软件学报 ISSN 1000-9825, CODEN RUXUEW
2026,37(3):1427−1446 [doi: 10.13328/j.cnki.jos.007507] [CSTR: 32375.14.jos.007507]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

容器文件系统隔离增强机制*

李 志[1,2,3,5], 夏书婷[1,2,3,5], 李圣杰[1,2,3,5], 刘维杰[7], 王振辰[1,2,3,5], 金 海[1,2,4,6]

[1](大数据技术与系统国家地方联合工程研究中心, 湖北 武汉 430074)
[2](服务计算技术与系统教育部重点实验室, 湖北 武汉 430074)
[3](大数据安全湖北省工程研究中心, 湖北 武汉 430074)
[4](集群与网格计算湖北省重点实验室, 湖北 武汉 430074)
[5](华中科技大学 网络空间安全学院, 湖北 武汉 430074)
[6](华中科技大学 计算机科学与技术学院, 湖北 武汉 430074)
[7](南开大学 密码与网络空间安全学院, 天津 300071)
通信作者: 刘维杰, E-mail: weijieliu@nankai.edu.cn

摘 要: 随着容器技术的广泛应用, 容器技术的安全性和隔离性受到广泛关注. 目前, 各类容器工具中长期存在大量容器逃逸漏洞, 其中由于容器文件系统隔离不足导致的安全漏洞已成为占比较大的一类安全威胁. 此类漏洞允许攻击者在容器与宿主机交互时操纵容器内文件路径解析过程或窃取宿主机中文件描述符来实施容器逃逸. 各容器工具社区虽实践了多种修复方法, 但仍无法彻底消除此类漏洞, 甚至因修复不彻底而引入了多个同类的新漏洞. 为彻底消除此类漏洞, 需从内核层面完善容器文件系统隔离机制. 因此提出了一种细粒度容器文件系统隔离增强机制, 将容器文件系统隔离从原有仅隔离文件系统挂载点扩展到 inode 级别. 该机制将对容器内文件的 inode 进行标识, 以区分容器与宿主机的文件对象, 继而基于标识设计并实施容器与宿主机间的访问控制, 以完成对容器与宿主机间文件系统隔离的增强. 实验结果表明该机制能够有效阻止所有文件系统相关的容器逃逸漏洞, 引入的平均开销低于 2%, 且远低于容器工具提供的漏洞补丁带来的开销.

关键词: 容器安全; 漏洞修复; 容器隔离; 文件系统隔离; 容器逃逸
中图法分类号: TP316

中文引用格式: 李志, 夏书婷, 李圣杰, 刘维杰, 王振辰, 金海. 容器文件系统隔离增强机制. 软件学报, 2026, 37(3): 1427–1446. http://www.jos.org.cn/1000-9825/7507.htm
英文引用格式: Li Z, Xia ST, Li SJ, Liu WJ, Wang ZC, Jin H. Isolation Enhancement Mechanism for Container File System. Ruan Jian Xue Bao/Journal of Software, 2026, 37(3): 1427–1446 (in Chinese). http://www.jos.org.cn/1000-9825/7507.htm

Isolation Enhancement Mechanism for Container File System

LI Zhi[1,2,3,5], XIA Shu-Ting[1,2,3,5], LI Sheng-Jie[1,2,3,5], LIU Wei-Jie[7], WANG Zhen-Chen[1,2,3,5], JIN Hai[1,2,4,6]

[1](National Engineering Research Center for Big Data Technology and System, Wuhan 430074, China)
[2](Key Laboratory of Services Computing Technology and System, Ministry of Education, Wuhan 430074, China)
[3](Hubei Engineering Research Center on Big Data Security, Wuhan 430074, China)
[4](Cluster and Grid Computing Lab, Wuhan 430074, China)
[5](School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan 430074, China)
[6](School of Computer Science and Engineering, Huazhong University of Science and Technology, Wuhan 430074, China)
[7](College of Cryptology and Cyber Science, Nankai University, Tianjin 300071, China)

* 基金项目: 国家自然科学基金 (62202191)
收稿时间: 2024-09-05; 修改时间: 2025-05-06; 采用时间: 2025-07-23; jos 在线出版时间: 2025-12-10
CNKI 网络首发时间: 2025-12-11

Abstract: With the widespread application of container technology, the security and isolation of containers have attracted significant attention. Currently, a large number of container escape vulnerabilities persist in various container tools, with the security vulnerabilities due to inadequate container file system isolation becoming a type of security threat that occupies a significant proportion. This kind of vulnerability allows attackers to manipulate file path resolution processes within containers or steal file descriptors from the host machine during interactions between containers and the host machine. Although multiple fix methods have been practiced by various container tool communities, these vulnerabilities cannot be thoroughly eliminated, and even new similar vulnerabilities are introduced due to the incomplete fix. It is necessary to improve container file system isolation mechanisms at the kernel level to thoroughly eliminate these vulnerabilities. Therefore, this study proposes a fine-grained isolation enhancement mechanism for container file systems, which extends container file system isolation from merely isolating file system mount points to the inode level. This mechanism involves marking the inode of files within containers to distinguish the file objects of containers and the host machine, followed by designing and implementing access control between containers and the host machine based on these markings to enhance file system isolation between containers and the host machine. Experimental results demonstrate that this mechanism can effectively prevent all file system-related container escape vulnerabilities, and the introduced average overhead is less than 2%, significantly lower than the overhead introduced by vulnerability patches provided by container tools.

Key words: container security; vulnerability fix; container isolation; filesystem isolation; container escape

容器技术是一种操作系统级别 (OS-level) 的虚拟化技术, 可为云应用提供轻量、高效、标准化的运行环境, 广泛用于云原生服务的构建、测试与部署. 容器技术的实现依赖于多种 Linux 内核机制, 命名空间 (namespaces)[1] 和控制组 (cgroup)[2]两种机制是保障容器与宿主机间、容器与容器间隔离的基础. 命名空间为容器实例提供独立的资源视图, 容器内文件系统挂载点 (mountpoint)[3]、网络设备及协议栈、进程间通信、用户组等资源经命名空间隔离后仅容器中进程可见. 控制组用于统计与限制容器实例中的资源用量, 确保不会因单一容器消耗过多资源而影响主机或其他容器. 然而, 命名空间与控制组对系统资源的隔离仍不完备, 引发了多种安全问题.

容器逃逸是一类典型的安全问题, 由于容器与宿主机间文件系统隔离不彻底, 文件路径解析错误和文件描述符 (file descriptor) 泄露等两类漏洞常被利用于容器逃逸. 这两类漏洞在流行的容器工具中占据相当高的比例, 包括 Docker[4]、Podman[5]等管理引擎, Containerd[6]等运行时 (runtime) 和 Kubernetes[7]等容器编排引擎. 据统计, 常用的容器工具 2017–2023 年报告的 27 个高危性漏洞中有近一半是文件路径解析错误漏洞[8]. 路径解析错误漏洞指容器中的攻击者能够在容器工具访问容器内文件时将目标文件替换为符号链接 (symbolic link, symlink), 诱导容器工具解析并访问符号链接所指向的宿主机文件, 从而实施逃逸. 例如, 当容器用户请求 Docker 将特定文件 (user-file) 复制到容器中指定路径 (container-dir) 时 (docker cp user-file [container-ID]:/container-dir), 实施复制行为的 Docker 进程会以宿主机的上下文解析容器中的路径 (container-dir), 若容器中的攻击者将该路径 (container-dir) 替换为一个指向敏感路径 (如/etc/passwd) 的符号链接, Docker 进程则会将容器用户指定的文件 (user-file) 复制到符号链接所指向的宿主机目录, 从而覆盖目录中的重要文件[9]. 文件描述符泄露漏洞主要源于攻击者在容器中获得了泄露在容器环境的主机文件的文件描述符, 然后利用该文件描述符进行逃逸. 具体而言, 攻击者可以在宿主机进程与容器交互时从容器内部获取宿主机进程所打开文件的文件描述符, 进而通过该文件描述符访问和修改主机上的文件, 破坏了容器与主机之间的隔离性.

当前容器工具对上述两类漏洞的修复方案并未彻底解决问题, 这不仅带来了显著的性能开销, 同时容易被绕过, 进而引发了多个新漏洞. 例如 CVE-2021-25741 漏洞[10]的成因就与上段介绍的 CVE-2017-1002101 的修复方案密切相关. 这是由于修补手段通常针对某一特定漏洞, 没有涉及漏洞产生的根本原因. 经过对漏洞成因的剖析后发现, 形成上述两类漏洞的根本原因在于容器和主机间文件系统的不完全隔离. 当前文件系统的隔离设计只隔离了不同挂载命名空间 (mount namespace)[11]的挂载点视图, 这种隔离手段并未能防止通过虚拟文件系统 (virtual filesystem, VFS) 层次进行的路径解析过程中的漏洞利用. 只要能通过 VFS 访问文件的步骤将访问路径转化为文件索引节点 (inode) 或是通过文件描述符直接获取文件对应的 inode, 内核便允许对相关文件的访问. 这种机制在某种程度上为攻击者提供了绕过隔离的可能性, 使得路径解析错误与文件描述符泄露漏洞成为一种持久且难以彻底根除的安全威胁.

完备的文件系统隔离机制是全面消除该类路径解析错误及文件描述符泄露漏洞的首选方法. 有效的主机与容器文件系统隔离机制应该达到以下目标: (1) 隔离容器和宿主机间文件系统相关的所有数据结构, 保护主机的敏感数据和系统文件, 同时防止主机被欺骗执行容器中的恶意可执行文件; (2) 具有完整的权限检查机制, 确保容器内的进程只能访问其具有合法权限的文件和目录. 这两点可以防止攻击者通过解析恶意符号链接或者访问主机上的文件描述符破坏容器环境的安全性.

本文提出了一种容器文件系统隔离增强机制, 将文件系统隔离拓展到文件的 inode 结构. 在容器初始化过程中, 通过在 inode 结构中添加指向 PID 命名空间的指针来标记文件 inode, 并根据容器文件系统、共享卷和宿主机文件系统的不同特性, 为这些 inode 分配不同的安全级别标志. 此外还精心设计了两个访问控制策略, 分别限制路径解析结果和控制流的转移, 来确保准确和全面的交互控制. 我们已经将源代码提交至开源网站[12], 用户可以下载并应用补丁.

本文贡献如下.

(1) 介绍了一个文件描述符泄露漏洞, 该漏洞揭示了 Podman 容器环境中文件描述符泄露的新途径, 并对攻击原理和过程进行了详细的说明, 展示了攻击者如何通过“podman top”命令存在的缺陷, 获得宿主机文件的文件描述符, 并以此为基础访问和修改宿主机文件.

(2) 分析了路径解析错误和文件描述符泄露漏洞的根本原因是容器文件系统隔离不完整. 通过对多个漏洞原理的研究分析后, 认为仅针对单个漏洞进行修复是无效的, 无法防止相似漏洞的再次出现, 甚至修复手段本身也可能引入新漏洞.

(3) 提出并实现了内核层增强容器文件系统隔离的方法 FCFS (fine-grained container file system). 这种方法通过将容器文件系统隔离扩展到 inode 级别, 并结合精心设计的访问控制策略, 来确保准确和全面的交互控制.

(4) 通过测试验证了本方法的有效性和高效性. 首先检验了 FCFS 的有效性, FCFS 成功捕获并阻止了所有不安全的跨范围访问进程, 可以完全消除上述两类安全风险. 其次验证 FCFS 的高效性, 测试评估了 FCFS 在 Docker、Podman 和 Kubernetes 上的实现, 引入的平均开销低于 2%, 性能远优于容器工具提出的各种用户层修复手段.

本文第 1 节介绍所需的基础知识, 包括虚拟文件系统和容器的文件系统隔离机制. 第 2 节介绍对两类漏洞的原理及现有修复手段, 分析解决漏洞的根本方法. 第 3 节介绍提出的容器文件系统隔离增强机制 FCFS. 第 4 节验证 FCFS 的有效性, 并通过对比实验验证其高效性. 最后总结全文.

1 基础知识

1.1 虚拟文件系统

VFS 是操作系统内核中的一个抽象层[13], 它允许不同类型的文件系统在统一框架下进行操作. VFS 的设计体现了“万物皆可为文件”的理念, 普通文件、目录、字符设备等都被视为文件. 它提供了一个通用的接口, 使得应用程序可以通过一致的方式访问各种不同的文件系统, 而不需要关心底层文件系统的具体实现细节, 同时也为不同文件系统的通信提供了媒介.

● 文件描述符

文件描述符是进程访问文件的句柄, 实际上是一个非负整数, 用来高效管理被打开的文件. 当进程启动后, 进程的进程控制块 (PCB) 中有一个指针指向进程的文件描述符表, 记录了该进程打开的所有文件. 进程的文件描述符表是一个数组, 数组中的每个元素是一个指向打开文件结构体的指针. 此外, 内核为所有打开的文件维护了一个系统级的描述表, 记录所有当前打开的文件. 每个条目是一个文件结构体, 包含文件的状态信息、当前偏移量等.

调用 open 系统调用打开文件时, 若成功打开, 内核会向进程返回一个指向打开文件的文件描述符. 文件描述符的值由操作系统自动分配, 规定从 0 依次增加. 在 POSIX 语义中, 0、1、2 这 3 个文件描述符值被赋予特殊含义, 分别是标准输入 (STDIN_FILENO), 标准输出 (STDOUT_FILENO) 和标准错误 (STDERR_FILENO). 因此用户进程打开文件得到的最小描述符只能是 3. 后续所有执行 I/O 操作的系统调用都通过返回的文件描述符来实现. 由

于每个进程都有自己的文件描述符表, 所以不同进程可以拥有相同的文件描述符, 这些文件描述符可能指向相同或不同的文件. 如果进程多次打开同一文件, 它的不同文件描述符可能指向相同文件. 文件描述符的生命周期与打开它的进程相关, 当进程关闭文件描述符或进程终止时, 文件描述符会被释放.

● dentry

dentry (即 directory entry)[14]是 Linux 内核对文件系统中目录项的抽象, 该结构体用于关联文件或目录的名称与对应的 inode, 并被缓存在内存中以提高目录项的查找与访问效率. 此外, dentry 结构体还记录了目录项间的层级关系. 遵循目录项间的父子关系, dentry 结构体中包含指向其父 dentry 和子 dentry 列表的指针, 从而将内存中各目录项的 dentry 组织成了一棵目录树. 当内核解析应用层传入的文件路径时, 内核会以“/”为边界将文件路径分割为多个路径组件, 继而在目录树中递归查找各路径组件对应的 dentry, 直到完成对最后一项路径组件的解析. 由于内存中缓存了大部分常用目录项的 dentry, 路径解析过程无需频繁进行磁盘 I/O 以读取文件相关信息, 提高了目录项查找效率. 而当系统内存紧张或文件状态改变时 (删除或重命名等), 部分 dentry 将会从内存中回收. 若路径解析过程中未查找到路径组件对应的 dentry, 则需从磁盘上重新读取目录项并创建相关 dentry 加入缓存.

● inode

文件系统将文件数据和文件元信息分开存放, 以提高文件系统的性能和灵活性. 文件数据保存在数据块 (data block) 中, 文件元信息则保存在 inode 中. 文件元信息是对文件属性的描述信息, 包括文件大小、拓展属性、链接数量和文件数据的块位置等[13]. 在文件存取过程, 内核需要找到文件的实际数据块来读取或写入文件内容, 该过程首先通过文件名逐步查找到文件对应的 dentry, dentry 中有指向文件 inode 的指针, 再根据 inode 中文件数据块的磁盘位置信息, 查找到文件的实际数据块, 实现文件的读写操作. 这样可以提高文件系统的性能, 减少文件查找的时间.

符号链接是一种特殊的文件类型, 其 inode 结构中存储的是指向目标文件路径名的指针, 而不是目标文件的 inode 号. 因此, 符号链接与目标文件之间的关系是通过路径名解析的, 而不是通过 inode 号直接关联, 目标文件 inode 号的链接数量不变. 当访问符号链接时, 文件系统会首先读取符号链接文件的 inode, 从 inode 中获取路径名, 然后对路径名进行解析, 最终找到目标文件的 inode, 访问目标文件.

● 文件操作流程

通过路径名操作文件时, 会触发路径查找 (path lookup), 这是 VFS 中的关键功能. 它对路径字符串进行一层层解析, 最终找到对应文件的 dentry 和 inode. 路径查找过程中会将路径名分割成各个路径组件, 每个组件是由“/”字符分隔的子字符串. 在初始化查找过程中, 内核会创建一个 nameidata 临时结构, 主要作用是保存查找过程中的中间结果. nameidata 结构由查找过程的初始组件初始化, 如果路径名以“/”字符开头, 则起点是调用进程根目录的 dentry. 对于每个路径组件, 内核会查找对应的 dentry. 如果找到, 则更新 nameidata 结构中的信息, 包括当前 dentry 和 inode 等; 如果找不到组件的 dentry, 则路径查找过程返回失败. 如果路径组件是一个符号链接, 内核会读取符号链接的目标路径, 并递归解析该目标路径, 以确保找到最终的文件 inode. 内核还会检测是否存在循环符号链接, 以防止无限递归.

当用户通过文件描述符对文件进行操作时, 内核会直接通过文件描述符从当前进程的文件描述符表找到系统级打开文件表中对应的文件信息, 其中包括指向文件 inode 的指针, 通过该指针访问缓存数据, 不需要经过路径解析. 图 1 展示了以上两种操作文件方式的过程.

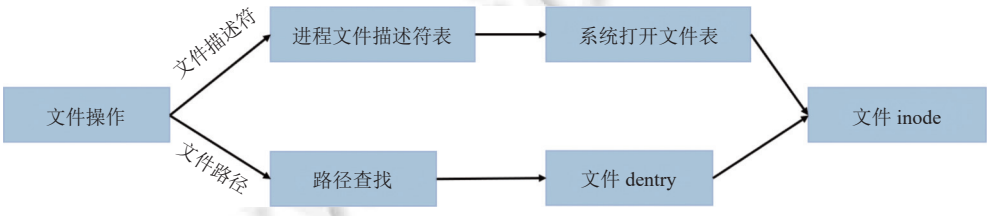


图 1 两类文件操作流程示意图

1.2 容器文件系统隔离机制

容器与主机的文件系统隔离机制通过挂载命名空间、chroot 等技术实现, 确保容器在一个独立的文件系统视

图中运行.

挂载命名空间是容器隔离的重要机制之一. 它允许每个容器拥有独立的挂载点视图, 使容器内的文件系统操作不影响主机或其他容器. 创建一个新挂载命名空间时, 会复制当前命名空间的挂载点列表, 之后任何在该新命名空间内进行的挂载或卸载操作都不会影响其他命名空间. 也就是说, 挂载命名空间实际上隔离的是文件系统挂载点, 使得进程只能看到属于其命名空间的挂载点. 当容器引擎 (如 Docker 或 Podman) 启动一个容器时, 会为其创建一个新的挂载命名空间. 容器内的文件系统布局 (如根文件系统、挂载的卷) 可以与主机完全不同.

chroot 系统调用通过更改程序的根目录, 确保进程只能看到并访问新的根目录及其子目录. chroot 将进程的根目录更改为指定目录, 使进程不能访问该目录之外的文件系统部分. 通过改变根目录, 进程被限制在新的文件系统视图中, 无法访问主机的其他部分. 设置 chroot 环境不改变实际系统文件, 又能实现安全隔离, 因此在容器化环境中用于隔离文件系统.

创建和隔离容器文件系统的过程如下. 逻辑容器创建并启动后, 从容器镜像 (image) 中提取根文件系统 (rootfs) 挂载到容器运行根目录下以 ID 为名的容器工作目录 (/var/lib/docker/containers/[容器 ID]). rootfs 包含了系统运行有关的文件、配置和目录, 但不包括系统内核. 然后调用设置 CLONE_NEWNS 参数的 clone 系统调用, 创建一个新的挂载命名空间, 准备挂载文件系统. 新的挂载命名空间从当前命名空间复制挂载点列表, 意味着容器进程此时仍然可以看到主机中的所有挂载点, 这显然达不到预期的隔离要求. 因此还需要利用类似 chroot 的 pivot_root 系统调用将容器的根目录切换到 rootfs, 确保容器只能访问当前根目录及其子目录, 无法访问主机的其他路径, 至此文件系统隔离完成.

除了上述的资源视图隔离方法外, Linux 内核本身也引入了一些安全模块来进行安全检查, 如 seccomp[15]和 AppArmor[16]. seccomp 主要用来限制某一进程可用的系统调用, 通过编写特定格式的过滤器, 可以对进程执行的系统调用的系统调用号和参数进行检查和过滤. 通过禁止进程调用不必要的系统调用, 减少了内核暴露给用户态进程的接口数量, 从而减少了内核攻击面. Docker 启动时会启用默认的 seccomp 配置文件, 仅保留部分 Linux 中较为常见且安全的系统调用. 而 AppArmor 通过内核安全模块实现强制访问控制——在自主访问控制之上, 进一步将程序限制在有限的资源集中. 管理员可以为每个进程制定详细的访问控制策略, 以限制对文件、网络资源等接口的访问.

目前的容器工具主要依赖于挂载命名空间和安全模块检查来实现文件系统隔离, 辅以不同的隔离技术. 例如, 大多数容器工具通过默认不授予容器 CAP_SYS_ADMIN 能力[17]来防止容器用户进行系统管理员操作, 如挂载或卸载文件系统等, 限制了受感染的容器对主机的威胁. 此外, Docker 还采用了一种写时复制 (copy on write) 文件系统[18]来保证不同容器间的数据隔离. 当多个容器共享同一镜像时, 只有容器对文件进行写操作时, 才会在容器对应的文件系统内生成文件副本, 并对副本进行修改, 不影响镜像源文件. 但是基于目前的隔离手段, 仍有很多漏洞利用容器和宿主机交互过程绕过容器与宿主机间的隔离, 给主机带来了安全威胁.

2 容器漏洞与修复

2.1 威胁模型

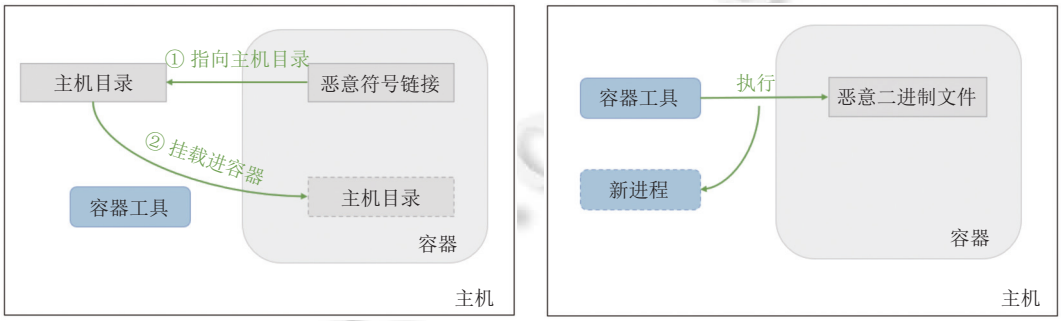
在现代化的云计算和微服务架构中, 容器技术作为轻量级的虚拟化解决方案, 允许多个容器在单一宿主机操作系统上共存. 每个容器都拥有独立的文件系统、网络空间以及进程空间, 这种隔离性为多租户环境提供了必要的安全屏障. 云平台中容器工具 (如 Docker、Podman 或 runC[19]) 负责管理容器实例从创建到销毁的整个生命周期. 本文的讨论基于一个基本假设: 云平台中宿主机操作系统及容器工具等核心组件均是可信的, 并能正确实施容器实例生命周期的管理. 但云平台无法杜绝外部攻击者利用容器中服务的漏洞入侵容器实例、恶意用户通过正常的接口部署包含恶意代码的容器实例. 攻击者和恶意用户拥有在受感染/恶意容器实例内执行任意代码的能力, 可以操纵容器实例中的进程、文件系统以及网络配置. 恶意用户与正常用户一样可以通过云平台提供的 API 与容器实例进行交互, 如请求容器工具将指定文件复制到容器实例中. 攻击者/恶意用户的最终目标是打破容器的隔离边

界, 实现容器逃逸, 即获得直接访问宿主机资源的权限, 进而横向移动获取整个云平台的控制权限.

随着持续集成与持续部署 (continuous integration and continuous delivery, CI/CD) 的组合实践与云原生应用开发、测试、部署过程的深度结合[20], 容器成为 CI/CD 实践的基础运行环境, 也增加了宿主机与容器实例间交互的需求. 这类交互通常由容器所有者发起, 意图将主机资源导入容器或将处理的结果从容器导入主机. 然而, 此类交互也为容器环境引入了新的攻击面, 允许容器化的恶意应用程序欺骗容器工具访问恶意负载 (payload). 路径解析错误与文件描述符泄露等两类漏洞均需利用容器-宿主机交互过程绕过容器与宿主机间的隔离, 进而窃取宿主机中敏感信息或进行提权攻击.

2.2 路径解析错误漏洞原理及修复

文件路径解析错误漏洞根据攻击者行为的不同, 可以分为两类: 一类是符号链接解析欺骗漏洞, 一类是诱导非法文件执行类漏洞. 图 2 简单展示了这两类漏洞的攻击过程.



(a) 符号链接解析欺骗漏洞攻击过程　　(b) 诱导非法文件执行漏洞攻击过程

图 2　两类路径解析错误漏洞攻击过程

在主机和容器交互期间, 容器工具可以访问容器文件系统中的符号链接, 对其读取或写入. 但是如果容器将符号链接指向了非预期位置, 由于该符号链接是在主机上下文中解析的, 容器工具在处理文件路径时可能会使容器访问到主机上的敏感目录. 这就是符号链接解析欺骗的大致原理. 例如, 容器工具提供的复制 (copy) 功能用于在容器和主机之间复制文件. 但是从容器向主机中复制文件的源文件参数和从主机向容器内复制文件的目的文件参数都由容器控制, 容器可以替换参数. 如果恶意符号链接替换了从容器向主机中复制文件时的源文件参数, 那么该符号链接指向的文件会被导入主机环境, 对主机环境造成破坏. 同样, 如果替换了从主机向容器中复制文件场景下的目的文件, 也会导致文件被复制到主机上的目录. Podman 中出现了类似的漏洞 CVE-2019-10152[21]和 CVE-2019-18466[22].

卷 (volume) 是一个可供一个或多个容器使用的特殊目录, 它将主机操作系统目录直接映射进容器, 并且可以在容器之间共享和重用. CVE-2021-30465 漏洞[23]产生的原因就是挂载数据卷时恶意符号链接替换了目标文件参数. runC 操作目标目录时, 在检查目标文件参数和实际挂载到该目录之间存在一个时间差, 攻击者在这段时间内将目标文件替换为一个指向主机目录 (如“/”) 的符号链接, 该主机目录随后会被挂载进容器中. 类似的漏洞还有 CVE-2017-1002101、CVE-2019-19921[24]和 CVE-2022-23648[25].

诱导非法文件执行是另一类利用符号链接解析进行攻击的方法. 除了读取或写入之外, 在主机与容器交互期间容器中的文件还可以由主机进程 (例如容器工具) 执行, 这种执行可能会导致容器中的恶意文件在主机上下文中起作用. 主机进程很难完全避免执行这类被篡改的容器文件. CVE-2022-0492 漏洞[26]的发生是由于容器修改了 cgroup 中一个特殊文件“release_agent”. 该文件由 cgroup 文件系统提供, 是主机和容器之间共享 cgroup 信息的交互接口, 只要攻击者能够对“release_agent”文件执行写操作, 就能强制内核以提升后的权限执行指定的二进制文件, 进而控制整个机器.

每当发现上述漏洞, 容器工具社区都会尝试修复. 但是多年来类似漏洞层出不穷, 并对多种容器工具产生严重

影响, 这是由于漏洞修复手段局限于用户层面, 没有从根本上解决漏洞产生的原因. 目前的修复手段可能引入高性能开销, 很容易被绕过, 甚至可能引入新的漏洞.

为了防止符号链接解析欺骗的发生, runC 会使用“securejoin”包的 SecureJoinVFS 函数来解析传进来的路径是否合法, 却忽略了解析检查过程和随后的路径相关操作 (例如复制或挂载) 之间存在竞争条件, 可以通过 TOCTTOU 攻击[27]来利用恶意符号链接替换已验证的路径, 导致漏洞 CVE-2018-15664[28]、CVE-2021-25741、CVE-2021-30465、CVE-2022-23648 和 CVE-2023-0778[29]等. Podman 为了避免此类竞争漏洞, 在 Podman 访问容器的文件系统时暂停容器, 然而当 Podman 与冻结的容器交互时, 容器仍然可以访问与其他容器共享的卷并进行 TOCTTOU 攻击[30]. 此外, 尽管 Podman 开发人员已经尽量避免此类漏洞的发生, Podman 的导出卷 (export volume) 功能还是发现了一个类似 TOCTTOU 漏洞 CVE-2023-0778, 攻击者可以在导出卷时使用符号链接替换卷中的普通文件, 从而能够访问主机文件系统上的任意文件.

在修复最早的符号链接欺骗漏洞 CVE-2017-1002101 时, 容器社区的思路是将解析检查过程和路径操作原子化来消除竞争条件, 确保恶意用户通过 subPath 挂载的路径不是非预期的符号链接. 即使 Kubernetes 已经修复了该漏洞, 但是其依赖组件中还是存在类似的攻击面, 导致了漏洞 CVE-2021-30465 和 CVE-2022-23648. 并且该修复方案本身也引入了新的漏洞, 具体来说, 在使用 mount 系统调用之前, Kubernetes 会先使用 openat 打开子路径并验证这个路径确实位于卷内, 提前验证路径的安全性, 然后 Kubernetes 使用特殊符号链接“/proc/[pid]/fd/[fd]”进行绑定挂载 (bind mount), 此符号链接始终指向打开文件的文件描述符. 在 kubelet 仍然持有文件描述符的情况下, 即使原始文件被替换, 这个特殊符号链接仍然会指向原始文件, 从而消除了竞争条件, 避免了类似攻击的发生. 但是这种原子化能被后续操作调用的第三方可执行文件破坏. 在 Kubernetes 的卷功能中, 解析检查后调用的 Linux 挂载工具[31]默认会解析该特殊链接, 并调用 mount 系统调用挂载解析结果, 这给漏洞 CVE-2021-25741 带来了机会. CVE-2021-25741 的补丁只是在调用 mount 时传递了--no-canonicalize 参数, 使 mount 系统调用不再解析符号链接.

此外, 所有容器工具都努力避免执行非法文件, 但是仍有许多不可预见的执行隐藏于其功能中. 例如, 执行 Docker 提供的复制命令“docker cp”时使用了一个运行在主机命名空间的 docker-tar 进程[32], 为了修复漏洞 CVE-2018-15664、防止出现符号链接解析问题影响到主机, docker-tar 会 chroot 到容器的根目录再归档其中请求的文件及目录, 这样能确保所有符号链接都在其目录下被有效地解析. 但是这也为后续更严重的漏洞 CVE-2019-14271[33]埋下了伏笔. 一般而言, docker-tar 运行过程中需要从主机文件系统中加载动态链接库, 但是加载前 docker-tar 会先 chroot 到容器文件系统, 最终导致所加载的动态链接库来源于不可信的容器环境. 此外, 虽然 docker-tar 会 chroot 到容器文件系统, 但是其本身仍运行在主机命名空间且具有 root 权限, 若所加载的容器中动态链接库存在恶意代码, 该恶意代码将在宿主机上下文中以 root 权限运行, 进而完成容器逃逸. 容器社区通过在复制功能开始时加载所有动态库来避免此漏洞, 但是 GNU CLibrary (glibc)[34]的更新对任何容器工具都是透明的, 这一更新使得包括“nsswitch”在内的 glibc 库在 chroot 触发上下文切换后能够自动重新加载, 可能会使此修复无效. 类似 CVE-2019-14271 的漏洞还有 Podman 中的 CVE-2022-1227 漏洞[35]. 该漏洞的原理是 podman top 功能调用了容器中的 nsenter 可执行文件.

2.3 文件描述符泄露漏洞原理及修复

文件描述符泄露漏洞的攻击者通过主机泄露在容器内的主机文件描述符来访问甚至修改对应的主机文件, 实现容器逃逸, 图 3 展示了该类漏洞的攻击过程.

Proc 文件系统是 Linux 内核信息的抽象文件接口, 用于通过内核访问进程信息. 每个正在运行的进程都在/proc 中有一个对应的目录, 命名为该进程的 PID. 这些目录包含了进程的各种状态和资源信息[36]. 其中有一个子目录/proc/[PID]/fd 包含了该 PID 所属进程打开的所有文件描述符, 通过查看这些文件描述符可以获取进程当前打开的文件及其路径./proc/[PID]/exe 是一种指向进程自身对应的本地程序文件的特殊符号链接文件, 例如执行 ls 命令时, /proc/[PID]/exe 会指向/bin/ls. 打开/proc/[PID]/exe 时, 若权限检查通过, 内核会直接返回指向目标文件的文件描述符, 不需路径查找过程. 在主机与容器交互期间 (执行 docker exec 命令), 容器内的攻击者可能欺骗进入容器

环境的容器工具 (runC) 去访问其中的/proc/[PID]/exe(/proc/[runc-pid]/exe), 就可以得到程序文件 (/usr/bin/runc) 的文件描述符 (/proc/self/fd/[fd]), 并通过文件描述符覆盖文件, 造成安全问题. 漏洞 CVE-2019-5736 即为容器内的进程通过此通道重写 runC 可执行文件造成的[37].

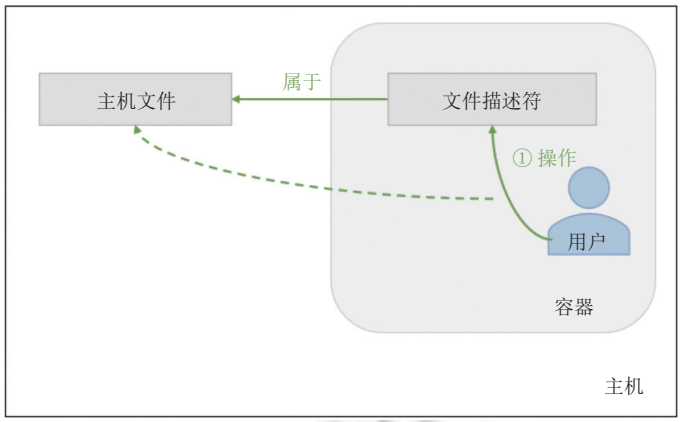


图 3 文件描述符泄露漏洞攻击过程

漏洞 CVE-2024-21626[38]是通过/proc/self/fd/实现的, 其中包含了当前进程打开的文件描述符. runC 启动过程中 openat2 系统调用缺少 O_CLOEXEC 标志 (执行 exec 前自动关闭文件描述符), 导致存在没有及时关闭的文件描述符, 该文件描述符指向主机上的/proc/fs/cgroup, 子进程仍可以利用它的/proc/self/fd/[fdnum] 符号链接访问主机文件, 实现容器逃逸.

类似漏洞还有本文发现的一个 Podman 上的安全缺陷, 漏洞原理如图 4 所示. Podman 提供“podman top”命令用于显示在容器中运行的进程[39], 一般情况下, 该命令会进入容器的挂载命名空间并在 Proc 中获取相应的进程信息. 但是当容器具有 SYS_PTRACE 特权时, “podman top”会调用“crun exec”在容器中运行 ps 命令 (①). 这个过程中, crun exec 会启动一个 crun 进程并将其加入容器的命名空间中 (②). crun 在加入容器的命名空间之前打开了一些文件描述符, 这些文件描述符对容器内的进程是可见的, 并且其中一个文件描述符指向 /run/user/1000/crun/[CONTAINER_ID]/seccomp.bpf 文件.

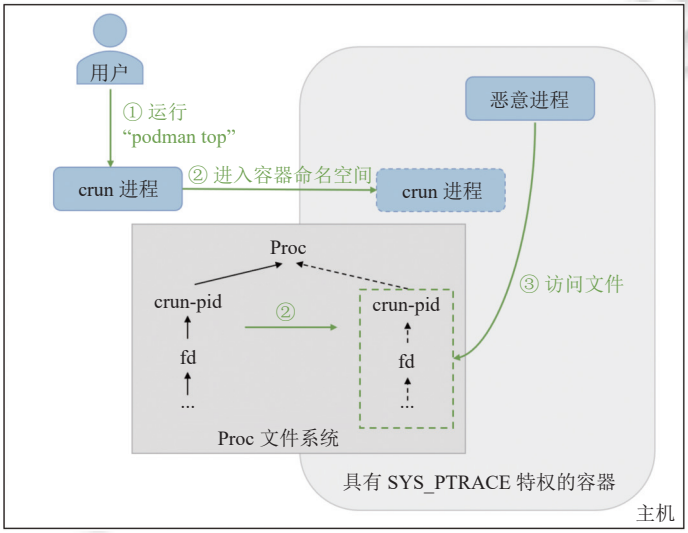


图 4 Podman 漏洞攻击示意图

因此容器内具有 SYS_PTRACE 特权的攻击者可以通过访问/proc/[crun-pid]/fd 来修改这个文件描述符指向的文件, 即 seccomp.bpf 文件 (③). 这会导致禁用 seccomp 过滤, 从而绕过 seccomp 限制, 提升权限. 此外, 在 /proc/[crun-pid] 目录下存在许多其他可能被利用的途径. 例如, 攻击者可以覆盖/proc/PID/map_files, 通过这种方式, 攻击者可以从 non-rootless 容器中逃逸, 获得主机系统的访问权限.

这类与伪文件系统相关的漏洞很难通过容器工具彻底修复. CVE-2019-5736 漏洞与 Proc 文件系统相关, 容器工具的修复只能阻止该漏洞的攻击, 而无法消除 Proc 文件系统中存在的攻击面. 漏洞补丁 (patch) 的原理是在 runC init 进程进入容器命名空间之前生成 runC 可执行文件的副本, 这样就能防止 CVE-2019-5736 利用宿主机中真正的 runC 文件. “podman top”为了保护非特权容器, 为具有 SYS_PTRACE 特权的容器提供了专用代码, 却为漏洞攻击提供了机会. Podman 团队目前给出的缓解方法是禁用特权命名空间, 限制容器权限, 但是在某些场景下需要容器具有 SYS_PTRACE 特权以便能够跟踪和调试容器内的进程. 例如, 在开发和调试过程中, 当需要使用 gdb 或其他调试工具时, 容器需要 SYS_PTRACE 特权才能够跟踪调试其他进程.

此外, 与 cgroup 文件系统接口相关的漏洞 CVE-2022-0492 也无法通过容器工具消除. 它们的解决方案也只是禁用特权用户命名空间, 降低容器的权限, 以避免 release_agent 文件被容器修改, 但这会影响容器运行. 最终, 为了解决这个问题, 内核开发者发布了一个补丁[40], 对存在漏洞的 cgroup 文件系统接口进行了访问控制. 该补丁增加了一段代码, 进程若想修改 release_agent 文件, 除了拥有 CAP_SYS_ADMIN 权限外还需要处于根命名空间中, 这意味着仅仅通过 unshare 命令获得的 CAP_SYS_ADMIN 权限不再能够修改 release_agent 文件, 因为使用 unshare 创建的新命名空间并不是根命名空间. 但是如果容器本身就存在 CAP_SYS_ADMIN 权限则还可以继续使用此方式逃逸.

2.4 总 结

如前所述, 本文的研究对象是主机和容器间交互产生的路径解析错误漏洞和主机文件描述符泄露漏洞: 攻击者通过符号链接欺骗、非法文件执行或获得主机文件描述符等手段突破隔离限制, 最终实现容器逃逸.

本文根据不同攻击方式将路径解析错误漏洞分成了两类——符号链接解析欺骗漏洞与诱导非法文件执行漏洞. 其中符号链接解析欺骗漏洞主要是由于容器将符号链接指向了非法位置, 容器工具解析符号链接时使容器访问到主机上的敏感目录, 诱导非法文件执行漏洞的原理是主机进程执行了容器内的恶意文件. 现有的漏洞修复手段局限在用户层, 可能引入高开销并且容易被绕过, 导致多年来类似的漏洞层出不穷.

文件描述符泄露漏洞通常与伪文件系统有关, 容器通过访问 Proc 文件系统内泄露的主机文件描述符直接访问或操作相关的主机文件实现容器逃逸. 这类漏洞很难通过修改容器工具实现彻底修复, 目前的修复手段只能阻止某一特定漏洞的攻击, 但无法从根本上消除 Proc 文件系统中的攻击面, 如果只是限制容器权限, 不仅可能影响容器的使用, 而且如果容器本身具有此权限仍能利用该漏洞逃逸.

seccomp 或 AppArmor 也都无法解决这两类漏洞. 首先, seccomp 是一种针对系统调用的过滤机制, 而这两类漏洞产生的核心原因不在于容器内的系统调用接口, 并且如果禁用涉及的系统调用 (如 open、write), 因为其过于常见且必要, 会严重影响容器进程的正常文件操作. 其次, AppArmor 根据路径进行访问检查而非底层的文件 inode 来限制进程对敏感文件的访问, 只有当进程访问到配置文件中所列的敏感文件时才会记录该行为, 本身无法根绝漏洞的产生, 且如果用户通过主机文件描述符访问文件, 会绕过文件路径校验环节.

现有容器文件系统隔离工作[8,41]能够解决部分问题. Patrol 在文件 dentry 上附加标签以进行跨命名空间文件系统访问检查, 该类方法无法检测到文件描述符泄露漏洞, 通过文件描述符对文件进行操作的流程 (第 1.2 节) 不会经过文件 dentry, 因此不会触发其访问控制策略. vKernel 利用文件 inode 的 i_opflags 位来标识是否敏感文件: 初始化时根据 AppArmor 配置文件识别敏感文件并设置访问权限哈希表; 访问文件时会拦截通用的权限检查函数并检查 i_opflags 位. 这种方法对于敏感文件的设置没有脱离 AppArmor 的局限性, 只有在容器访问 AppArmor 配置中的文件时才能发挥作用, 无法防御住诱导非法文件执行漏洞.

3 容器文件系统隔离增强机制设计和实现

从内核隔离层面来看, 这两类漏洞持续存在的根本原因是容器和主机间不完全的文件系统隔离. 如第 1.2 节

中所述, 容器文件系统通常通过挂载命名空间和 pivot_root 提供一定程度的隔离, 但它们并不能完全覆盖所有 VFS 对象, 比如文件描述符和 inode. 内核解析用户空间的文件路径后无法区分 inode 对象是否属于容器, 即使访问可能是通过非法通道 (如"/proc/self/exe") 进行的, 只要能顺利得到文件 inode 对象便会允许访问. 这种不完全的隔离意味着如果该进程不受安全检查限制, 则可能将容器文件系统视为受信任的环境. 因此, 为了完全消除风险, 必须重构内核的文件系统隔离机制.

虽然内核中有各种可跨容器共享的文件相关的结构, 例如 dentry、fd_array 等, 但是经过对第 1.2 节中文件操作流程的分析, 无论是以路径名还是文件描述符对文件进行操作, 最终都需要得到该文件的 inode 对文件进行真正的操作, 所以从 inode 层面对内核文件系统进行隔离是更深层更有效的方式. 具体来说, 为了提供更精细的控制, 需要将文件系统隔离扩展到 VFS 中的 inode 对象, 确定每个 inode 是属于容器文件系统还是属于主机. 然后, 这种扩展的隔离机制可以基于进程和 inode 属性强制执行访问控制策略, 以防止容器和主机之间的非法访问. 通过实现这些措施可以提高容器环境的整体安全性. 这种方法从问题的根源入手, 不仅能消除现有漏洞, 还能预防未来可能出现的新型攻击.

为了彻底解决容器与主机交互过程中存在的符号链接解析欺骗漏洞以及文件描述符泄露漏洞, 本文提出了一种增强的容器文件系统隔离机制 FCFS, 其架构如图 5 所示. FCFS 首先通过在文件系统内的文件 inode 上附加特定标签, 将隔离的范畴进一步延伸至文件 inode 层级, 将该过程称为"静态渲染", 它赋予文件 inode 以身份标识, 从而初步区分出文件属于主机环境还是容器文件系统. 又由于在实际运行时, 主机进程可能进入容器文件系统, 面临被恶意攻击者诱导访问恶意符号链接的风险, 进一步引入了"动态路径查找渲染"的概念. 这一机制会在路径解析的过程中动态调整对 inode 的访问权限, 确保即便在运行环境中, 也能精准控制对每个 inode 的访问. 在经过了静态和动态渲染后, 内核可以准确分辨文件 inode 属于主机还是容器文件系统. 基于这种能力, FCFS 能够制定并实施合适的访问控制策略, 有效阻止试图跨越文件系统的恶意攻击行为. 将渲染过程和访问控制嵌入到 Linux 提供的路径查找过程中, 强化隔离效果的同时还确保了整个过程的透明度和一致性, 使隔离措施成为系统运行的有机组成部分, 而不是一个附加的独立安全层.

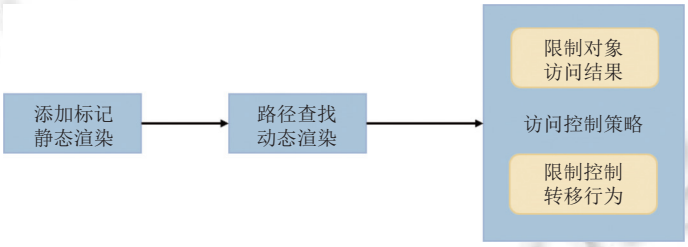


图 5 FCFS 隔离机制生效流程

为了有效增强容器文件系统隔离, 需要面对并解决以下挑战.

● 挑战 1: 安全性

首要目标是彻底解决容器与主机交互过程中与文件系统有关的漏洞, 希望从根本上消除可能存在的风险, 不仅能够防御现有攻击, 还能防止该类问题再次出现. 所以文件系统隔离方案应该能够隔离恶意符号链接, 并保护主机的文件描述符, 对跨命名空间文件访问进行全面、严格的权限检查, 守护主机容器的交互行为安全. FCFS 从文件 inode 层有效识别主机文件和容器文件, 加以访问控制策略, 能够有效保护主机文件, 防御容器和主机交互过程中的恶意行为.

● 挑战 2: 兼容性

Linux 内核设计的一大核心原则是保证兼容性, 这确保了软件与硬件之间的无缝协作, 消除了技术壁垒. 因此应该为用户应用程序提供透明的保护, 无需对容器工具或容器内的用户空间应用程序做任何修改. 保证在不影响用户层活动的情况下快速部署隔离增强机制, 同时保持系统原有功能的完整性和操作的流畅性. 目标是让安全增

强机制成为系统的一部分, 而非负担, 确保在提升安全性的同时, 系统的稳定性和兼容性不受损害. FCFS 能够在不改变用户空间应用程序行为并且不更改任何硬件的同时实现安全保护.

● 挑战 3: 性能

在无服务器计算场景下, 性能是衡量方案优劣的重要指标. 云服务提供商和用户均期望在享受容器带来的敏捷性与灵活性的同时, 无服务器功能能够实现瞬时部署, 且运行时的安全检查不应成为性能的绊脚石. 因此必须精心设计隔离机制, 确保在提供坚实隔离保障的同时, 不牺牲内核的响应速度和执行效率. 这意味着既要确保安全性, 也要兼顾性能, 使安全防护与高效运行并驾齐驱. FCFS 将渲染过程和访问控制集成到 Linux 路径查找过程中, 尽量减少冗余代码, 将性能开销降至最低.

下面详细介绍本文提出的容器文件系统隔离增强机制的设计和具体实现.

3.1 静态渲染

在容器初始化过程中, 需要识别 VFS 中容器文件系统的 inode 对象, 并区分容器文件系统和主机文件系统中的 inode 对象. 这种识别对后续的访问控制至关重要. 这一过程还必须考虑到容器与宿主机之间以及容器之间的文件共享机制, 因为许多现有的容器工具提供了“shared volume”接口 (如“docker run -v”), 允许容器之间或主机与容器之间共享文件. 共享卷允许跨容器甚至与宿主机进行数据交换, 打破了容器原本的封闭边界, 需要额外的安全考量.

因此在考虑实际的容器文件访问场景时, 需要设计一套多层次的安全策略, 定义多个安全级别. 具体来说, 可以将目标文件分为 3 类: 容器文件系统内的文件、宿主机文件系统上的文件以及在宿主机和多个容器间共享的文件. 每类文件对应不同的安全级别: 在上述威胁模型中 (第 2.1 节), 容器可能由攻击者操纵, 是不可信的, 所以必须对每个容器的文件系统采用最严格的访问控制策略, 将容器文件的风险等级视为最高, 即安全级别最低; 主机文件被认为是安全的, 具有最低的风险等级 (即最高的安全级别), 只能由主机进程访问; 作为容器文件系统的延伸, 共享卷允许跨容器或与宿主机的数据共享, 因此需要在运行时设置标签以区分有权限访问共享卷的主体, 可以认为其安全性介于容器和宿主机文件之间.

其次引入了安全域的概念, 它是一组受特定安全策略约束的 inode 集合, 属于容器文件系统或共享卷. 为了区分不同的安全域, 在现有的 PID 命名空间结构上进行了扩展——增加了一个枚举类型的变量“security_level”, 用于表示 inode 的所属安全级别. 该标志包含 3 个预定义的值: “strict (−2)”“normal(0)”和“shared(−1)”, 分别对应上述提到的 3 种文件类型. 此外还在 inode 结构中新增一个指向 PID 命名空间的指针, 以此来标识当前 inode 所属的命名空间. 这种渲染策略可以利用命名空间区分不同的容器, 判断文件的危险级别, 防止主机和容器之间的非法访问.

作为容器启动初始化过程的一部分, pivot_root 函数将容器进程的根目录切换到容器自身的 rootfs, 即容器的根文件系统, 这一流程标志着容器环境的创建完成. 切换 rootfs 成功后, 根据设计, 会从目标目录开始递归地遍历其下的所有 dentry. 系统会追踪每一个 dentry 中指针指向的 inode, 并对其进行标记. 如果遇到的 dentry 是一个挂载点, 即容器文件系统中被映射到其他文件系统的一个特殊目录, 那么, 不仅是这个挂载点, 连同其下整个文件系统的 inode 也需被标记. 容器内 inode 的安全级别标志被设定为“strict”. 这些 inode 的访问将受到最严格的控制, 任何试图访问它们的尝试都将受到严密的检查, 以确保容器的隔离性. 主机文件系统 inode 分配“normal”, 即被认为是安全的. 然而并非所有的 inode 都能如此简单地划归到容器内部或主机文件系统. 在处理共享卷时, 系统会采取一种更为灵活的策略. FCFS 将共享卷分配到一个专门为共享卷准备的临时 PID 命名空间中, 并将其安全级别标志设置为“shared”. 在特定条件下容器和宿主机可以访问共享卷, 但这种访问同样受到严格控制, 以防止未经授权的访问和数据泄露. 后文图 6 展示了 3 类目标文件的静态渲染结果.

3.2 路径查找渲染

在主机进程与容器之间的交互中, 仅依靠基于初始 PID 命名空间的 inode 标识不足以进行访问控制. 因为静态渲染虽能为文件系统中的 inode 附上标签, 却无法捕捉到路径查找过程中动态变化的情况. 路径查找可能跨越容器与宿主机之间的命名空间, 识别并评估这一过程中的安全风险尤为关键. 如果主机进程访问一个指向主机上

文件的容器内符号链接, 从结果来看是主机进程访问了主机文件 inode, 只设置静态渲染的情况下似乎是合法的, 但是实际上该主机进程被恶意符号链接诱导非法访问了容器外文件, 跨越了容器的安全边界. 因此需要将动态路径查找渲染与静态渲染相结合形成多阶段渲染策略, 确保能够及时识别并阻止任何进入低安全级别的命名空间的行为, 增强容器与主机之间的隔离安全.

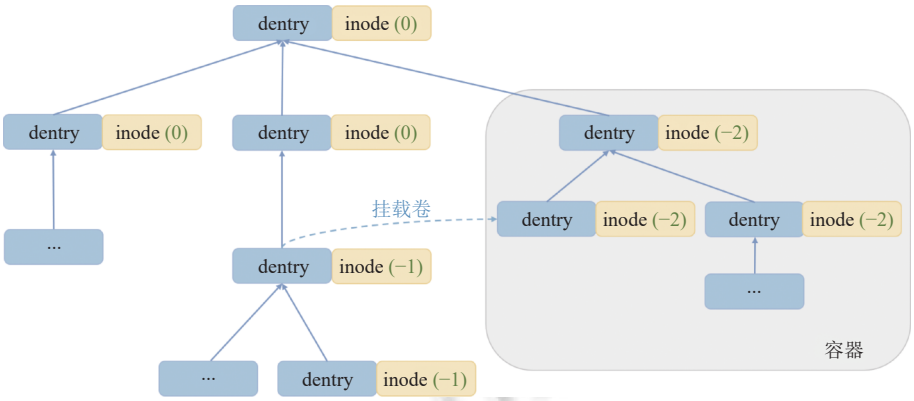


图 6 静态渲染

路径查找过程中, nameidata 临时结构的作用是记录查找的中间结果. 通过它可以找到当前路径分量, 以及当前文件系统的挂载点目录. 为了后续保存目前路径查找过程中最危险的安全级别, 在 nameidata 结构中添加了一个存储 PID 命名空间的成员. 路径查找从进程的初始命名空间开始, 因此初始化时将 nameidata 结构中的标签设置为当前进程的 PID 命名空间指针.

FCFS 通过拦截路径查找过程中调用的内核函数 path_to_nameidata 实现动态渲染. 该函数用于在每步查找时更新 nameidata 结构. 具体来说, 在每一步更新 nameidata 结构时, 将 nameidata 结构中的当前标签与刚找到的部件 inode 的标签进行比较, 并将 nameidata 中的标签更新为安全级别较低的那个标签. 通过这种方式, 即使路径查找返回到较高安全级别的命名空间, nameidata 结构中的标签也不会回退, 从而防止安全风险. 图 7 展示了容器进程访问共享卷中的资源的多重渲染 (静态渲染和动态渲染) 过程图, 绿色箭头表示找到一个部件并调用 path_to_nameidata 函数的过程, 箭头上面的数字表示当前 nameidata 结构中安全标签的级别, 图中访问主机 inode 后, 当前安全标签级别为 0, 进入容器访问容器文件系统 inode, 安全标签级别更新为最低——“strict”, 即−2. 随后沿着 dentry 树访问共享卷的文件 dentry 和 inode, 由于共享卷文件 inode 被标记为“normal”, 即−1, 大于−2, 所以 nameidata 结构中的安全级别不会更新, 其值为−2.

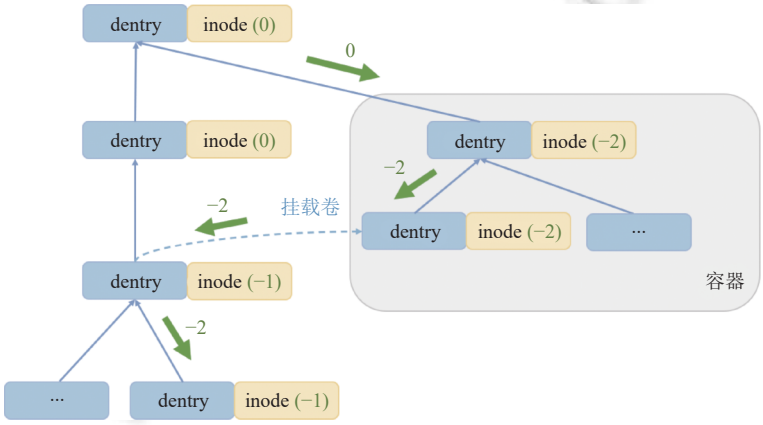


图 7 多重渲染过程

这种渲染在路径查找过程中是不可逆的, 以确保可以识别从可信环境到不可信环境的访问. 当主机进程进入容器文件系统时, nameidata 结构的标签会更新为容器的 PID 命名空间标签 (−2). 如果遇到指向主机文件系统的符号链接, 或者容器内恶意进程想通过主机泄露的文件描述符访问主机文件, 解析最后文件 inode 属于主机文件系统, nameidata 中的安全级别标签仍指示路径查找曾跨越容器命名空间, 加以访问控制策略, 可以禁止这种恶意访问.

3.3 控制访问策略

在完成了静态和动态渲染之后, 接下来的步骤是强制执行由标记确定的访问控制策略. 首先, 根据之前描述的威胁模型 (第 2.1 节), 容器被认为是不受信任的环境, 这意味着容器内的进程不能读取、写入或执行宿主机上的任何文件, 反之亦然. 此外, 共享卷作为一种特殊的文件系统资源, 既可以被容器访问, 也可以被宿主机访问, 需要特别注意. 为此定义了 3 种主要的访问控制策略, 并为了确保这 3 种策略的有效执行, 在内核中添加了放置在关键函数的特定检查点. 这些检查点主要集中在路径查找的 complete_walk 函数中, 该函数是路径查找过程的必要函数, 用于处理路径查找最终结果. 因此在这里实施访问控制可以有效地覆盖路径查找过程, 确保不会遗漏任何请求.

根据数据访问和控制流策略设计的原则, 制定了以下 3 条访问控制规则.

● 规则 1 (P1): 禁止低安全等级进程或跨低命名空间访问高安全等级文件.

该策略限制进程对文件的访问范围, 即低安全级别的进程不允许访问高安全等级的文件. 一旦容器中进程通过路径查找或文件描述符获取到的文件不在容器中, 即容器中进程非法访问了容器外的文件, 该过程将被禁止. 另外, 宿主机中进程解析文件路径时进入容器文件系统后将被标记低安全等级, 若在该解析过程中因容器文件系统中的符号链接再次回到宿主机文件系统中, 则禁止后续访问, 防止宿主机进程被恶意容器内符号链接误导而进行非法访问.

P1 的实现依赖于检查进程的标签是否与渲染阶段分配给文件 inode 的标签一致. 具体来说, 如果进程中的 PID 命名空间标签、nameidata 中的 PID 命名空间标签和最终获得的文件 inode 中的 PID 命名空间标签不同, 且如果进程标签的安全级别低于在查找过程结束时的最终 inode 的标签, 将阻止该次访问 P1, 可以有效防护链接欺骗和文件描述符泄露漏洞. 针对本文提出的 Podman 文件描述符漏洞, 容器中的恶意进程获得了主机泄露在容器环境中的主机文件描述符, 最终路径查找到主机上的文件, 文件 inode 标签为 0, 进程的 PID 命名空间标签为−2, 该次访问被禁止. 符号链接欺骗漏洞的防御也是类似原理.

● 规则 2 (P2): 允许主机进程访问共享卷.

该策略作为对 P1 的补充, 在容器进程访问挂载卷文件时, 因为容器进程的安全等级标签为“−2”, 低于挂载卷文件的安全等级标签“−1”, 该访问被禁止. P2 规定允许该类访问.

在实现时, 在 complete_walk 函数中设置了访问控制检查: 如果最终获得的文件 inode 安全标签为−1, 即为共享卷, 即使进程安全等级标签或 nameidata 中的 PID 命名空间标签低于该标签, 依然允许该次访问. P2 消除了 P1 过于严格的访问控制对正常容器使用过程中访问共享卷带来的问题.

● 规则 3 (P3): 禁止高安全级别的进程执行低安全等级的文件.

该策略关注的是控制流的转移, 规定从不受信任的容器发起的控制流转移必须停留在不受信任的区域内. 换句话说, 不允许任何处于较高安全级别的进程 (例如宿主机进程) 执行来自较低安全级别环境的文件 (例如容器内的文件). 例如, 一个宿主机进程使用容器内的可执行文件创建出一个新的进程且此进程将在宿主机上运行的情形下, 因为容器内的可执行文件对宿主机而言不可信, 若以宿主机的上下文执行此不可信的可执行文件, 其中包含的恶意负载将有权限访问宿主机中的资源.

为了实现 P3, 首先向 nameidata 结构中添加了一个访问模式标志, 这个标志在路径查找开始时初始化, 用于指示在路径查找完成后是否会执行目标文件. 当触发了需要进行路径查找的系统调用 (例如 sys_open 或 sys_execve) 时, 会检查调用该系统调用的参数并据此对访问模式标志进行初始化. 如果调用系统调用服务的进程的 PID 命名空间和目标文件 inode 中的 PID 命名空间标签不同, 那么就需要检查是否满足 P3 的要求. 如果进程具有执行权限 (即 nameidata 中的访问模式标志指示执行意图), 并且进程的安全级别标志大于 inode 中的安全级别, 将阻止对

inode 的访问. P3 实现了对执行文件行为的监控, 并确保不会发生在较高安全级别的环境中执行较低安全级别的文件的行为, 有效地限制了控制流的转移. P3 可以防御类似 CVE-2019-14271 漏洞的攻击: 进入容器环境的 docker-tar 进程加载了恶意的 nsswitch 动态库并准备执行时, 由于定位动态链接库之前需要进行路径查找, 在此过程得到最终需要执行的文件 inode 后发现进程的安全级别标志 (0) 大于 inode 中的安全级别 (−2), 终止该次执行流程.

3.4 优化兼容性

为了使 FCFS 在不改变用户空间应用程序行为的情况下实现对容器文件系统的安全隔离和访问控制, 进行了如下优化.

首先, 在容器启动过程中, 为了在不修改容器工具的情况下正确对容器文件系统对象进行标识, FCFS 通过截获容器启动时使用的系统调用 pivot_root, 并基于其参数 new_root 对容器文件系统进行标识. pivot_root 用于将进程的根目录切换到一个新的文件系统 (容器实例的文件系统), 它的 new_root 参数指示了容器文件系统的根目录. 当 pivot_root 被调用时, FCFS 会截获该系统调用并检查调用进程与其父进程是否在同一个挂载命名空间中. 如果不是, 则从 new_root 参数所指示的路径开始递归式遍历其下所有文件对象的 dentry (即遍历属于容器的文件对象), 使用当前进程的安全标签来标记 dentry 相关联的 inode. 如此可以确保所有相关的 inode 都被赋予了正确的标签, 不会对后续访问控制造成影响.

其次, 当符号链接解析需跨越多个安全域时, FCFS 会根据安全域的根目录修正符号链接的解析范围, 而不是仅基于访问控制规则阻止发生越界的符号链接解析过程, 保证应用程序功能的完整性. 为实现该功能, 在用于标识安全域的 PID 命名空间结构体中添加了一个名为 root_path 的成员, 用于存储各安全域的根目录. 当内核函数 trailing_symlink 解析符号链接时, FCFS 会通过 inode 中的标签查询当前符号链接所在安全域, 并使用该安全域的根目录重置符号链接解析的上下文, 以保证符号链接只能在其所属的安全范围内解析. 例如, 当宿主机进程尝试访问容器中符号链接且最终将该符号链接解析到宿主机路径时, 该过程会因符号链接解析越界而被访问控制规则所阻止. 优化后的 FCFS 则会在路径解析过程进入容器文件系统范围后, 将符号链接的解析起点 (即根目录) 重置为容器的根目录, 使容器中的符号链接永远在容器内部完成解析, 保证宿主机进程正常访问容器中文件的同时防止符号链接解析越界.

4 实验评估

本节将讨论在不同的基准测试下, FCFS 为 Linux 内核及应用层容器工具带来的性能开销. 实验结果表明, FCFS 可完全兼容现有的容器生态, 对 Linux 内核及容器工具的性能影响可忽略不计.

4.1 实验设置与参数

实验环境的硬件配置为 6 核 3.70 GHz AMD Ryzen 5 7500F CPU 和 16 GB RAM 的虚拟机, 操作系统版本为 Ubuntu 18.04, 内核版本为 v5.6.0.

4.2 有效性分析

首先评估了 FCFS 的有效性, 即对上述两类漏洞的抵御能力. 为验证 FCFS 对攻击行为的响应结果, 本文选取了 Docker、Podman、Containerd、runC 以及 Kubernets 等 5 个主流容器工具中的 13 个安全漏洞以及本文在第 4.3 节提到的“podman top”漏洞, 这些漏洞具有较高代表性, 涵盖了本文研究的两类漏洞类型. 通过实验复现所有漏洞的攻击过程, 并开启系统日志以观测 FCFS 中访问控制策略的激活情况, 并与 Patrol[8]和 vKernel[41]进行了对比, 实验结果如后文表 1 所示. 标记“√”表示该漏洞能被发现或防御, 可以看到所有漏洞利用过程中所有越界访问行为均被 FCFS 捕获并被访问控制策略拦截, 有效抵御了各类路径解析错误和文件描述符泄露漏洞.

4.3 兼容性及性能评估

实验首先使用两种标准测试集 SPEC 2017[42,43]和 Filebench[44]对 FCFS 进行性能与兼容性的评估, 实验结果分别如图 8 和图 9 所示.

表 1 对抗漏洞的有效结果对比

漏洞类型	漏洞	Patrol	vKernel	FCFS
路径解析错误漏洞	CVE-2017-1002101	√	√	√
	CVE-2018-15664	√	√	√
	CVE-2019-10152	√	√	√
	CVE-2019-14271	√	—	√
	CVE-2019-18466	√	√	√
	CVE-2021-25741	√	√	√
	CVE-2021-30465	√	√	√
	CVE-2022-0492	√	—	√
	CVE-2022-1227	√	—	√
	CVE-2022-23648	√	√	√
	CVE-2023-0778	√	√	√
文件描述符泄露漏洞	CVE-2019-5736	√	√	√
	podman top漏洞	—	√	√
	CVE-2024-21626	—	√	√

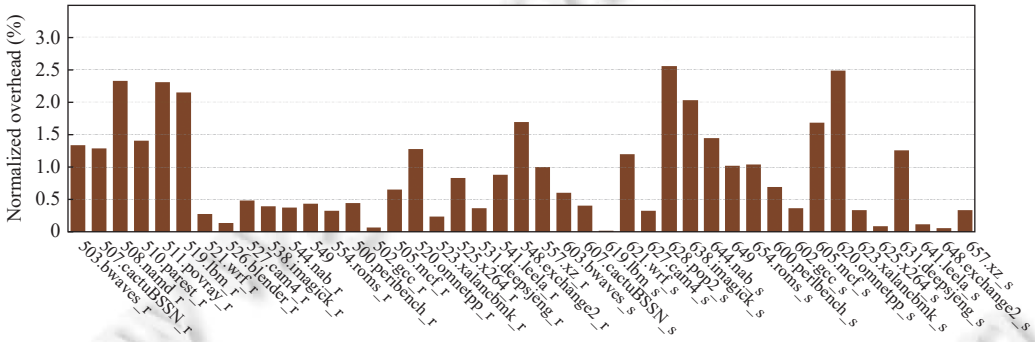


图 8 SPEC 2017 开销

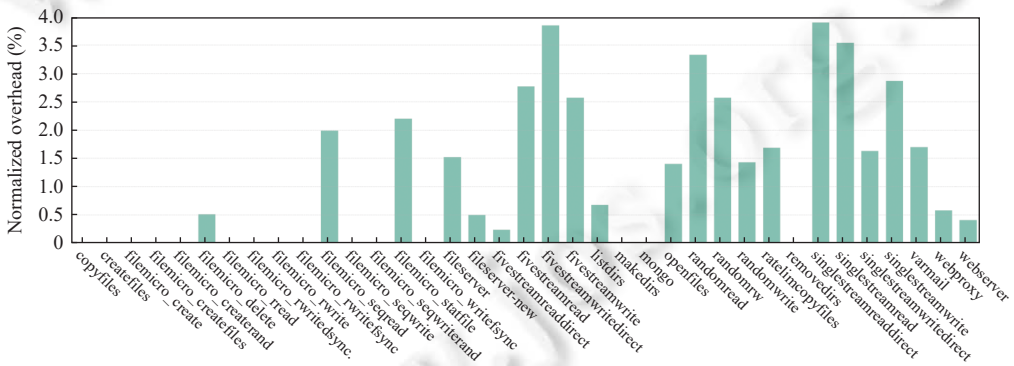


图 9 Filebench 开销

SPEC 2017 测试集中的 43 个基准测试包含了一系列实际应用程序和算法, 可模拟各种计算场景以全面评估系统的计算能力. 实验结果显示, 集成了 FCFS 的 Linux 内核可成功运行所有基准测试, 表明 FCFS 并未对系统兼容性产生影响. 同时, 与未集成 FCFS 的内核相比, FCFS 为 SPEC 2017 中测试样例引入的最大开销不超过 2.6%, 其中 31 个基准测试的开销低于 1.5%.

Filebench 测试集中包含 37 个与文件系统和存储系统相关的测试基准, 包括文件的创建、读取、写入、删除

等操作. 与原生 Linux 内核相比, FCFS 在 Filebench 测试集中引入的最大开销不超过 4.0%, 其中 19 个基准测试的开销低于 2%, 有 15 个基准测试的开销趋近于 0.

其次, 实验测试了一组与路径解析直接相关的系统调用 (open、stat、chmod 和 rename)[45], 并量化了 FCFS 为其引入的开销. 图 10 显示了原始内核和集成了 FCFS 的内核中使用这些系统调用处理不同文件路径时的时间开销. 实验中使用的文件路径具有不同的长度 (“1-comp”“2-comp”和“4-comp”分别表示路径/X, /X/Y 和/X/Y/Z/A), 或包含符号链接 (“1-symlink”表示符号链接/C→/X/Y/Z/C) 及父目录 (“1-dot”表示路径/X/Y/…/C). 实验结果显示 FCFS 为此类系统调用引入的开销小于 150 ns, 且不超过原始系统调用观察到的延迟的 6.9%, 此类开销主要源于 FCFS 在路径查找过程中实施的访问控制.

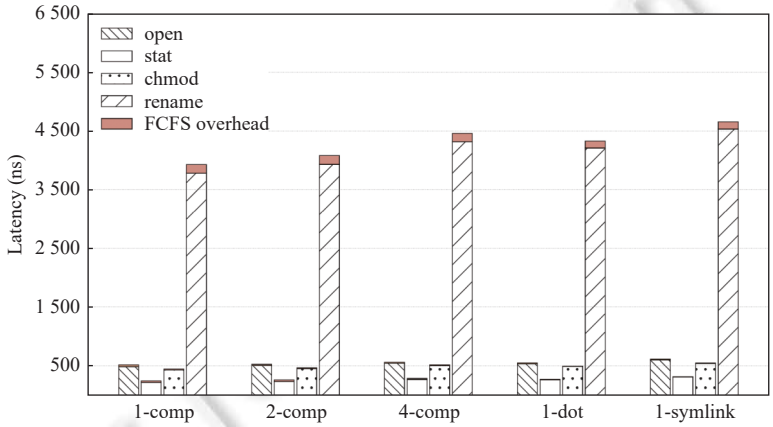


图 10 系统调用开销

为了进一步评估路径查找在复杂符号链接结构中的延迟, 还开展了递归符号链接实验, 构建了深度为 5 层、10 层、20 层和 40 层的递归符号链接路径, 并运用 open、stat、chmod 和 rename 这 4 个系统调用进行测试. 实验结果图 11(a) 显示, 随着递归深度的增加, 路径查找的延迟呈现出明显的上升趋势. 不过, 即便在如此复杂的递归符号链接结构下, 这些开销也小于 120 ns, 修改后内核的延迟也未超过原始内核延迟的 6.7%, 表明其在处理复杂符号链接时仍具有较高的效率. 此外还开展了多层挂载实验, 搭建了深度为 5 层、10 层、20 层和 40 层的多层挂载路径, 每个挂载点均挂载了独立的文件系统, 并在最深的挂载层创建了测试文件. 实验结果图 11(b) 表明, 多层挂载结构对路径查找延迟的影响较为显著. 随着挂载深度的增加, 路径查找需要逐层解析挂载点, 导致延迟逐渐上升. 在 5 层挂载时, 延迟增加幅度相对较小, 而当挂载深度达到 40 层时延迟增长更为突出. 尽管如此, 这些开销也小于 160 ns, 修改后内核的延迟在极端情况下也未超过原始内核延迟的 6.7%, 说明其能够有效应对多层挂载路径查找的复杂性, 同时保持较低的延迟开销.

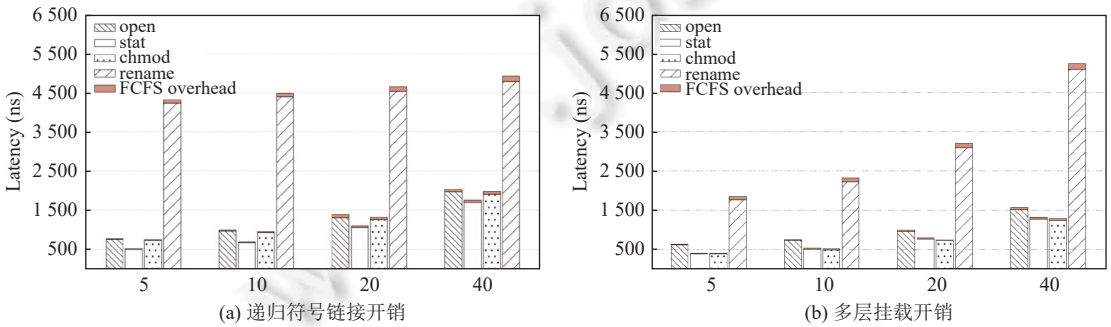


图 11 递归符号链接及多层挂载开销

4.4 对容器工具的性能影响

为消除各类路径解析错误和文件描述符泄露漏洞, 容器工具中存在大量漏洞补丁. 实验评估了 FCFS 为应用层中容器工具引入的开销, 并与漏洞补丁带来的开销进行了比较. 实验选取了 3 款主流的容器工具 Docker、Podman、Kubernetes 进行评估, 并使用其最早存在漏洞的版本 (docker-ce v18.03.1、Kubernetes v1.7.13 和 Podman v1.5.1) 作为测试对象.

根据 FCFS 的设计, 容器启动过程中 FCFS 需对容器中文件对象进行标记, 因此会产生一定性能开销, 延长容器的启动时间. 此外, CVE-2017-1002101、CVE-2019-5736、CVE-2021-30465、CVE-2021-25741 和 CVE-2022-23648 漏洞的补丁也会影响容器的启动时间. 为量化此类性能损耗, 实验选取了 DockerHub[45]中 120 个最流行的镜像, 并测量了不同容器工具使用这些镜像启动容器实例的时间. 图 12 显示了不同容器工具启动容器实例时的性能开销分布, 其中箱线图的上限与下限分别设置为 5% 和 95%. 对于所有这些容器工具, FCFS 引入的管理费用的中位数小于 3.8%, 第 95 个百分位数低于 7%. 其中 Docker 的开销最低, 中位数为 3.68%, 第 95 百分位为 5.80%. 相比之下, 由漏洞引起的开销补丁更高, 中位数约为 6.52%.

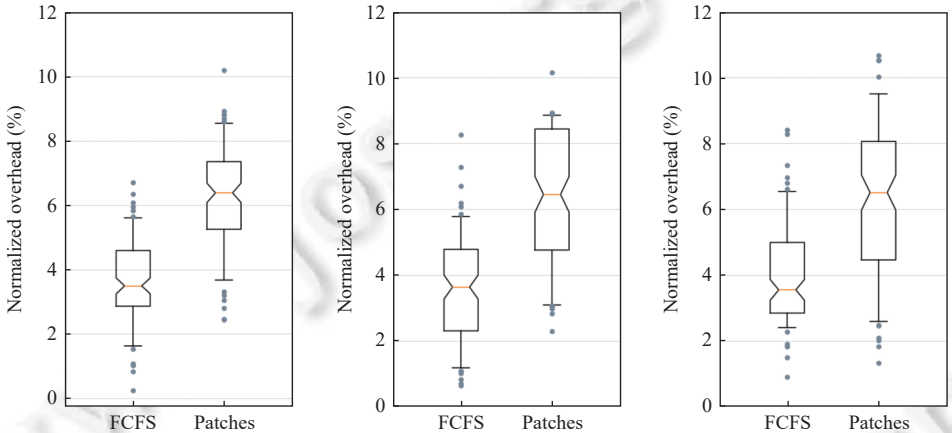


图 12 Docker、Podman 和 Kubernetes 性能下降分布图

此外, CVE-2018-15664, CVE-2019-14271, CVE-2019-10152 和 CVE-2019-18466 等漏洞均存在于 Docker 和 Podman 的“copy”功能中, 因此该功能中存在数个补丁. 为评估 Docker 和 Podman 在 FCFS 环境下和使用补丁情况下执行“copy”功能的性能损耗, 实验量化并比较了“docker cp”和“podman cp”将不同大小的文件 (1 MB–1 GB) 从容器复制到主机的时间, 实验结果如后文图 13 所示. 其中, 图 13(a) 显示了 FCFS 在“docker cp”过程中引入的开销最大不超过 4.01%, 而漏洞补丁带来的最小开销为 85.96% 并随着复制文件的增大而增长. 图 13(b) 显示了 FCFS 对于“podman cp”过程引入的开销都在 1.51% 以下. 相比之下, 漏洞补丁引入的开销则从 19.83% 到 74.82% 不等, 远高于 FCFS 引入的开销.

5 相关工作

容器技术的出现深刻地改变了在云中开发和部署分布式应用程序的生态系统. 但是由于容器隔离机制的不完整, 导致一些安全问题的出现, 目前已经有许多研究指出了容器隔离机制的脆弱性. Gao 等人[46]指出由于现有命名空间缺少上下文检查以及某些 Linux 子系统没有完整的命名空间, 未隔离的伪文件系统可能是泄露宿主机信息的通道, 攻击者可以通过这些信息分析容器同驻情况以及监控宿主机资源使用情况, 发起一种新型的电力攻击. Gao 等人[47]在 2019 年又提出恶意容器可以打破 cgroup 的资源控制, 极大地耗尽主机资源并发起多种攻击. Yang 等人[48]发现攻击者能够破坏 PID 命名空间的隔离. 在不利用任何内核漏洞的情况下, 非特权容器可以很轻易地耗尽共享

的内核变量和数据结构实例, 从而导致对其他容器的 DoS 攻击. 他们还在多个共享内核容器环境中进行了攻击实验, 结果表明所有环境都容易受到抽象资源攻击.

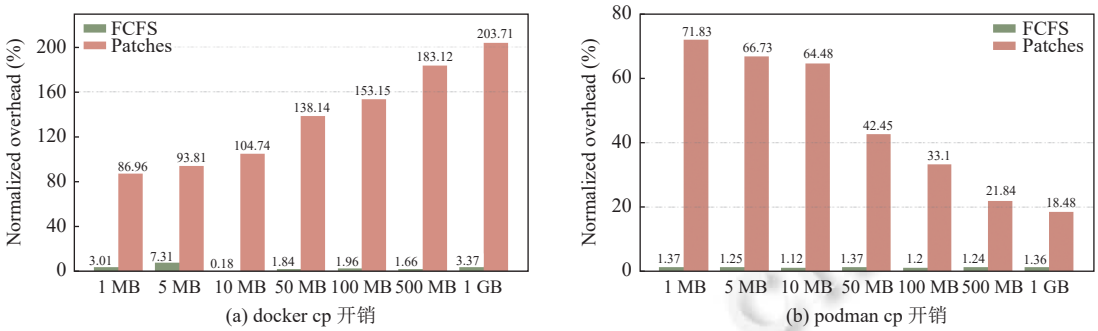


图 13 docker cp 和 podman cp 的开销与 FCFS 开销对比

Jian 等人[49]研究了 Docker 的容器逃逸攻击并试图在进程执行过程中基于命名空间状态检查来解决逃逸攻击. 他们认为, 攻击者程序获得了 root 访问权限后会尝试更改命名空间, 因此提出的解决方案会检测命名空间状态, 并标记可能表明容器遭到入侵的任何更改, 这有助于检测异常并进一步防止逃逸攻击, 但是并不能提高容器隔离的完整性. Wang 等人[50]从页面缓存隔离入手, 提出将页面缓存分为私有缓存和共享缓存两种, 并利用技术精准控制两种页面缓存, 增强容器的内存隔离能力. Abbas 等人[51]提出的跨命名空间检测方法可以识别大多数已经存在的符号链接欺骗漏洞, 但是在检测到威胁后 PACED 不会采用任何相关的防御策略来保护系统, 并且不能从根本上消除此类恶意攻击. Li 等人[8]尝试增强容器文件系统的隔离, 但是只涉及 dentry 层, 当攻击者通过主机文件描述符访问文件时, 不需要经过 dentry 安全检查, 不会触发 Patrol 的访问控制策略, 也即无法防御住主机文件描述符泄露漏洞. vKernel[41]根据 AppArmor 配置文件来标识文件 inode 是否是敏感文件, 在容器访问文件时拦截通用的权限检查函数并检查访问对象是否敏感文件. 这种方法只有在容器访问 AppArmor 配置中的文件时才能发挥作用, 无法防御如 CVE-2019-14271 这类原理是宿主机进程被欺骗执行了容器内的恶意文件的漏洞. 而 FCFS 能够精准判断出文件属于主机还是容器, 利用 inode 标签和访问控制策略杜绝了容器越权访问主机文件、主机进程被欺骗执行恶意容器文件等漏洞的产生, 对跨命名空间文件访问进行全面、严格的权限检查, 守护主机容器的交互行为安全, 并且不需要修改任何硬件.

容器社区还提出了安全容器的方案来隔离内核漏洞, 比如 Kata 容器[52]和 gVisor[53]. Kata 容器通过硬件虚拟化来达到容器隔离, 它在虚拟机实例内运行容器实例. 虚拟机保证了强大的隔离性, 同时对客户机内核进行了修剪, 以减少性能开销. 但是 Kata 容器的隔离仍然存在问题. 它不强制执行设备 cgroup, 攻击者有机会访问客户机虚拟机上的 /dev 文件[54]. Yang 等人[55]提出 Kata 运行时不检查共享卷中挂载点的有效性, 容器内的攻击者可以利用多个漏洞实施攻击, 例如伪装成 Kata-agent 并在挂载点创建符号链接, 将新容器的根文件系统挂载到符号链接指向的主机上的任何位置. 尽管 gVisor 容器通常采用的虚拟文件系统无法直接从主机访问, 但容器和主机之间的共享卷驻留在主机文件系统上. 当容器工具利用复制功能将符号链接文件从 gVisor 容器复制给其用户时, 该卷中存在的恶意符号链接将在主机上解析, 从而可能导致主机文件泄露. 因此为了守护主机和容器交互的安全性, 需要设计和实现更稳健的文件系统隔离机制.

6 总结及未来工作

容器依赖于命名空间和 cgroup 来实现隔离, 但容器与宿主机之间的文件系统隔离仍然存在安全漏洞, 例如路径解析错误漏洞和文件描述符泄露漏洞. 本文介绍了一种文件描述符泄露漏洞的攻击过程, 并分析了上述两类漏洞的修复手段, 认为在容器工具层面无法根本性地解决问题. 为此提出了一种基于内核的容器文件系统隔离机制 FCFS, 将文件系统隔离拓展到 inode 级别, 实现更细粒度的隔离, 并不需要修改任何硬件. 此外还通过测试验证了

FCFS 的有效性和高效性, 结果表明 FCFS 可以完全阻止这两类漏洞, 同时引入的开销几乎可以忽略不计. 由于 FCFS 未改变 Linux 内核原有的系统调用语义与文件系统行为, 开源社区和企业用户可以直接将其应用于主流 Linux 发行版 (如 CentOS、Ubuntu、Debian 等), 无需对上层应用或容器引擎做出适配性修改, 具有较高的可移植性和兼容性. FCFS 不仅根本性解决了路径解析错误漏洞和文件描述符泄露漏洞, 确保了准确和全面的主机和容器交互控制, 同时保持了良好的兼容性, 这项工作为容器技术的安全应用提供了重要参考. 由于不同操作系统在内核架构、文件系统管理和进程调度等方面存在差异, 因此未来的工作可能是针对鸿蒙操作系统 (HarmonyOS) 等新型操作系统的特性对 FCFS 进行适配性设计和优化, 并且探索更复杂的权限场景下 FCFS 的优化方案, 尝试进一步结合基于角色的访问控制 (role-based access control)[56]等机制, 以支持多租户使用场景.

References

[1] Linux. namespaces(7)—Linux manual page. 2025. https://www.man7.org/linux/man-pages/man7/namespaces.7.html

[2] Linux. cgroup(7)—Linux manual page. 2025. https://man7.org/linux/man-pages/man7/cgroups.7.html

[3] Linux. mountpoint(1)—Linux manual page. 2025. https://man7.org/linux/man-pages/man1/mountpoint.1.html

[4] Docker. Docker: Accelerated containerized application development. 2025. https://www.docker.com/

[5] Podman. The best free & open source container tools. 2025. https://podman.io/

[6] Containerd. containerd: An industry-standard container runtime with an emphasis on simplicity, robustness and portability. 2025. https://containerd.io/

[7] Kubernetes. Production-grade container orchestration. 2025. https://kubernetes.io/

[8] Li Z, Liu WJ, Wang XF, Yuan B, Tian HL, Jin H, Yan SM. Lost along the way: Understanding and mitigating path-Misresolution threats to container isolation. In: Proc. of the 2023 ACM SIGSAC Conf. on Computer and Communications Security. Copenhagen: ACM, 2023. 3063–3077. [doi: 10.1145/3576915.3623154]

[9] CVE. CVE-2017-1002101. 2017. https://www.cve.org/CVERecord?id=CVE-2017-1002101

[10] CVE. CVE-2021-25741. 2021. https://www.cve.org/CVERecord?id=CVE-2021-25741

[11] Linux. mount_namespaces(7)—Linux manual page. 2025. https://man7.org/linux/man-pages/man7/mount_namespaces.7.html

[12] Gitee. FCFS. 2025. https://gitee.com/ClownandBox/FCFS

[13] Zhong BS, Zhang YC, Zhou MJ. Analysis about virtual file system of Linux. Computer and Modernization, 2010(9): 76–78 (in Chinese with English abstract). [doi: 10.3969/j.issn.1006-2475.2010.09.021]

[14] Song NY, Kim H, Han H, Yeom HY. Efficient dentry lookup with backward finding mechanism. In: Proc. of the 2018 Int'l Conf. on High Performance Computing in Asia-Pacific Region. Chiyoda: ACM, 2018. 299–308. [doi: 10.1145/3149457.3149468]

[15] Linux. seccomp(2)—Linux manual page. 2025. https://man7.org/linux/man-pages/man2/seccomp.2.html

[16] AppArmor. Linux kernel security module. 2025. https://apparmor.net/

[17] Linux. capabilities(7)—Linux manual page. 2025. https://man7.org/linux/man-pages/man7/capabilities.7.html

[18] Dua R, Kohli V, Patil S, Patil S. Performance analysis of Union and CoW file systems with Docker. In: Proc. of the 2016 Int'l Conf. on Computing, Analytics and Security Trends. Pune: IEEE, 2016. 550–555. [doi: 10.1109/CAST.2016.7915029]

[19] GitHub, Inc. opencontainers/runc. 2025. https://github.com/opencontainers/runc

[20] Azure. Add continuous integration to your container builds. 2025. https://learn.microsoft.com/en-us/azure-sphere/app-development/continuous-integration

[21] CVE. CVE-2019-10152. 2019. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-10152

[22] CVE. CVE-2019-18466. 2019. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-18466

[23] CVE. CVE-2021-30465. 2021. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-30465

[24] CVE. CVE-2019-19921. 2019. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-19921

[25] CVE. CVE-2022-23648. 2022. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-23648

[26] CVE. CVE-2022-0492. 2022. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-0492

[27] Bratus S, D'Cunha N, Sparks E, Smith SW. TOCTOU, traps, and trusted computing. In: Lipp P, Sadeghi AR, Koch KM, eds. Trusted Computing—Challenges and Applications. Berlin: Springer, 2008. 14–32 [doi: 10.1007/978-3-540-68979-9_2]

[28] CVE. CVE-2018-15664. 2018. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-15664

[29] CVE. CVE-2023-0778. 2023. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-0778

[30] GitHub, Inc. daemon: Archive: Pause containers before doing filesystem operations. 2025. https://github.com/moby/moby/pull/39252

[31] Linux. mount(8)—Linux manual page. 2025. https://man7.org/linux/man-pages/man8/mount.8.html

[32] Dockerdocs. Docker container cp. 2025. https://docs.docker.com/reference/cli/docker/container/cp/

[33] CVE. CVE-2019-14271. 2019. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-14271

[34] UTC. Auto-reloading for /etc/nsswitch.conf. 2025. https://sourceware.org/bugzilla/show_bug.cgi?id=12459
[35] CVE. CVE-2022-1227. 2022. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-1227
[36] Mouw E. Linux kernel procfs guide. 2001. https://www23.big.or.jp/~sian/linux/DocBook/procfs-guide/index.html
[37] Avrahami Y. Breaking out of Docker via runC—Explaining CVE-2019-5736. 2019. https://unit42.paloaltonetworks.com/breaking-docker-via-runc-explaining-cve-2019-5736/
[38] CVE. CVE-2024-21626. 2024. https://www.cve.org/CVERecord?id=CVE-2024-21626
[39] podman-top—Display the running processes of a container. 2019. https://docs.podman.io/en/latest/markdown/podman-top.1.html
[40] Linux. cgroup-v1: Require capabilities to set release_agent—kernel/git/torvalds/linux. git—Linux kernel source tree. 2022. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=24f6008564183aa120d07c03d9289519c2fe02af
[41] Huang H, Wang HL, Rao J, Wu S, Fan H, Yu C, Jin H, Suo K, Pan LS. vKernel: Enhancing container isolation via private code and data. IEEE Trans. on Computers, 2024, 73(7): 1711–1723. [doi: 10.1109/TC.2024.3383988]
[42] SPEC. http://www.spec.org/index.html
[43] Henning JL. SPEC CPU2006 benchmark descriptions. ACM SIGARCH Computer Architecture News, 2006, 34(4): 1–17. [doi: 10.1145/1186736.1186737]
[44] Sun Microsystems. FileBench. 2004. http://www.nfsv4bat.org/Documents/nasconf/2004/filebench.pdf
[45] Tsai CC, Zhan Y, Reddy J, Jiao YZ, Zhang T, Porter DE. How to get more value from your file system directory cache. In: Proc. of the 25th Symp. on Operating Systems Principles. Monterey: ACM, 2015. 441–456. [doi: 10.1145/2815400.2815405]
[46] Gao X, Gu ZS, Kayaalp M, Pendarakis D, Wang HN. ContainerLeaks: Emerging security threats of information leakages in container clouds, In: Proc. of the 47th Annual IEEE/IFIP Int'l Conf. on Dependable Systems and Networks. Denver: IEEE, 2017. 237–248. [doi: 10.1109/DSN.2017.49]
[47] Gao X, Gu ZS, Li ZF, Jamjoom H, Wang C. Houdini's escape: Breaking the resource rein of Linux control groups. In: Proc. of the 2019 ACM SIGSAC Conf. on Computer and Communications Security. London: ACM, 2019. 1073–1086. [doi: 10.1145/3319535.3354227]
[48] Yang NZ, Shen WB, Li JK, Yang YT, Lu KJ, Xiao JT, Zhou TY, Qin CG, Yu W, Ma JF, Ren K. Demons in the shared kernel: Abstract resource attacks against OS-level virtualization. In: Proc. of the 2021 ACM SIGSAC Conf. on Computer and Communications Security. ACM, 2021. 764–778. [doi: 10.1145/3460120.3484744]
[49] Jian ZQ, Chen L. A defense method against docker escape attack. In: Proc. of the 2017 Int'l Conf. on Cryptography, Security and Privacy. Wuhan: ACM, 2017. 142–146. [doi: 10.1145/3058060.3058085]
[50] Wang K, Wu S, Li SB, Huang Z, Fan H, Yu C, Jin H. Precise control of page cache for containers. Frontiers of Computer Science, 2024, 18(2): 182102. [doi: 10.1007/s11704-022-2455-0]
[51] Abbas M, Khan S, Monum A, Zaffar F, Tahir R, Eyers D, Irshad H, Gehani A, Yegneswaran V, Pasquier T. PACED: Provenance-based automated container escape detection. In: Proc. of the 2022 IEEE Int'l Conf. on Cloud Engineering. Pacific Grove: IEEE, 2022. 261–272. [doi: 10.1109/IC2E55432.2022.00035]
[52] GitHub, Inc. Kata containers. 2025. https://github.com/kata-containers
[53] GitHub, Inc. Gvisor. 2025. https://github.com/google/gvisor
[54] CVE. CVE-2022-2023. 2022. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-2023
[55] Yang YT, Shen WB, Ruan BN, Liu WM, Ren K. Security challenges in the container cloud. In: Proc. of the 3rd IEEE Int'l Conf. on Trust, Privacy and Security in Intelligent Systems and Applications. Atlanta: IEEE, 2021. 137–145. [doi: 10.1109/TPSISA52974.2021.00016]
[56] Sandhu RS, Coyne EJ, Feinstein HL, Youman CE. Role-based access control models. Computer, 1996, 29(2): 38–47. [doi: 10.1109/2.485845]

附中文参考文献

[13] 钟柏松, 张宇成, 周明建. Linux 虚拟文件系统分析. 计算机与现代化, 2010(9): 76–78. [doi: 10.3969/j.issn.1006-2475.2010.09.021]

作者简介

李志, 博士, 副教授, CCF 专业会员, 主要研究领域为云原生安全, 操作系统与虚拟化安全, 云原生黑产分析, AI Agent 安全.
夏书婷, 硕士生, 主要研究领域为容器安全.
李圣杰, 硕士生, 主要研究领域为云原生安全.
刘维杰, 博士, 副教授, CCF 专业会员, 主要研究领域为系统安全, 虚拟化, 程序分析及其在大模型安全领域的应用.
王振辰, 硕士生, 主要研究领域为系统安全.
金海, 博士, 教授, 博士生导师, CCF 会士, 主要研究领域为计算机系统结构, 虚拟化技术, 集群计算, 网格计算, 并行与分布式计算, 对等计算, 普适计算, 语义网, 存储与安全.