security of the host OS is not simple. In this thesis, we propose a novel and automated technique to discover and minimize bugs in the OS kernel implementation that have lead to security exploits in containers in the past.

2.3 Kernel features enabling containerization

Namespaces and cgroups are the kernel features that help in isolating the processes from each other.

2.3.1 Namespaces

Namespaces are a feature of the Linux kernel that partitions kernel resources such that each set of resources are accessible only by a specific set of processes [7]. Namespaces enable a set of processes to have different views of the system than the other set of processes. As of kernel version 4.10, there are 7 kinds of namespaces namely,

- mnt (mount points, filesystems)
- pid (processes)
- net (network stack)
- ipc (System V IPC)
- uts (hostname)
- user (UIDs)
- control group (cgroup)

Namespaces isolation is necessary in the world of containers. Without namespaces, a process running in container A could interfere with the resources used by another process running in container B. By namespacing these resources, the process in container A cannot access the resources outside its namespace.

Three system calls are used for creating/joining namespaces:

- ***clone()*** - clone system call creates a new process and a new namespace. The new process is attached to the new namespace.
- ***unshare()*** - unshare system call does not create a new process. It creates a new namespace and attaches the current process to it.
- ***setns()*** - setns system call associates a process with an existing namespace.

2.3.2 Cgroups

Cgroups is a Linux kernel feature that helps in the allocation of resources such as CPU time, system memory, network bandwidth etc. among user-defined groups of processes [8]. A cgroup is a collection of processes that are bound by the same criteria and associated with a set of parameters or limits. A cgroup can be hierarchical, inheriting limits/parameters from its parent group. The cgroup interface provides access to multiple resource subsystems. Some of the available subsystems are discussed below:

- blkio : blkio subsystem sets limits on input/output access to and from block devices such as physical drives (disk, solid state, or USB).
- cpu : cpu subsystem uses the scheduler to provide cgroup tasks access to the CPU.
- cpuacct : cpuacct subsystem generates automatic reports on CPU resources used by tasks in a cgroup.

- cpuset : cpuset subsystem assigns individual CPUs and memory nodes to tasks in a cgroup.
- memory : memory subsystem sets limits on memory use by tasks in a cgroup and generates automatic reports on memory resources used by those tasks.
- devices : devices subsystem allows or denies access to devices by tasks in a cgroup.
- perf_event : perf_event subsystem identifies cgroup membership of tasks and can be used for performance analysis.
- net_cls : net_cls subsystem tags network packets with a class identifier (classid) that allows the Linux traffic controller (tc) to identify packets originating from a particular cgroup task.

2.4 Fuzzing

Fuzzing (fuzz testing) is an automated testing methodology that feeds crafted inputs to the programs under test. The main goal of fuzzing is to generate invalid, unexpected, or random inputs to a target program, observe the execution of the target program, and report a vulnerability whenever an abnormal event (crash, exception) is captured. Fuzz testing was originally developed by Barton Miller at the University of Wisconsin in 1989. Fuzzing is simple yet can often reveal serious defects that are overlooked when a program

is written and debugged. It can be a black box technique, requiring no access to source code or can be white/grey box technique when augmented with code analysis tools.

2.4.1 Syzkaller

Operating system kernel is usually tested by fuzzing operating system interfaces or system calls. Syzkaller is a state-of-the-art fuzzer built particularly for fuzzing Linux system calls. Syzkaller has reported around 350 new bugs in the mainstream kernel. Syzkaller performs fuzzing by adding, removing or changing a system call and its parameters in a sequence of system calls by obtaining feedback from code coverage information. Syzkaller monitors whether the kernel crashes or hangs and saves the information about the crash/hang for bug localization and reproduction. A high-level operational view of syzkaller is depicted in Figure 2.2[9].

2.5 Motivation

Container adoption surveys indicate that amount of money spent by companies on license and usage fees for container technologies increases by 5% every year [10] . This increasing investment is an indication that containerization is becoming more popular every year. Docker is the most talked-about infrastructure technology of the past few years. It is found that Docker adoption in the industry has increased to 18.8% in 2017, with about 40% increase from (13.6%)

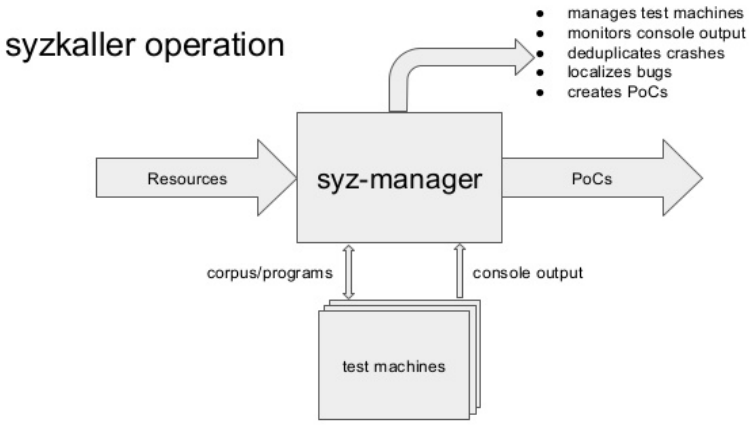

Figure 2.2: A high-level Operational view of syzkaller

2016 [11]. Several companies have debated on the security concerns for production container deployments. Container engines and orchestration tools depend on the integrity of the shared host OS kernel for the integrity and isolation of the containers that run on top of it. Hence a hardened OS is vital for the container based infrastructure. Linux kernel [12] which has around 13,500 developers and a rapid development rate of 7.8 patches per hour (average), is highly prone to implementation bugs that can lead to vulnerabilities. According to the National Vulnerability Database, the number of kernel vulnerabilities reported in 2017 is around 627, with an increase of 33% from the number of vulnerabilities reported in 2016 [13]. Thus an automated way of detecting the vulnerable bugs is critical for the secure container systems.

Several bugs in the implementation of namespaces and cgroups have been reported in different kernel versions in the past [14]. These bugs, when

exploited, can cause serious security issues for containers. For instance, the *do_change_type* function in fs/namespace.c in Linux kernels before 2.6.22 failed to check if the caller has the *CAP_SYS_ADMIN* capability, which allows local users to gain privileges or cause a denial of service by modifying the properties of a mountpoint. This bug was reported as CVE-2008-2931 (CVE - Common Vulnerabilities and Exposures) with a high CVSS score depicting its impact [15]. Bugs as common as missing checks in some function in the kernel code can thus open the door for an exploit that can affect all the containers running on the host machine.

This bug shows the need for validation of the kernel implementation. With the difficulty of tracing the function pointers in the kernel(Figure 1.2), verification becomes difficult. Hence we need an efficient method to generate system calls to direct the fuzzer to explore bugs in the namespaces and cgroup implementation.

Chapter 3

Design Overview

Figure 3.1 represents the architecture of the proposed system to automatically detect bugs pertaining to namespaces and cgroups implementation in the Linux kernel code. In this chapter, we will provide an overview of each component of the system and we will present examples to illustrate their significance.

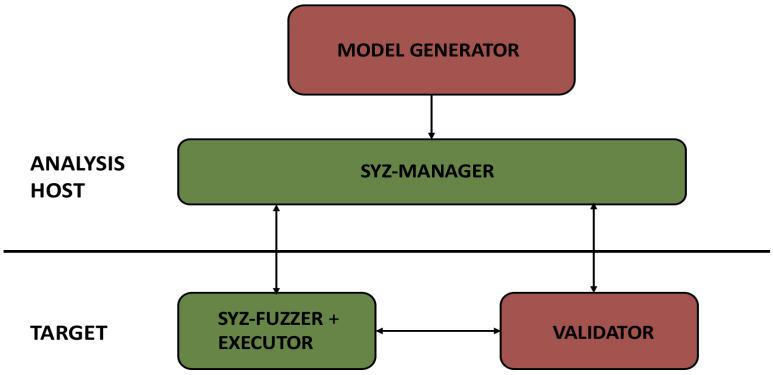

Figure 3.1: Model-driven Fuzzing Architecture

3.1 Model Generator

The model generator generates a set of system calls with the aim to better explore the bugs in the namespaces and cgroup implementation of the kernel. We propose two model generators based on Markov model and DIFUZE.

3.1.1 Markov Model Generator

Markov model generator is built on the observations of previously found bugs in the implementation of namespaces and cgroups. We build the Markov chain from system call traces observed in the programs that exploited vulnerabilities in namespaces and cgroups in the past. Each system call in the trace is a state in the Markov chain and the transition probabilities relationship between states are derived from the sequence of the system call traces in the input data set. The Markov model generator provides an optimal sequence of system calls which when fed into the syzkaller have higher probability of exploiting bugs similar to the training set.

We obtained the Jaccard similarity coefficient on the system call trace for all the previously reported bugs and found that the coefficient was consistently close to zero. This means that the similarity between the traces is low and hence we need an effective model generator to capture different paths of system calls.

For a set of reported bugs, we were able to implement C program that exploits the bug and generate the system call traces. With these system call

traces, we build a Markov model that generates system call patterns to feed into syzkaller. Our Markov model implementation considers only system calls irrespective of the input arguments. We collected system call traces of the test frameworks of LXC and Docker container engines and assign more weights to the system calls that are not tracked by these frameworks. We do this to ensure that the system calls not present in the traces of userspace test frameworks are well covered in our test inputs. System calls with more weights are given priority during the sequence generation. Figure 3.2 highlights a single sequence generated with the Markov model we built.

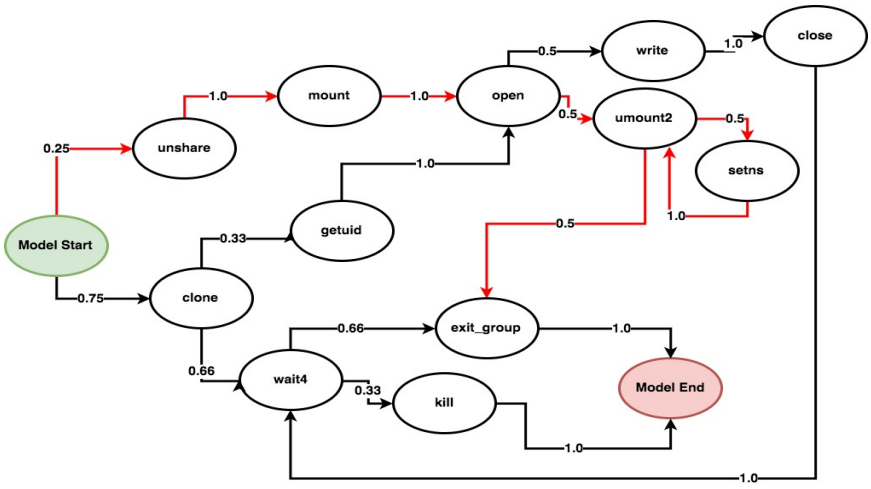

Figure 3.2: Markov model depicting a single sequence generation

3.1.2 DIFUZE-based Generator

DIFUZE is an interface-aware fuzzing framework for kernel drivers [16]. DIFUZE takes the source code of the kernel as input and walks through a series of steps to recover the interface for device drivers and generates the correct structures to fuzz the interface. In the interface recovery step, DIFUZE analyzes the provided kernel source using combination of analyses implemented in LLVM. We leverage these LLVM analyses from DIFUZE to identify the system call name, system call definition and internal structures that could be possibly accessed by each system call. With static instrumentation and LLVM analyses, a group of system calls that access the same structures in the kernel code can be obtained and grouped into a fuzz group. Our DIFUZE-based model generates a fuzz group that access the most common data structures in the kernel implementation of cgroups and namespaces, which is then fed to the syz-manager through the syzkaller config file. The syz-manager forwards the syz-manager commands to the syz-fuzzer.

3.1.2.1 Version Control-based Fuzz group Filter

The number of system calls generated from the DIFUZE-based model is usually high. This is because the generated fuzz group includes all the system calls that can access the kernel structure of our interest. In order to effectively reduce the size of the fuzz group we employ a version control-based filtering technique. The filtering technique helps to select system calls that are most recently and frequently updated in the kernel code. We believe that

recently and frequently updated functions have high importance in terms of functionality and are prone to implementation bugs.

In order to implement this filtering technique, we use "GNU cflow" [17] tool to obtain the function call graph of all the system calls present in the fuzz group generated by DIFUZE-based generator. GNU cflow is a static analysis tool that reports hierarchical function flow graph of the source analyzed. Since cflow cannot track function pointers inside a function, we limit the function call graph of the system calls as reported by cflow in our filtering technique. We then use git log [18] to obtain the list of functions updated in the recent past (last 30K commits). For each of the system call, we find its frequency count which is the number of references of the system call or functions in its call graph (obtained from cflow earlier) in the git log history obtained. We then select the system calls with large frequency counts.

Figure 3.3 illustrates the steps involved in generating a fuzz group from the kernel source and then filtering them based on the version control.

The two system call generators mentioned above are complementary to each other in the sense that the Markov model generator tries to capture system call sequences from the exploits observed in the past whereas the DIFUZE-based generator captures all the system calls that can be involved in the functionality of namespaces and cgroups. Hence, the latter covers system calls that can potentially cause exploits but are not observed in the past.

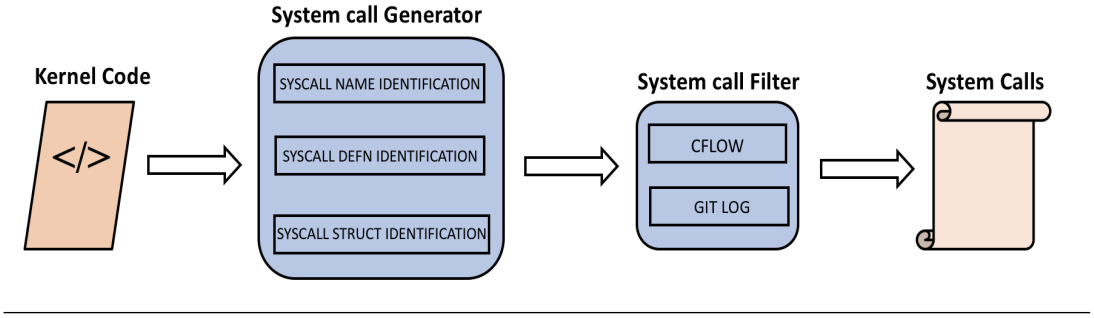

Figure 3.3: DIFUZE-based fuzz group generator

Our model generator ensures calling context of system calls are valid. For example, a read system call would never occur without a prior call to open system call and a valid file descriptor passed from open to read call. Our system captures all these dependencies and when the model generates a set of system calls with a read call, it automatically adds the dependent system call (open in this case) to the generated sequence if it is not present.

3.1.3 Previous Vulnerabilities

Here we discuss some of the namespaces related vulnerabilities observed in various Linux kernel versions and the implementation bugs that lead to them. In interest of space, we discuss only a subset of the existing vulnerabilities.

We implemented C programs to recreate all the below mentioned vulnerabilities.

CVE-2008-2931 occurred in the Linux kernels before 2.6.22. It did not verify that the caller has the *CAP_SYS_ADMIN* capability. This allowed local

users to gain privileges or cause a denial of service by modifying the properties of a mountpoint.

CVE-2009-1338 was observed in Linux kernels before 2.6.28. It did not consider PID namespaces when processing signals directed to PID -1. This allowed local users to bypass the intended namespace isolation and send arbitrary signals to all processes in all namespaces via a kill command.

CVE-2009-3621 was observed in the Linux kernel 2.6.31.4 and earlier. It allowed local users to cause a denial of service (system hang) by creating an abstract-namespace *AF_UNIX* listening socket, performing a shutdown operation on this socket, and then performing a series of connect operations to this socket.

CVE-2013-1956 was observed in Linux kernels before 3.8.6. It did not check whether a chroot directory existed that differs from the namespace root directory. This allowed local users to bypass intended file system restrictions via a clone system call.

CVE-2013-1957 was observed in the *clone_mnt* function in fs/namespace.c in the Linux kernel 3.8.6 and earlier. It did not properly restrict changes to the *MNT_READONLY* flag, which allows local users to bypass an intended read-only property of a filesystem by leveraging a separate mount namespace.

CVE-2013-1959 was observed in Linux kernels before 3.8.9. It did not have appropriate capability requirements for the *uid_map* and *gid_map* files. This allowed local users to gain privileges by opening a file within an

unprivileged process and then modify the file within a privileged process.

CVE-2014-5207 was observed in fs/namespace.c in the Linux kernel 3.16.1. It does not properly restrict clearing *MNT_NODEV*, *MNT_NOSUID*, and *MNT_NOEXEC* and changing *MNT_ATIME_MASK* during a remount of a bind mount, which allows local users to gain privileges, interfere with backups and auditing on systems that had atime enabled, or cause a denial of service (excessive filesystem updating) on systems that had atime disabled via a "mount -o remount" command within a user namespace.

CVE-2016-1576 was observed in the *overlayfs* implementation in the Linux kernel through 4.5.2. It did not properly restrict the mount namespace, which allows local users to gain privileges by mounting an overlayfs filesystem on top of a *FUSE* filesystem, and then executing a crafted setuid program.

CVE-2016-6213 was observed in fs/namespace.c in the Linux kernel before 4.9. It did not restrict how many mounts may exist in a mount namespace, which allows local users to cause a denial of service (memory consumption and deadlock) via *MS_BIND* mount system calls, as demonstrated by a loop that triggers exponential growth in the number of mounts.

CVE-2017-5967 was observed in the Linux kernels through 4.9.9. It allowed local users to discover real PID values (as distinguished from PID values inside a PID namespace) when *CONFIG_TIMER_STATS* is enabled by reading the /proc/timer_list file, related to the *print_timer* function in kernel/time/timer_list.c and the *__timer_stats_timer_set_start_info* function in

kernel/time/timer.c

3.2 Syz-Manager

The syz-manager process starts and monitors the target machine. It is also responsible for starting a syz-fuzzer process inside the target machine. It runs on the analysis host and takes care of storing the input programs and crash information.

3.3 Syz-Fuzzer and Executor

The syz-fuzzer guides the fuzzing process by generating different combinations of the input system call group components and executes them with the help of the syz-executor. It also sends back the coverage and results of the executed program to the syz-manager. The syz-fuzzer starts syz-executor processes within the target machine.

Each syz-executor process executes a single sequence of system calls. It gets the program to execute from the syz-fuzzer process and sends the results back. It is designed to have minimal interference with the fuzzing process. Before and after the execution of each of the system call in the input sequence, the syz-executor triggers the validator module to perform Access Control Violation (ACV) checks. The result of the validation process is returned to the syz-executor, which then continues with the next system call execution based on the results.

3.4 Validator

The validator module monitors the state of the system altered by each system call in the input program executed by the syz-executor. Before the execution of the system call, the validator checks if the system call can be executed without violating any access controls during its execution. If it predicts a possibility of violation and is not detected by the kernel (absence of a valid error code) [19] upon execution, then the validator module reports an ACV to the syz-fuzzer which can be asserted through an analyst.

The validator module is capable of detecting access control violations related to all the system calls that are present in the input data set of the Markov model generator. Nearly 70 system calls related to File system (23), Network (9), Processes(27), Namespaces(3), Signals(4) are supported by our validator module.

The validator keeps track of the process and file information with their *uid, gid*, capabilities in the root namespace and the current namespace they belong to. The syz-fuzzer process registers its PID (process ID) with the validator and it becomes the root of the process tree tracked by the validator. The validator then tracks all the syz-executor processes created by the syz-fuzzer. The validator runs with root permissions in the root namespace and is capable of collecting information about all the syz-executor processes. The validator perfoms checks before and after the execution of each system call. Before the execution of each system call, the validator determines if the call is allowed to perform the action intended. It also checks the state of the

resources/system altered after the execution of the system call and if the state is altered when it was not allowed to (determined by the validator before the execution of the system call), the validator module reports an ACV to the syz-fuzzer. Figure 3.4 depicts the series of communication involved between the syz-fuzzer, syz-executor and the validator.

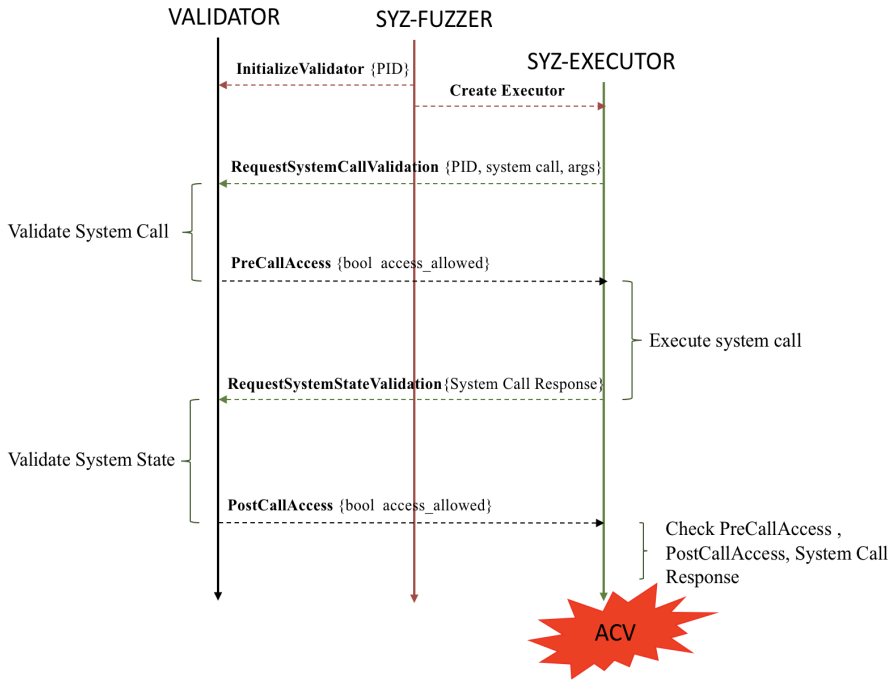

Figure 3.4: Communication between Validator and Syzkaller components

The syz-fuzzer registers it *pid* (Process Id) with the validator through the *InitializeValidator* command. This *pid* then becomes the root node of the process tree tracked by the validator. The syz-fuzzer then creates a

syz-executor process to fuzz the target VM. Before the execution of each of the system call, the syz-executor passes its *pid, system call name and arguments* to the validator through *RequestSystemCallValidation* command. In case of file-related system calls in the Linux kernel (open, openat, close, read, write, chmod, fchmod, unlink, and stat, lstat, fstat, lseek, dup), the validator checks the compatibility of flags and file access modes for each of the call. When the flags and access modes requested do not match, an access denied flag is returned by the validator through the *PreCallAccess* command. The syz-executor then executes the system call and stores the return code of the execution. It now requests the validator to perform state validation check through the *RequestSystemStateValidation* command. The validator checks if the state of the file is altered and returns a flag to the syz-executor. In the absence of a valid error code returned from the execution when the validator predicted access denial before the actual execution, an **ACV VIOLATION** is reported.

In order to illustrate the kind of checks the validator performs we consider the following example. In case of an open system call requested in read-only mode, the validator checks if the process is inside a user namespace and finds the *uid* (User Id) mapping of the process in the root namespace and confirms if the user has search permissions to the file. It also retrieves the *uid* (*uid* mapped in root namespace) of the user who owns the file being read. The validator then checks if the process *uid* matches the file owner *uid* or if the file owner *uid* falls within the mapped *uid* range of the process. Similar checks

are done for the process *gid* (Group Id) against the file *gid*. If the process *uid, gid* does not match the file owner *uid* and *gid*, the validator verifies if the file has read permissions for other users as indicated by the *S_IROTH flag*. An ACV flag is raised in the absence of the proper read permissions indicated by the mode bits of the file and corresponding error code when the open system call attempts to open the file in read mode. Similar checks are also performed even if the process *uid, gid* and file owner *uid, gid* match.

To depict the kind of vulnerabilities our validator module can detect, we consider the following example. The capabilities implementation in the Linux kernels before 3.14.8 does not properly consider that namespaces are inapplicable to inodes, which allows local users to bypass intended chmod restrictions by first creating a user namespace, and setting the *setgid* bit on a file with group ownership of root. This access control violation vulnerability was reported in 2014 (CVE-2014-4014). According to the vulnerability, if a non-privileged user (outside the namespace) owns a file with *gid* set to 0(root), the user can set the *setgid* bit on that file inside the user namespace (since user is root inside the namespace) due to the missing check whether the caller is allowed to perform the chmod operation based on the mapping to *uid* and *gid* outside the namespace. Since our validator module checks *uid, gid* mapping inside and outside namespaces, it was able to detect this vulnerability. It is worth mentioning that traditional fuzzers looking for crashes and exceptions will not be able to detect bugs that involve penetrating access controls. Our technique can successfully detect these class of vulnerabilities also.

The model generator, fuzzer and the validator module together can possibly detect DoS attacks and Access Control Violation attacks that can be exploited due to the bugs in the implementation of namespaces and cgroups in the Linux kernel.

Chapter 4

Evaluation and Results

In this Chapter we specify the implementation details of the proposed design, evaluate its efficiency and present the results obtained.

4.1 Implementation

We engineered our system to be completely automated. The user has to only provide the kernel version to be installed on the target machine, the model generator feeds the system calls generated to the syz-manager on the analysis host, which then starts the execution of the fuzzer on the target host.

We implemented the Markov model generator, Version control based filter in Python. DIFUZE-based model uses LLVM 3.8 to implement the system call name, definition extraction and structure identification. Syzkaller is a Linux system call fuzzer, that is integrated with the model generator to fuzz the generated system calls. The model generator automatically converts the generated sequence to syzkaller acceptable format. The validator module which interacts with syz-executor component of the syzkaller is implemented in Golang as a library and is dynamically loaded in the syz-executor. The end-to-end automation is implemented in Python.

Kernel Version	Linux Distro
v3.13	Ubuntu 14.04LTS
v4.0	Fedora 22
v4.1	CoreOS 766.0.0
v4.4	openSUSE Leap 42.3
v4.17	Latest kernel

Table 4.1: Linux kernel versions used in the evaluation

4.2 Evaluation

We determine the effectiveness of the proposed solution by evaluating its coverage metrics and bug-finding capabilities. The evaluation is performed on five different Linux kernel versions which are present in popular Linux distributions (specified in Table 4.1). We evaluate the effectiveness of core component of our system, the model generator. We then perform evaluation of the bug finding capabilities of system.

4.2.1 Model Generator Evaluation

We proposed two model generators complementary to each other such that the Markov model generator aims to capture a sequence of system calls that are similar to the ones observed from its training set whereas the DIFUZE based model generator aims to generate a possible set of system calls that access namespaces created at runtime. Both models together ensure completeness of fuzzing system calls that have caused exploits in the past and the system calls that can potentially cause exploits but not observed in the past. We

use code coverage as a metric to evaluate the DIFUZE based fuzzing where exploring new code paths increases the possibility of hitting a bug. For the Markov model generator, we use node and edge coverage metrics to identify if the generated sequences have exercised all nodes and edges possible from the training set.

Table 4.2 shows some of the important kernel structures involved in the implementation of namespaces and cgroups and the number of system calls accessing them as identified by LLVM analyses, the number of system calls finally added to the fuzz group after the version control based filtering is performed and number of bugs found and the code coverage metric as reported by syzkaller.

Table 4.3 shows the number of system calls in each of the sequences generated from the Markov model, number of nodes and edges covered in the overall Markov chain by each sequence and number of bugs found. In interest of space, we have identified each sequence with a sequence number instead of the actual sequence. With ten sequences, we were able to cumulatively achieve 100% node coverage and around 70% edge coverage. We continued the fuzzing process to achieve 100% edge coverage.

In interest of space, we have depicted only a subset of experiments performed in Table 4.2 and Table 4.3.

Kernel Structure	System calls Reported by DIFUZE	System calls Filtered by Git log	Code Coverage	Bugs Found
nsproxy	146	63	7100	1
ns_common	15	10	6830	0
pid_namespace	101	41	7388	0
user_namespace	92	43	5465	0
uts_namespace	4	4	5608	0
fs_struct	115	55	6489	0
files_struct	115	55	7895	1
cred	146	62	6264	0
cgroup_subsys	62	30	624	0
completion	146	63	7072	1
css_set	115	55	6751	1

Table 4.2: Kernel structures and system calls as recovered by DIFUZE-based Model Generator

4.3 Evaluation Setup

To evaluate the efficiency of our model generator we run the following configurations of syzkaller :

- **Syzkaller-mg**: Syzkaller only fuzzes the system calls provided by the model generator.
- **Syzkaller-base**: Syzkaller can fuzz all the system calls available on the kernel.

In order to support the code coverage collection in syzkaller, we enabled the *CONFIG_KCOV* while building the kernel. Since *KCOV* utility was

Sequence Number	Sequence Length	Nodes Covered (Total: 72)	Edges Covered (Total: 1038)	Bugs Found
1	80	34	456	0
2	15	13	151	0
3	13	9	168	0
4	13	8	64	0
5	101	33	436	1
6	20	13	118	0
7	32	14	310	0
8	32	20	206	0
9	14	7	16	1
10	9	8	55	0

Table 4.3: Coverage Stats of system call sequences from Markov-model generator

introduced in the kernel version 4.6, we ported the KCOV utility to the older kernel versions [20]. We also enabled *KASAN* [21], a dynamic memory error detector in the kernel. It helps syzkaller to detect use-after-free and out-of-bounds bugs. Syzkaller does not depict the percentage of code coverage for every file because it might be misleading as there are varying amounts of code that can't be covered (init, interrupts, workqueue/rcu/timer callbacks, etc).

In order to effectively compute the fuzzing duration based on coverage, we performed code coverage check at the end of every hour and continued fuzzing as long as the code coverage increased by atleast 1%. We stop fuzzing when the coverage almost saturates. We performed coverage-driven fuzzing

for inputs driven by DIFUZE-based generator. We do not have code coverage feedback enabled for fuzzing the system calls fed by the Markov model generator because we try to generate sequences similar to system call sequences that have triggered a bug in the past. When syzkaller is enabled with code coverage, it does not generate fuzzing inputs that match or closely match the input sequence of the observed bug as the inputs are driven with the sole aim of code coverage. For inputs driven by Markov model, we performed fuzzing for 2 million syzkaller programs or until a vulnerability is successfully triggered.

4.4 Results

Our tool was able to find 4 unique bugs in the five kernel versions that were used for fuzzing. Table 4.4 shows the vulnerabilities found, number of system calls enabled for fuzzing and the the average time to trigger the vulnerabilities by two configurations of syzkaller - **Syzkaller-mg, Syzkaller-base** used in our evaluation. We observe that Syzkaller-mg is able to trigger the bugs specified faster than Syzkaller-base configuration. This is because Syzkaller-mg has a reduced input set of system calls that are more related to the kernel subsystem of interest, whereas in the base configuration syzkaller tries to fuzz all the system calls supported in the entire kernel. In our experiments, when syzkaller tries to fuzz all the system calls (Syzkaller-base), several crashes occurred with ioctl and other Device Management related system calls which do not belong to the subsystem of our interest. This increases the delay in fuzzing the system calls of interest.

Vulnerability	Syzkaller-mg		Syzkaller-base	
	System calls	Trigger Time	System calls	Trigger Time
Process hang	32	0.15 h	267	3.27 h
Null-pointer-dereference	37	7.01 h	267	9.16 h
Denial-of-Service	14	1.34 h	267	>12.00 h

Table 4.4: Number of System calls enabled and Average time to trigger the vulnerabilities

In the next few subsections, we will present case studies of the bugs found during our experiments.

4.4.1 Case Study I : Process hang

We generated different system call sequences from the Markov model generator and fed them into syzkaller to perform extensive fuzzing. With one of the inputs, syzkaller was able to discover an exploit that causes a hang on the target VM. The hang was observed when syzkaller generated a *socket()* system call with *AF_INET/PF_INET* as the domain argument with *SOCK_RAW* type argument and a random protocol number greater than 255 (*IPROTO_XXX*). The socket call gives back a file descriptor, which when tried to connect through *connect()* system call, the target VM hangs. Similar behavior is observed when the socket file descriptor is passed as an input argument to *write()* system call. This bug was observed in Linux kernel versions through 4.1. The bug is a missing check for valid protocol range allowed for socket

type *SOCK_RAW* in *inet_create()* function in net/ipv4/af_inet.c. The latest versions of the kernel include checks for wrong protocol number input argument in the *inet_create()* function as the *SOCK_RAW* type argument can only be provided with a protocol number between 0-255 and returns -1 (EINVAL) preventing the hang.

Listing 4.1 shows the sequence of system calls to trigger the vulnerability.

```
1 memset(r, -1, sizeof(r));
2 r[0] = syscall(__NR_socket, 0x2ul, 0x3ul, 0x80000000ul);
3 syscall(__NR_mmap, 0x20001000ul, 0x1000ul, 0x3ul,
4         0x32ul, 0xffffffffffffffful, 0x0ul);
5 *(uint16_t*)0x20001000 = (uint16_t)0x1;
6 memcpy((void*)0x20001002,
7        "\x2e\x2f\x66\x69\x6c\x65\x30\x00", 8);
8 syscall(__NR_connect, r[0], 0x20001000ul, 0xaul);
```

Listing 4.1: The sequence of system calls to trigger bug I

4.4.2 Case Study II : Null-pointer-dereference

We discovered an undisclosed vulnerability while fuzzing the Linux kernel version 4.0 with a set of system calls generated by DIFUZE-based generator for the kernel structure *nsproxy*. The crash was induced due to a null-pointer-dereference on a socket buffer (skb structure) by *sendto()* system call. The crash is not observed in the latest Linux kernel versions since the check for the existence of the socket buffer is added. The older versions are still susceptible to this bug. Listing 4.2 shows the sequence of system calls to trigger the vulnerability. This bug can be exploited to trigger a DoS attack.

We could not find any CVE report referring to this bug. So it is considered to be an undisclosed vulnerability.

Listing 4.2 shows the sequence of system calls to trigger the vulnerability.

```
1 memset(r, -1, sizeof(r));
2 syscall(__NR_mmap, 0x20000000ul, 0x7000ul, 0x3ul,
3         0x32ul, 0xffffffffffffffful, 0x0ul);
4 r[1] = syscall(__NR_socket, 0x2ul, 0x2ul, 0x1ul);
5 syscall(__NR_mmap, 0x20007000ul, 0x1000ul, 0x3ul,
6             0x32ul, 0xffffffffffffffful, 0x0ul);
7 syscall(__NR_sendto, r[1], 0x20005000ul, 0x8ul,
8         0xfffffffffffffffcul, 0x20007000ul, 0x10ul);
```

Listing 4.2: The sequence of system calls to trigger bug II

4.4.3 Case Study III : Denial of Service

This case study demonstrates our claim that the Markov model will be able to generate system calls that are slightly different to the observed ones and aids in exploiting similar bugs. With one of the input set of system calls from the Markov model, syzkaller was able to trigger an exploit which was reported previously (CVE-2015-4178). The sequence of calls that led to this exploit was similar to that of CVE-2014-9717 which is present in our training set. Syzkaller generated input crashed the target VM paving way for a DoS attack. This exploit was observed in Linux kernel version 3.13 and is due to a NULL Pointer Dereference on *fs_pin* struct members *m_list* and *s_list*.

Listing 4.3 shows the sequence of system calls to trigger the vulnerability.

```
memset(r, -1, sizeof(r));
syscall(__NR_unshare, 0x20000ul);
syscall(__NR_mmap, 0x20000000ul, 0x1000ul, 0x3ul,
        0x32ul, 0xffffffffffffffful, 0x0ul);
syscall(__NR_mount, 0x20841ffbul, 0x20000ffeul,
        0x2076cffbul, 0x4000ul, 0x20000000ul);
r[1] = syscall(__NR_open, 0x20000000ul, 0x0ul, 0x40ul);
syscall(__NR_mmap, 0x20000000ul, 0x1000ul,
        0x2000007ul, 0x20010ul, r[1], 0x0ul);
syscall(__NR_umount2, 0x20000000ul, 0x2ul);
syscall(__NR_setns, r[1], 0x20000ul);
syscall(__NR_unshare, 0x10020000ul);
syscall(__NR_unshare, 0x20000ul);
syscall(__NR_mmap, 0x200d6000ul, 0x3000ul, 0x3000000ul,
        0x10ul, r[1], 0x0ul);
syscall(__NR_umount2, 0x200d8000ul, 0x0ul);
```

Listing 4.3: The sequence of system calls to trigger bug III

4.4.4 Case Study IV : Access Control Violation

The validator module detected an improper return code error in Linux kernel version 3.13 when the *open* system call tried to open a specific read-only file with write-only flag argument. A file descriptor was returned when the file */proc/self/ns/mnt* which has *READONLY* permissions was opened with *O_WRONLY* flag by the *open* system call. The kernel versions from 4.0 return proper error code (errno:13, EACCESS) in this scenario. Since our validator performs checks before and after the execution of the system call, it was able to predict that the open system call with the above mentioned arguments should

be denied access. Since the actual execution of the system call did not deny access and returned a file descriptor instead, the validator was able to detect this violation. Since the file */proc/self/ns/mnt* provides a handle to the *mnt* namespace of the process, we believe that an improper access to the file might have security impact on the namespace. Listing 4.4 shows the sequence of system calls to trigger the vulnerability.

```
1 memset(r, -1, sizeof(r));
2 syscall(__NR_mmap, 0x20002000ul, 0x1000ul, 0x3ul,
3         0x32ul, 0xffffffffffffffffful, 0x0ul);
4 r[1] = syscall(__NR_open, "/proc/self/ns/mnt", 0x1ul,
5                 0x28ul);
6 syscall(__NR_mmap, 0x20003000ul, 0x1000ul, 0x3ul,
7         0x32ul, 0xffffffffffffffffful, 0x0ul);
```

Listing 4.4: The sequence of system calls to trigger bug IV

CVE	Kernel Version
CVE-2017-9242	v4.4
CVE-2017-14106	v4.4
CVE-2015-7872	v4.1
CVE-2014-4014	v3.13

Table 4.5: Linux kernel versions used in the evaluation

We performed our experiments in the *namespace* mode of syzkaller. In this mode a namespace sandbox is created and the fuzzing happens inside the created namespace. We believe this setting closely resembles containerized environments.

In addition to the above case studies, we encountered several crashes and hang of the target VM while fuzzing with several inputs from both of our model generators. The bugs pertaining to some of those crashes were either already reported and patched (listed in Table 4.5) or were not reproducible by the either syzkaller or an analyst. We ascertain these non-reproducible bugs to an internal kernel state or race condition reached during fuzzing that is not fully reproducible. For example the sequence of system calls in Listing 4.5 caused the target VM running the latest Linux kernel build to crash. But the crash was not reproducible.

```
memset(r, -1, sizeof(r));
syscall(__NR_mmap, 0x20000000ul, 0x90e000ul, 0x3ul,
        0x32ul,0xffffffffffffffful, 0x0ul);
r[1] = syscall(__NR_semget, 0x0ul, 0x4003ul, 0x0ul);
syscall(__NR_getrusage, 0x0ul, 0x20003000ul);
syscall(__NR_semop, r[1], 0x20003feeul, 0x3ul);
syscall(__NR_semop, r[1], 0x20000fe2ul, 0x0ul);
syscall(__NR_memfd_create, 0x20006000ul, 0x0ul);
syscall(__NR_shmget, 0x0ul, 0x3000ul, 0x78000000ul,
        0x20000000ul);
```

Listing 4.5: The sequence of system calls that triggered the non-reproducible crash

The coverage metrics obtained with our experiments indicate that we have judiciously covered the implementation of the system calls that can potentially be involved in namespaces and cgroups effectively.

Chapter 5

Related Work

This chapter discusses related work and is divided into two sections. First section deals with prior work on different fuzzing methodologies adapted for testing Linux kernels and the second section deals with other techniques used for kernel testing.

5.1 Fuzzing Techniques

Kernel system calls have been validated with different fuzzing techniques in the past [16, 22, 23, 24, 25, 26, 27, 28, 29]. We discuss recent works on different approaches used for enhancing the fuzzing process to intelligently drive inputs to the fuzzer.

5.1.1 DIFUZE

DIFUZE [16] is an automated, interface-aware fuzzing tool that analyzes the kernel source and extracts device driver interface information (ioctl commands and type of the arguments) to generate valid input arguments and trigger the execution of the kernel drivers. DIFUZE focuses on fuzzing ioctl interfaces on Android kernel. The following figure depicts the workflow of DIFUZE.

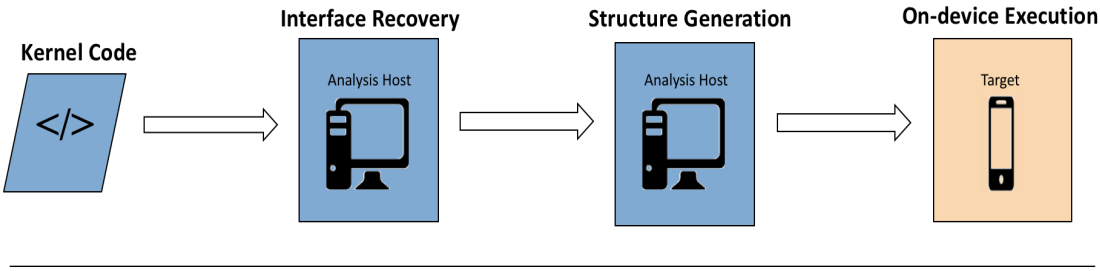

Figure 5.1: High-level workflow of DIFUZE from analysis to target execution

Our work leverages the LLVM analyses in DIFUZE as a tool to get information about the system calls accessing similar kernel structures. We utilize the LLVM analyses of DIFUZE to gain knowledge about all the system calls that accesses (reads/writes to) the structures used in the implementation of namespaces. With this information, we track the system calls that were not present in any of the previous vulnerabilities but are still related to namespaces. This ensures that our model generator does not miss any system call that can potentially cause vulnerabilities.

5.1.2 SemFuzz

SemFuzz [22] is an semantics-based automatic generator of proof-of-concept exploits on different Linux kernel vulnerabilities. The authors use the patch and discussion about a CVE (Common Vulnerabilities and Exposures catalog) to automatically generate proof-of-concept kernel exploits targeted at the individual CVEs. This tool uses Natural Language Processing to analyze CVEs and

gathers information regarding affected Linux kernel versions, functions and type of the vulnerability. It also analyzes the patch and its description in Linux git logs. With the information collected, SemFuzz forms a sequence of calls and then iteratively mutates the parameters of every call to eventually reach the vulnerability. Semfuzz was able to trigger around 18 reported vulnerabilities and two new bugs. This shows that the kernel is susceptible to new exploits that are very similar to previously reported exploits. Our work differs in that rather than intelligently fuzzing call sequences to trigger the reported exploits, our process uses previous knowledge of several exploits to generate new call sequences that can guide the fuzzing process to better explore the code region of interest.

5.1.3 Inferred Model-based Fuzzer

Inferred Model-based Fuzzer (IMF) [23] learns value and ordering dependency between kernel functions observed in different executions of an input program and guides the fuzzer with dependency information collected. IMF was evaluated on the latest macOS. IMF uses sequence replication and parameter mutation strategies in fuzzing, whereas we use both sequence mutation and parameter mutation while ensuring proper calling context for the system calls. For example, a call to write system call will never happen without a prior call to open system call and a valid file descriptor is passed from open to write call. IMF only identifies exploits based on a hang or a crash, but our tool can detect potential exploits based on access control violations too.

5.1.4 Directed Greybox Fuzzing

Directed Greybox Fuzzing (DGF) [24] guides the fuzzing process by calculating inter-procedural distances to the target program location based on the call graph and control-flow graphs gathered from program analysis done at compile-time. Using this distance metric, DGF identifies seeds that are closer to the target locations and prioritizes them during the fuzzing process. We on the other hand use compile time model generators to drive reduced set of inputs to the fuzzer so that the search space of the fuzzer is reduced at runtime. DGF was evaluated on security-critical userspace libraries and may be hard to extend it to Linux kernel code due to the presence of numerous input-dependent function pointers making it hard to generate the control-flow graphs at compile time.

5.1.5 perf_fuzzer

perf_fuzzer [25] is a fuzzing tool built specifically to test perf_event_open() system call. It is built on the Trinity system call fuzzer and employs domain-specific knowledge about the system call to generate input sequences accordingly. perf_fuzzer was able to find several new bugs in the kernel implementation of perf_event_open() system call showing that targeted domain knowledge can find bugs that more generic fuzzers would miss. System-call specific fuzzers are gaining traction recently. Instead of targeting one specific system call we target specific feature of the kernel and fuzz all the system calls related to it.

5.2 Other Techniques

In addition to fuzzing, few other methodologies were proposed to test the Linux kernel [30, 31, 32, 33, 34, 35]. Combinatorial testing is a software testing methodology that provides better coverage metrics compared to fuzzing. **Eris** [30] applies combinatorial testing to the system call interfaces. It is built on the Trinity system call fuzzer. ERIS test framework performs input parameter modeling for the system call APIs and feeds them into ACTS testing tool for producing test cases. ERIS currently uses crash as the test oracle. ERIS also identified the importance of testing the namespaces subsystem, but their solution did not include modeling of namespace-hierarchies.

PR-Miner (Programming Rule Miner) [31] uses a data mining technique to infer programming rules from large software code bases (eg. Linux kernel) to automatically detect violations to the inferred programming rules, which are strong indications of bugs. Each extracted rule can contain information regarding functions, variables and data types in the code. PR-Miner does not require any annotation from programmers. It was able to find uncovered bugs in the Linux kernel.

Chapter 6

Conclusions and future work

This Chapter concludes the thesis with the contributions made, limitations of the proposed design, failed ideas and possible future improvements.

6.1 Conclusion

In this thesis, we proposed a model-driven fuzzing technique to increase the effectiveness of automated security evaluation of namespaces and cgroups implementation in the Linux kernel. We proposed two model generators, a Markov-model based generator to trigger bugs similar to previously observed ones and a DIFUZE-based generator to explore system calls that can potentially cause exploits but are not observed in the past. The model generator drives syzkaller - a state of the art Linux system call fuzzer. We implemented a validator module that monitors the state of the system altered by each system call executed by the fuzzer on the target machine. The system built is completely automated and is effective in finding bugs leading to system crashes and access control violations.

We show that our technique is effective in generating system call traces similar to the observed ones through node and edge coverage metrics. We

carry out a meticulous evaluation, on five kernel versions, to demonstrate that our implementation of model-based fuzzer is effective, finding two previously unknown vulnerabilities.

6.2 Identified Limitations

Our Markov-model based generator, at each step makes the prediction based on the current state and not on the events that occurred before it. It is desirable to have a model generator that can capture longer histories and make decisions accordingly. To this reason, we built a sequence generator using a Recurrent neural network (RNN) [36]. The RNN can exploit the sequential ordering of the input training sequences and generate sequences similar to the training set. In our use case, RNN predicts the probability of a system call given the preceding calls and generates new sequences. It is a generative model. Given an existing sequence of system calls we sample a next system call from the predicted probabilities, and repeat the process until we have a full sequence of arbitrary length. Since we do not have large training data sets for the network to learn, our RNN based model generator was not able to capture the sequences effectively.

6.3 Future Improvements

As mentioned in the previous section, in order to feed the RNN based model generator with a large training set, an automated way of generating the system call traces for a large number of observed CVEs need to be developed.

We suggest developing an utility similar to the the "system calls retrieval" component of SemFuzz [22] in order to get the system calls associated with a CVE by analyzing the CVE and git log information through Natural-language processing (NLP) [37]. With a large input dataset, we believe RNN based generator can better predict sequences.

The validator module in the proposed design is currently capable of detecting Access Control Violations. The capability of the validator module can be extended to detect other common types of vulnerabilities like Memory corruption, deadlock and uncontrolled resource consumption.

We also plan to augment efficient bug minimization capability to our technique using type/semantic based analysis of the malicious program.

Bibliography

[1] "OS virtualization," https://en.wikipedia.org/wiki/Operating-system-level_virtualization.

[2] "Syzkaller - google/syzkaller," https://github.com/google/syzkaller.

[3] "Docker," https://www.docker.com/what-container.

[4] "Linux containers," https://linuxcontainers.org/.

[5] "rkt - coreos," https://coreos.com/rkt/.

[6] "Hardware virtualization," https://en.wikipedia.org/wiki/Hardware_virtualization.

[7] "Linux namespaces," https://en.wikipedia.org/wiki/Linux_namespaces.

[8] "Cgroups," https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/ch01.

[9] "syzkaller: the next gen kernel fuzzer," https://www.slideshare.net/DmitryVyukov/syzkaller-the-next-gen-kernel-fuzzer.

[10] "Container adoption survey," https://portworx.com/2017-container-adoption-survey/.

[11] "Container adoption," https://www.theregister.co.uk/2017/09/11/container_adoption_still_low_says_cloud_foundation/.

[12] "Linux kernel source tree," https://github.com/torvalds/linux.

[13] "NVD - national vulnerability database," https://nvd.nist.gov/.

[14] "CVE security vulnerability database," https://www.cvedetails.com/.

[15] "CVE - common vulnerabilities and exposures," https://cve.mitre.org/.

[16] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna, "Difuze: Interface aware fuzzing for kernel drivers," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 2123–2138.

[17] "Gnu cflow," https://www.gnu.org/software/cflow/.

[18] "Kernel.org git repositories," https://git.kernel.org/.

[19] "Linux man pages online," http://man7.org/linux/man-pages/index.html.

[20] "Kernel: add kcov code coverage," https://lwn.net/Articles/671640/.

[21] "The kernel address sanitizer," https://lwn.net/Articles/612153/.

[22] W. You, P. Zong, K. Chen, X. Wang, X. Liao, P. Bian, and B. Liang, "Semfuzz: Semantics-based automatic generation of proof-of-concept exploits," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 2139–2154.

[23] H. Han and S. K. Cha, "Imf: Inferred model-based fuzzer," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 2345–2358.

[24] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 2329–2344.

[25] V. M. Weaver and D. Jones, "perf fuzzer: Targeted fuzzing of the perf event open () system call," Technical Report UMAINEVMW-TR-PERF-FUZZER, University of Maine, Tech. Rep., 2015.

[26] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[27] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009, pp. 474–484.

[28] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-fuzz: fuzzing by program transformation."

[29] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," *IEEE Transactions on Software Engineering*, 2017.

[30] B. Garn and D. E. Simos, "Eris: A tool for combinatorial testing of the linux system call interface," in *Software Testing, Verification and Validation Workshops (ICSTW), 2014 IEEE Seventh International Conference on*. IEEE, 2014, pp. 58–67.

[31] Z. Li and Y. Zhou, "Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5. ACM, 2005, pp. 306–315.

[32] B. Liang, P. Bian, Y. Zhang, W. Shi, W. You, and Y. Cai, "Antminer: mining more bugs by reducing noise interference," in *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*. IEEE, 2016, pp. 333–344.

[33] P. T. Breuer and S. Pickin, "One million (loc) and counting: Static analysis for errors and vulnerabilities in the linux kernel source code," in *International Conference on Reliable Software Technologies*. Springer, 2006, pp. 56–70.

[34] L. Luo, Q. Zeng, C. Cao, K. Chen, J. Liu, L. Liu, N. Gao, M. Yang, X. Xing, and P. Liu, "System service call-oriented symbolic execution of android framework with applications to vulnerability discovery and exploit generation," in *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 2017, pp. 225–238.

[35] R. Johnson and D. Wagner, "Finding user/kernel pointer bugs with type inference." in *USENIX Security Symposium*, vol. 2, no. 0, 2004, p. 0.

[36] "Recurrent neural network - wikipedia," https://en.wikipedia.org/wiki/Recurrent_neural_network.

[37] "Natural-language processing - wikipedia," https://en.wikipedia.org/wiki/Natural-language_processing.

[38] "Cscope," http://cscope.sourceforge.net/.