基于异构观测链的容器逃逸检测方法

张云涛[1]，方滨兴[1,2]，杜春来[3]，王忠儒[4]，崔志坚[3]，宋首友[1,5]

（1. 北京邮电大学网络空间安全学院，北京 100876；2. 广州大学网络空间先进技术研究院，广东 广州 510006；
3. 北方工业大学信息学院，北京 100144；4. 中国网络空间研究院信息化研究所，北京 100010；
5. 北京丁牛科技有限公司，北京 100081）

摘　要：针对现有容器逃逸检测技术漏报率较高的问题，提出一种异构观测的实时检测方法。首先对利用内核漏洞的容器逃逸行为建模，选取进程的关键属性作为观测点，提出以“权限提升”为检测标准的异构观测方法；然后利用内核模块实时捕获进程的属性信息，构建进程起源图，并通过容器内外进程边界识别技术缩小起源图规模；最后基于进程属性信息构建异构观测链，实现原型系统 HOC-Detector。实验结果表明，HOC-Detector 可以成功检测测试数据集中利用内核漏洞的容器逃逸，并且运行时增加的总体开销低于 0.8%。

关键词：容器逃逸；内核漏洞；开放起源模型；异构观测链
中图分类号：TP393.081
文献标志码：A
DOI: 10.11959/j.issn.1000−436x.2023008

Container escape detection method based on heterogeneous observation chain

ZHANG Yuntao[1], FANG Binxing[1,2], DU Chunlai[3], WANG Zhongru[4], CUI Zhijian[3], SONG Shouyou[1,5]

1. School of Cyberspace Security, Beijing University of Posts and Telecommunications, Beijing 100876, China
2. Cyberspace Institute Advanced Technology, Guangzhou University, Guangzhou 510006, China
3. School of Information Science and Technology, North China University of Technology, Beijing 100144, China
4. Chinese Academy of Cyberspace Studies, Institute of Information Technology, Beijing 100010, China
5. Beijing DigApis Technology Co., Ltd., Beijing 100081, China

Abstract: Aiming at the problem of high false negative rate in container escape detection technologies, a real-time detecting method of heterogeneous observation was proposed. Firstly, the container escape behavior utilizing kernel vulnerabilities was modeled, and the critical attributes of the process were selected as observation points. A heterogeneous observation method was proposed with “privilege escalation” as the detection criterion. Secondly, the kernel module was adopted to capture the attribute information of the process in real time, and the process provenance graph was constructed. The scale of the provenance graph was reduced through container boundary identification technology. Finally, a heterogeneous observation chain was built based on the process attribute information, and the prototype system HOC-Detector was implemented. The experiments show that HOC-Detector can successfully detect all container escapes using kernel vulnerabilities in the test dataset, and the increased runtime overhead is less than 0.8%.

Keywords: container escape, kernel vulnerability, open provenance model, heterogeneous observation chain

---

收稿日期：2022−08−16；修回日期：2022−11−11
通信作者：王忠儒，wangzhongru@cac.gov.cn
基金项目：国家自然科学基金资助项目（No.62172006）；广东省重点研发计划基金资助项目（No.2019B010136003）；国家重点研发计划基金资助项目（No.2019YFA0706404）
Foundation Items: The National Natural Science Foundation of China (No.62172006), The Key Research and Development Program of Guangdong Province (No.2019B010136003), The National Key Research and Development Program of China (No.2019YFA0706404)

0 引言

在数字经济背景下，云计算被写入《中华人民共和国国民经济和社会发展第十四个五年规划和 2035 年远景目标纲要》。中国信息通信研究院数据显示，预计到 2024 年，我国云市场规模将接近 7 800 亿元。以容器为代表的云原生技术蓬勃发展，为云计算数字化赋能提供了重要推动力，并为软件开发和系统运维带来了颠覆性的变革。容器技术[1]是一种轻量级的虚拟化技术，可以实现应用程序在虚拟化隔离环境中的稳定运行。在传统操作系统级别的虚拟化技术中，每个实例都需要运行客户端操作系统的完整副本，因此造成了大量的硬件模拟开销[2]。容器是进程级的虚拟化，不需要模拟硬件，相比于传统虚拟化技术少了很多开销，从而可以轻量高效地运行应用程序。容器编排通过微服务体系结构模式来实现业务流程自动化，是大规模容器应用的重要工具。容器编排工具（如 Kubernetets（K8S）[3]和 Docker Swarm[4]）的出现极大地方便了容器化应用程序的部署、扩展和管理。目前，容器技术已经广泛应用于大型互联网公司的生产环境和云服务供应商，如 Amazon Fargate、Microsoft Azure Kubernetes 等。

容器技术虽然已经在实际生产环境被广泛使用，但存在较多潜在的安全威胁，如容器逃逸、资源隔离失效、镜像安全威胁、横向移动攻击、拒绝服务攻击、运行环境未加固等[5]。其中，容器逃逸直接影响了承载容器的底层基础设施的安全性和可用性，进而对生产环境造成了安全隐患。具体来说，容器逃逸[6]是指攻击者利用程序、系统的漏洞或缺陷，突破容器与宿主机之间的隔离机制，获得在宿主机上的命令执行能力。目前，导致容器逃逸的主要因素包括配置不当、应用程序漏洞和系统内核漏洞。相比于前两者，系统内核漏洞导致的逃逸危害更大、威胁更广。这是因为容器与宿主机共用一个内核，攻击者可以利用内核漏洞进行攻击，从而突破隔离“逃逸”至宿主机，对宿主机和其他容器造成了极大的安全威胁。因此，如何检测利用内核漏洞的容器逃逸具有重要的研究意义。由于 Docker 是当前使用范围最广的容器引擎之一，本文专注于检测 Linux 平台下利用内核漏洞实现的 Docker 容器逃逸。

目前，Linux 平台下对利用内核漏洞实现容器逃逸的检测方法主要是基于运行时的异常检测[7-8]。Salamero[7]使用 eBPF 实现了一个进程异常行为检测工具 Falco，可通过连续监视 Linux 内核中的系统调用来捕捉异常行为。但是 Falco 主要基于预先设定的规则来判断异常行为，不能有效防御未知漏洞的攻击。Jian 等[8]提出基于容器内进程所属命名空间（namespaces）[9]的变化来检测容器逃逸行为。但该方案存在 2 个缺陷：检测滞后性和检测失效性。

为了解决现有容器逃逸检测中漏报率较高的问题，本文提出了一种异构观测的实时检测方法，并实现了基于异构观测链的容器逃逸检测原型系统，简称 HOC-Detector。首先，通过大量复现利用内核漏洞实现的容器逃逸攻击案例，对容器逃逸流程进行建模，将相应的攻击案例划分为直接逃逸和间接逃逸。容器逃逸后获得 root 权限的逃逸进程是容器内进程的子进程则为直接逃逸；反之则为间接逃逸。然后，对已有利用内核漏洞的容器逃逸行为进行观测，并总结、提炼出一系列观测点，包括进程的用户 ID（uid）、组 ID（gid）、能力（capabilities）[10]、根目录（root directory）以及 namespaces 等。同时提出容器逃逸是由一系列以“权限提升”为目的的攻击手段组合而成的。最后，基于捕获的进程行为生成容器进程的起源图，结合观测点构建容器进程逃逸行为的异构观测链，以是否出现“权限提升”为检测标准，对容器进程全生命周期进行观测，实现对利用内核漏洞实施容器逃逸的检测。本文的主要贡献总结如下。

1) 首先调研并复现了 10 个利用内核漏洞（CVE-2016-5195、CVE-2017-7308、CVE-2017-11176、CVE-2017-18344、CVE-2017-1000112、CVE-2018-18955、CVE-2020-14386、CVE-2021-22555、CVE-2022- 0185、CVE-2022-0847）的容器逃逸案例，对利用内核漏洞的容器逃逸流程进行建模，提炼出容器逃逸行为的主要观测点和基于“权限提升”的容器逃逸检测标准。

2) 利用 Linux 内核模块实现了对容器进程操作行为的捕获和进程属性信息的提取，进一步生成了进程起源图。

3) 提出异构观测的检测方法，基于容器进程起源图和观测点构建异构观测链，从多个角度对容器进程所执行的操作进行全生命周期的异构观测，检测是否有权限提升的情况。

4) 为了验证系统的有效性，选取 4 个具有代表性的容器逃逸内核漏洞，并与容器逃逸检测工具 NS-Detector 进行对比。其中，利用这 4 个漏洞实现的容器逃逸涵盖了本文总结出的两类逃逸方式：直接逃逸和间接逃逸。实验结果表明，本文所提方法可以实时检测直接和间接逃逸。同时，基于真实场景的实验分析表明，本文的系统有较小的性能开销。

1　相关工作

Linux 下的容器技术基于内核中的 namespaces[9]实现隔离控制，基于 cgroups[11]实现资源分配。由于容器进程与宿主机进程共享系统内核，攻击者可以在容器内利用内核漏洞实现容器逃逸，进而对整个物理机环境造成危害[6]。为了在逃逸后获得 root 权限的交互式命令解释器（shell），攻击者通常会先提升当前进程的权限，因此可以通过动态实时监控容器内进程是否有提权行为来防御逃逸。Lin 等[12]提出的利用内核漏洞的提权一般包含 4 个步骤：绕过内核地址空间布局随机化（KASLR, kernel address space layout randomization）[13]、绕过管理模式访问保护（SMAP，supervisor mode access prevention）和管理模式执行保护（SMEP，supervisor mode execution prevention）[14]、覆盖内核函数指针劫持控制流和调用内核函数 commit_creds()。Lin 等认为前 3 个步骤很难直接检测，所以提出通过检测进程是否调用函数 commit_creds()来阻止容器内进程的提权行为。但是，在实践中存在一些其他的内核提权方法，并不会通过调用函数 comm_cred()来进行提权。比如，攻击者可以利用 Linux 内核写时复制（copy-on-write）中的条件竞争漏洞[15]，绕过虚拟动态共享对象（vDSO, virtual dynamic shared object）[16]对进程内存权限的限制，将漏洞利用代码（shellcode）[17]注入 vDSO 实现提权并逃逸。

此外，Jian 等[8]提出了一种基于进程所属 namespaces 状态变化的方案 NS-Detector（namespaces detector）来检测 Docker 容器的逃逸行为。Jian 等认为，当攻击者从容器中逃逸至宿主机并获得一个可控的 shell 时，该进程仍属于容器中进程的子进程，但其 namespaces 已突破容器与宿主机间的隔离，属于宿主机进程的 namespaces。该方案主要存在 2 个缺陷：一是检测滞后性，因为容器逃逸通常在最后一步才会突破 namespaces 的隔离；二是检测失效性，攻击者在容器内利用内核漏洞提权并获得对宿主机目录操作权限后，可以通过服务 cron 创建定时任务的方式来获取一个反弹 shell[18]，实现“间接”容器逃逸。此时，获得的可控 shell 与容器中进程无直接关系，该方案不能检测出此类型的容器逃逸。

另一类常见的检测容器逃逸的方法是安全容器[19-20]。与普通容器相比，安全容器运行在一个独立的微型虚拟机中，拥有完整的操作系统内核。安全容器是通过隔离层增强容器的安全性，即在容器和内核之间增加隔离层来阻止逃逸。典型的安全容器技术有 Kata[19]和 gVisor[20]。安全容器虽然实现了容器和内核的安全隔离，但中间隔离层的引入增加了攻击面，可能会导致其他未知的风险。

首先，本文讨论的容器逃逸检测技术主要针对普通容器。其次，Lin 等的工作用于检测内核提权，Jian 等提出的 NS-Detector 是目前唯一用于检测利用内核漏洞实现容器逃逸的工作，与本文研究内容直接相关。总体来说，当前对利用内核漏洞实现容器逃逸行为的检测方案主要面临检测漏报率较高的问题。在 2.3 节中，本文对利用内核实现的容器逃逸建模后将逃逸行为划分为直接逃逸和间接逃逸。当前的容器逃逸检测方案仅能检测直接逃逸，无法有效检测间接逃逸。本文提出异构观测链的方法可以在保证检测直接逃逸的前提下检测间接逃逸。

2　预备知识

2.1　开放起源模型 OPM

2008 年，Moreau 等[21]提出了开放起源模型（OPM, open provenance model）。OPM 的本质是一个包含不同对象间依赖关系的有向无环图，可用于描述某一对象在特定时间阶段的所有操作。OPM 示例如图 1 所示，共定义了 3 类节点。

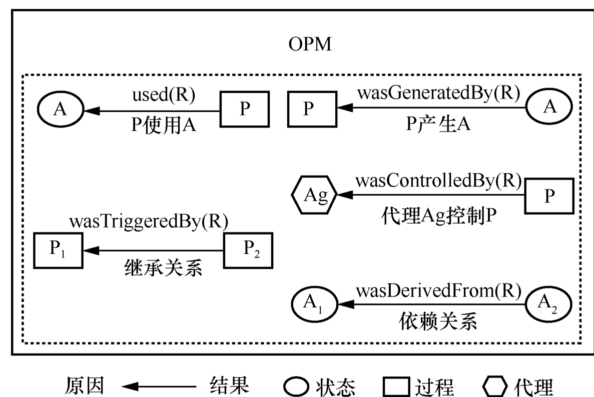


图 1　OPM 示例

1) 状态（artifact）：用椭圆形表示，代表不可变的状态，对应于计算机中的数据对象。

2) 过程（process）：用矩形表示，代表施加在数据对象上的行为，可以产生新的状态节点。

3) 代理（agent）：用六边形表示，用于控制或影响过程的执行。

3 类节点之间的边代表不同的因果（依赖）关系：有向边的头部是结果，尾部（箭头）是原因。如图 1 所示，3 类节点之间有 5 种依赖关系。

1) used：该依赖关系代表过程 P 的执行需要使用状态 A。

2) wasGeneratedBy：该依赖关系代表过程 P 的执行产生了状态 A。

3) wasTriggeredBy：该依赖关系代表过程 P_2 的执行需要过程 P_1 先执行。一般 P_1 是父进程，P_2 是子进程。

4) wasControlledBy：该依赖关系代表过程 P 的开始和结束都由代理 Ag 控制。

5) wasDerivedFrom：该依赖关系代表状态 A_2 描述的数据对象依赖于状态 A_1 所代表的数据对象。

在 OPM 中，一个过程可能使用或产生多个状态，也有可能被多个代理控制，所以需要准确地识别这些状态集和代理集。OPM 引入角色的概念，如图 1 所示，每条边上标注了角色（R），并且给每个状态和代理规定唯一标识的角色。

2.2 逃逸相关内核数据结构

在 Linux 系统中，根据特权级别不同，将进程运行状态划分为用户态和内核态。应用程序一般运行在用户态，当其执行系统调用或发生中断而陷入内核中执行时，就称进程处于内核态。当进程在内核态时，可以直接访问操作系统内核中与进程属性相关的数据结构。因此，攻击者可以利用内核漏洞劫持控制流来执行恶意代码修改容器进程与权限相关的数据结构，从而提升容器进程的权限，进一步实现逃逸。内核中与容器逃逸相关的重要数据结构如图 2 所示。

1) task_struct：内核中与进程管理和控制相关最重要的数据结构。该结构体伴随进程整个生命周期，记录进程的当前状态以及控制进程运行的全部信息（打开的文件、信号量、进程状态、地址空间等）。

2) cred：系统通过该结构体控制进程对其他资源的操作权限判定。当进程操作消息队列、共享内存和信息量等数据对象时，系统需要对进程的 euid、egid 进行检查；当进程执行文件相关的操作时，系统需要对进程的 fsuid、fsgid 进行检查。同时，Linux 使用能力（capability）机制，以细粒度方式控制普通进程能否执行“特权”操作。例如，进程要挂载（mount）一个文件系统，那么进程需要有对应的 capability，即 CAP_SYS_ADMIN。

3) fs_struct：用于描述进程的文件系统信息。结构体中的 root 和 pwd 分别代表进程的根目录和当前工作目录。

4) nsproxy：用于描述进程各个 namespaces 的状态。其中，pid namespaces 为进程提供了一个具有独立进程 ID 的运行环境。在每一个 pid namespaces 中，进程的 pid 从 1 开始编号，且和其他 pid namespaces 中的 pid 互不影响。

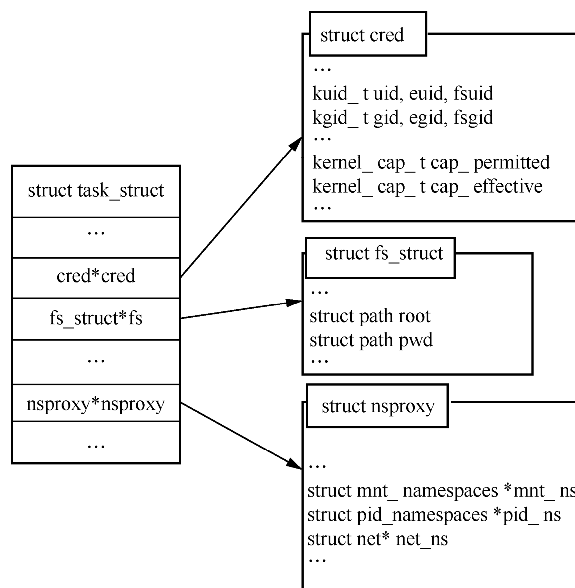


图 2　内核中与容器逃逸相关的重要数据结构

2.3 容器逃逸模型

利用内核漏洞实现容器逃逸一般可分为两步，攻击者首先利用内核漏洞劫持内核的控制流，然后执行恶意代码实现容器逃逸。在复现并观测大量利用内核漏洞的容器逃逸后，本文将逃逸行为建模为如图 3 所示的流程。根据逃逸后获取的 root 权限进程（shell）是否为容器中进程的子进程，将容器逃逸分为两类：直接逃逸和间接逃逸。

2.3.1 直接逃逸

在直接逃逸中，攻击者在劫持内核控制流之后一般会提升当前容器进程的权限，确保逃逸后可以获得 root 权限的进程。如图 3 所示，攻击者首先触

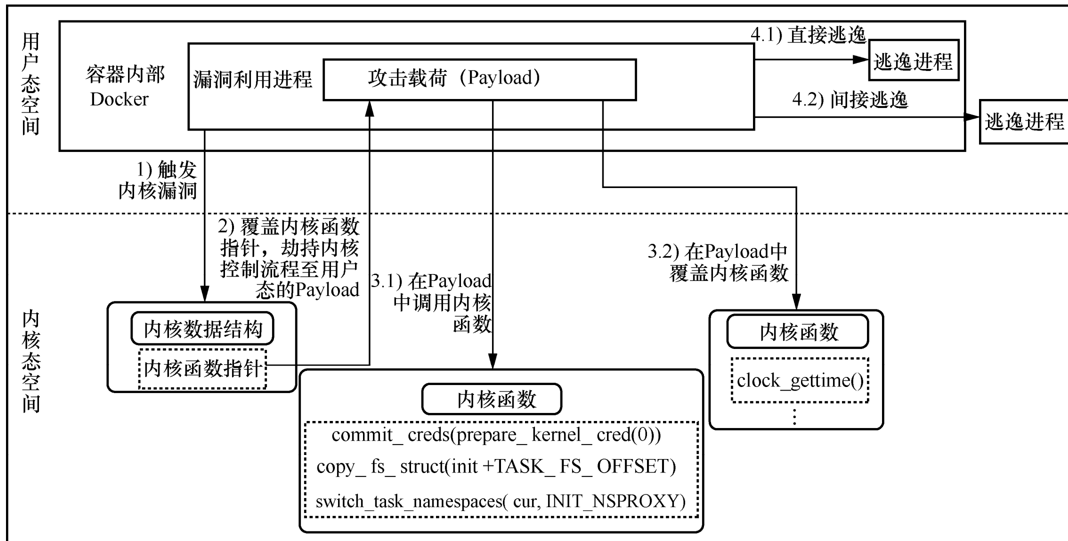


图 3　利用内核漏洞的直接、间接逃逸流程

发内核漏洞，通过覆盖内核函数指针，将内核控制流劫持到用户态。在 Payload 中调用内核函数 commit_creds(prepare_kernel_cred(0)) 将当前进程（cur）的 cred 替换为 root 权限的 cred。然后，执行内核函数 switch_task_namespacess(cur, INIT_NSPROXY) 突破系统对容器内进程的 Namespaces 隔离。最后，cur 进程创建一个 root 权限的逃逸进程，实现直接容器逃逸。

2.3.2　间接逃逸

间接逃逸与直接逃逸最大的区别是不需要容器内的进程突破 namespaces 的隔离也能获得 root 权限的逃逸进程。如图 3 中的 3.1) 所示，攻击者劫持控制流后，在 Payload 中依次执行内核函数 commit_creds(prepare_kernel_cred(0)) 和 copy_fs_struct (init + TASK_FS_OFFSET)，分别提升容器内当前进程的权限和将进程根目录切换为宿主机上 1 号进程的根目录。此时，攻击者可以通过改写宿主机文件的方式实现逃逸。例如，容器中的进程在宿主机 /var/spool/cron/crontabs/目录下新建 root 文件，向其中写入执行反弹 shell 的命令，当宿主机的 root 权限的进程 cron 执行该文件中的命令后获得一个 root 权限的交互式 shell。最后得到的 root 权限进程与容器中的进程并无直接继承关系，实现间接容器逃逸。另一种实现间接逃逸的方式是在劫持内核控制流后不调用内核函数，而是用 shellcode 覆盖内核中的函数。如图 3 中的 3.2) 所示，攻击者覆盖内核中的 clock_gettime()函数，随后宿主机上 root 权限的进程调用 clock_gettime()函数获取时间时会开启一个 root 权限的 shell，实现间接逃逸。

因此，利用内核漏洞实现的容器逃逸在其逃逸过程中所执行的操作一般会表现出“权限提升”的特点。从攻防双方来看，攻击者可以凭此探索新的容器逃逸方法，一旦有新的内核漏洞，就可以考虑是否可用于容器逃逸；防守者则可以针对此特征来检测容器进程生命周期中有无权限提升发生，以此来防御利用内核漏洞实现的容器逃逸。

进程执行不同操作前后的属性变化能体现出权限提升，而进程属性本质上由内核中的数据结构来控制。因此，为了观测进程生命周期中是否存在权限提升，本文基于内核中的数据结构选取观测点列表 S 为[fs_root，cap_permited，uid，gid，mnt_ns，pid_ns，net_ns]，其中，fs_root 表示进程的根目录，cap_permited 表示进程所能够拥有特权的上限，uid 和 gid 分别表示进程所属的用户 ID 和组 ID，观测点列表中剩余部分均与进程的 namespaces 相关。

为检测上述 2 类容器逃逸，本文基于进程的操作构建进程起源图，以此来描述进程行为之间的关系。随后，结合进程起源图和观测点构建异构观测链，对进程完整生命周期进行监控，即检测进程行为引发的观测点属性变化情况，若存在权限提升的属性变化，则判定进程正在进行逃逸。

3　系统实现

本文所提出的基于异构观测链的容器进程实时检测系统旨在通过对容器内的进程进行全生命周期的监控来检测是否存在利用内核漏洞的容器逃

逸行为。为此，一方面需要确保监控和检测过程的实时性，在不能引入过高的性能开销的同时也需要保证系统的稳定运行；另一方面需要保证检测的有效性。因此，系统的设计目标包括以下 4 个方面。

1) 保证捕捉进程行为和提取进程属性信息的实时性。基于此要求，HOC-Detector 需要及时捕获进程的操作，然后提取其行为、属性信息。

2) 尽量减小逃逸检测时延。为减小检测逃逸的时延，除需要快速提取进程的行为、属性信息外，还需要缩小逃逸检测的范围。

3) 引入较小的性能开销。在确保进程信息提取的实时性、检测低时延的前提下，尽量减小 HOC-Detector 对系统造成运行时开销。

4) 能够防御未知威胁。对于使用未知内核漏洞进行容器逃逸的行为，HOC-Detector 要能准确地捕捉，以确保逃逸检测的有效性。

3.1 异构观测链概述

进程的动态行为可由其所执行的一系列操作来刻画。当进程进行容器逃逸时，往往会执行一些特别的操作来绕过系统中的安全机制，而操作前后进程的属性变化可以反映出这些操作是否合法。为此，本文借助 OPM 将系统中进程执行的所有操作抽象为进程起源图，从起源图上提取容器内进程的观测链。如图 4 所示，该观测链由进程、文件等对象以及表征其相关操作的有向依赖边构成。

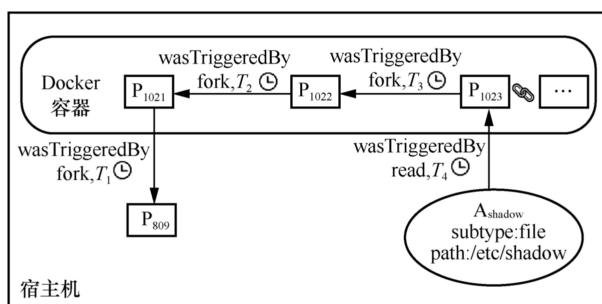


图 4 基于进程行为起源图的异构观测链

本文将观测链中依赖边的尾部（箭头）节点称为主体，用 T（tail）表示；边的头部节点称为客体，用 H（head）表示；主体和客体均可以是进程、文件等。进程用 P_{pid} 表示，其中 pid 为进程号；文件等其他资源用 A_{name} 表示，其中 name 为标识符。主体与客体间的行为用 O_{op} 表示，其中 op∈｛fork, clone, read, write,⋯｝；主体对客体之间的操作关系用 $TO_{op}H$ 表示。

如图 4 所示，节点代表 containerd-shim 进程，在 T_2 时刻，容器中的 init 进程 P_{1021} 复制了一个子进程 P_{1022}，该操作可以表示为 $P_{1021}O_{fork}P_{1022}$。在 T_3 时刻，P_{1022} 运行漏洞利用程序后创建了一个新的进程 P_{1023}，该操作可以表示为 $P_{1022}O_{fork}P_{1023}$。在 T_4 时刻，该进程读取了宿主机中/etc/shadow 文件中的内容 $P_{1023}O_{read}A_{shadow}$。从图 4 中得到的观测链为 $(P_{1021}O_{fork}P_{1022})\wedge(P_{1022}O_{fork}P_{1023})\wedge(P_{1023}O_{read}A_{shadow})$。其中，符号 $\wedge$ 用来连接观测链上进程的操作。

在得到容器内进程的观测链后，本文基于 2.3 节中提出的观测点列表 S，在进程生命周期中不同的时间节点对进程的操作进行异构观测。采用 3.5 节中提出的检测标准来评估进程操作是否存在权限提升，以此来判断是否发生了逃逸行为。使用异构观测链的方法来检测容器逃逸有 2 个明显的优势，一是异构观测链涵盖进程在不同时间点所执行的操作，可以在早期就检测到进程的逃逸行为，显著降低逃逸检测的时延；二是异构观测链包含进程生命周期中所有相关的操作，可以对进程实现全面的监控，显著降低逃逸检测的漏报。

3.2 总体架构

HOC-Detector 总体架构如图 5 所示，由进程日志获取模块（AuditLogger）、进程起源图生成模块（PPGGenerator）、异构观测链构建模块（HOCBuilder）、攻击检测模块（AttackDetector）和预先配置的观测点（ObservationPoints）五部分组成。在 HOC-Detector 运行过程中，AuditLogger 在系统内核中监视进程，并将其行为、属性信息保存在宿主机的日志文件中；与此同时，PPGGenerator 从日志文件中读取内容，采取基于因果关系的方法构建进程的起源图，并将其传递给 HOCBuilder；HOCBuilder 基于起源图并结合用户提供的观测点构建异构观测链，AttackDetector 对异构观测链进行分析，检测容器进程是否存在逃逸行为。

为了对基于内核的容器逃逸有更清晰的了解，本文在 2.3 节中通过对内核漏洞导致的容器逃逸进行大量复现后，对其逃逸流程进行建模。同时，从是否能很好地表征权限提升的角度选取进程的属性信息作为观测点。如图 5 中的 ObservationPoints 所示，观测点列表为 $[S_{[1]}, S_{[2]}, \cdots, S_{[m]}]$。同时，为了增加对未知威胁的防御能力，可以动态调整 HOC-Detector 中的观测点。

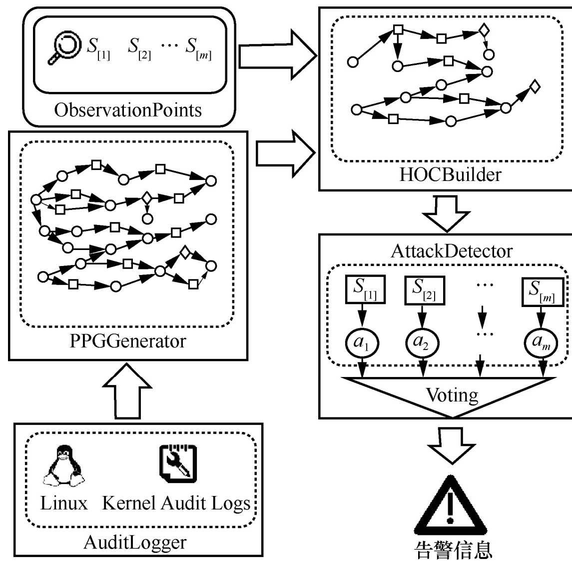


图 5　HOC-Dectector 总体架构

在 HOC-Detector 的运行过程中，AuditLogger 需要持续实时地捕捉进程的行为和记录属性的信息。因此，本文使用内核编程的方式通过动态可加载的内核模块在内核中截取进程所执行的系统调用，记录其系统调用名、参数和返回值等信息，实现对进程行为的实时监控。HOC-Detector 的设计目标是检测利用内核漏洞的容器逃逸，而容器本质上是宿主机上使用 namespaces 和 cgroups 实现资源隔离的普通进程。宿主机上的进程数目庞大，为了实现容器逃逸检测的实时性，本文需要重点关注容器相关的进程。因此，需要识别出容器内外进程的边界，减轻后续异构观测链构建和攻击检测的复杂度。PPGGenerator 对记录的进程日志信息进行语义解析，构建进程起源图。随后，HOCBuilder 结合起源图和观测点构建异构观测链。最后，AttackDetector 基于异构观测链检测进程的整个生命周期中是否存在权限提升的情况，判断是否存在容器逃逸行为。如图 5 中的 AttackDetector 所示，该模块对进程操作前后的每个观测点 S_i 进行对比，判断其是否存在权限提升，a_i 表示检测结果。当超过半数的观测点存在权限提升时，本文判定当前进程存在逃逸行为。

3.3　基于内核模块的进程信息提取

Linux 系统中的/proc 目录是一种虚拟文件系统，该目录下的文件描述了内核当前的运行状态，可以在用户态通过读取/proc/pid 目录下文件的内容来查看当前正在运行进程的信息。该方法主要面临 2 个挑战：一是系统中的进程数目庞大，频繁读取/proc 目录下文件中的内容开销很大；二是攻击者在实施攻击的过程中可能会隐藏一些进程，导致不能在/proc 目录下找到对应进程的信息。因此，本文通过内核模块编程的方式在内核态提取与进程相关的关键属性信息，并将其保存到宿主机的日志文件中。

本文所选取的观测点可以通过进程描述符 task_struct 获取，所以首先需要得到进程的 task_struct 结构体，然后根据其结构体成员提取进程观测点信息。该功能由 HOC-Detector 中的 AuditLogger 模块实现，采用 Linux Audit[22]技术在内核中捕捉系统中所有进程的各种操作行为，如系统调用、文件读写、执行命令等。其具体实现流程如下。

1) 在内核态截取与新进程创建相关的系统调用，例如 clone、fork 和 vfork。当进程执行这些系统调用后，将其返回值传给/include/linux/pid.h 中的函数 find_vpid()以获取新进程的 struct pid 结构体指针，它指向的结构体保存了新进程的进程描述符信息。

2) 把步骤 1)中指向 struct pid 的指针作为函数 pid_task()的参数，得到进程的 task_struct 结构体。

3) 利用__task_cred()读取 task_struct 中的指针 cred 所指向的结构体，从中提取进程的用户和组权限 uid、gid、euid、fsuid 等。同时，提取进程的细粒度权限信息 cap_permitted。

4) 通过 task_struct 中的指针 fs 指向的结构体获得进程根目录 root，然后根据 root 中的 dentry 得到根目录项中表示所有子目录的链表指针 d_subdirs。最后，通过遍历 d_subdirs 得到根目录下所有子目录或子文件的名字。

5) 通过 task_struct 中的指针 nsproxy 指向的结构体获得进程 namespaces 的信息。

6) 系统中的每个进程可能属于多个不同的 namespaces，通过结构体 struct pid 中的 numbers 数组来获取进程在不同 pid namespaces 中的进程编号。在容器逃逸场景下，pid 表示进程在宿主机环境下的进程号（全局进程号），vpid 表示容器中进程在其隔离的 pid namespaces 中的进程号（局部进程号）。

如图 6 所示，runc 进程（父进程，pid=2556）正在执行容器初始化，它通过执行系统调用号为 56 的 clone 来创建一个子进程（pid=2557）。本文将进程编号 2557 作为参数依次执行上面描述的操作来提取新进程的信息。为了方便对比在新进程创建时有无权限提升的情况发生，本文也会提取父进程的相关属性信息。如图 6 所示，子进程的命名空间与

type=USER msg=audit(1647870972.048:10015): info=child fs_root=container ns_syscall=56 ns_subtype=ns_namespaces ns_operation=ns_NEWPROCESS **pid=2557 vpid=1** level=1 uid=0 gid=0 euid=0 suid=0 fsuid=0 egid=0 sgid=0 fsgid=0 cap_permitted=0000003ffffffff mnt_ns=4026532496 net_ns=4026532501 pid_nes=4026532499

type=USER msg=audit(1647870972.048:10015): info=current root_path=host ns_syscall=56 ns_subtype=ns_namespaces ns_operation=ns_NEWPROCESS **pid=2556 vpid=2556** level=0 uid=0 gid=0 euid=0 suid=0 fsuid=0 egid=0 sgid=0 fsgid=0 cap_permitted =0000003ffffffff mnt_ns =4026531840 net_ns=4026531957 pid_ns=4026531836

type=SYSCALL msg=audit(1647870972.048:10015): arch=c000003e syscall=56 success=yes **exit=2557** a0=8011 a1=7ffd13ab1080 a2=55dab7f780d0 a3=0 items=0 ppid=2532 **pid=2556** auid=4294967295 uid=0 gid=0 euid=0 suid=0 fsuid=0 egid=0 sgid=0 fsgid=0 tty=(none) ses=4294967295 comm="runc:[1:CHILD]" exe="/" key=(null)

图 6　内核模块提取的进程观测点信息

父进程的命名空间（mnt_ns, pid_ns, net_ns）不一致，并且子进程的局部进程编号为 1（vpid=1），说明该子进程是新启动的容器中的 init 进程。为了方便表示进程的根目录，本文将容器内的进程根目录 root_path 标识为 container，容器外进程标识为 host。

3.4　容器内外进程边界识别

宿主机操作系统使用 Linux 中的 Namespaces 技术来限制容器进程可见的资源，使运行在同一宿主机下不同容器中的程序之间互不影响。使用 cgroups 技术来限制进程可使用的资源，确保宿主机上所有容器公平使用宿主机上的资源，如 CPU、内存、磁盘和网络等。所以，容器仅是宿主机上的一个特殊的进程，与其他进程相比没有本质的区别。但 Linux Audit 在记录进程信息时并不会区分当前进程是普通进程还是容器进程，而容器内进程的行为是本文检测容器逃逸的关键。为了优化异构观测链的构建以及提高容器逃逸的检测效率，本文提出建立对容器内外进程边界的识别，该过程仅需关注与容器内进程及其行为相关的进程。

通过分析 Docker 容器的初始化流程可以发现，容器的启动有固定的模式，基于此来识别容器内外进程的边界。容器的初始化以执行系统调用 unshare 创建新的命名空间系统开始，以执行系统调用 execve 在容器内创建新的 init 进程结束。本文以 Docker 容器的初始化为例，如图 7 所示，其初始化流程介绍如下。

1) dockerd 通过 gPRC 接口向容器运行时管理引擎 containerd 发送指令创建容器，随后 containerd（pid=1405）会启动一个进程 containerd-shim（pid=2522），由它负责创建一个新的容器。

2) 该进程（pid=2522）会复制出一系列的 containerd-shim 进程，其中一个 containerd-shim（pid=2532）会创建新的 runC 进程来完成容器初始化操作。

3) 进程 runc（pid=2540）会复制出子进程 runc:[0:PARENT]，该进程的子进程 runc:[1:CHILD] 会执行系统调用 unshare 创建新的命名空间，标志着开始进行实际的容器初始化。

4) 进程 runc:[1:CHILD] 会复制一些子进程 runc:[2:INIT]来完成一些特定的初始化任务，包括设置/proc 和/rootfs 等。

5) 最后，进程 runc:[1:CHILD]将复制一个子进程（pid=2563），该子进程即容器内的 init 进程，它将执行系统调用 execve 来运行容器默认的启动程序（如 bash）。

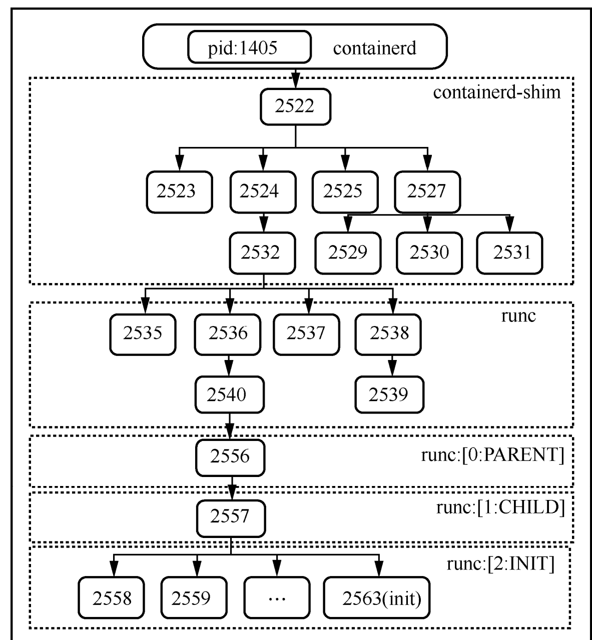


图 7　Docker 容器初始化流程

HOC-Dectector 中的 PPGGenerator 模块基于系统收集的进程行为信息构建进程起源图，本文使用上述容器初始化模式在起源图上标记容器内外进程的边界，仅关注容器内的进程，减轻后续构建观测链的工作量。

3.5 异构观测链的构建和逃逸行为的检测

观测链的构建依赖于能够将进程生命周期内所有行为之间的依赖关系抽象为有向的进程起源图。HOC-Detector 中的 HOCBuilder 模块利用开源框架 SPADE[23]来提取进程起源图，利用 3.4 节中描述的方法识别容器内外进程的边界和负责启动容器的 containerd-shim 进程。然后，以启动容器的 containerd-shim 进程为根节点，以广度优先的方式遍历起源图，直到覆盖所有子节点或者达到最大深度，最终从完整的起源图中提取出与容器进程相关的子图。如图 4 所示，本文将从 containerd-shim 进程的节点到叶节点的一个节点序列称为观测链。

如 2.3 节所述，本文选取的观测点列表为 S。本文在 AttackDetector 中将主体 T 和客体 H 之间单个观测点的检测标准 Δ 定义为

$$\Delta(\mathrm{T}_{S_{[i]}},\mathrm{H}_{S_{[i]}})=\begin{cases}1, \mathrm{T}_{S_{[i]}}<\mathrm{H}_{S_{[i]}}\\0, 其他\end{cases} \tag{1}$$

其中，i 表示对观测点列表 S 的索引，$\mathrm{T}_{S_{[i]}}$ 和 $\mathrm{H}_{S_{[i]}}$ 分别表示主体和客体在观测点 $S_{[i]}$ 处的值。当 $\mathrm{T}_{S_{[i]}}$ 的值“小于” $\mathrm{H}_{S_{[i]}}$ 时，$\Delta(\mathrm{T}_{S_{[i]}},\mathrm{H}_{S_{[i]}})=1$，也即在此观测点处存在权限提升。本文将进程在观测点处的值从大到小标记为 3 类：宿主机上 1 号进程对应的值，空值（None）和其他值。例如，客体在观测点处的值 $\mathrm{H}_{S_{[i]}}$ 与宿主机 1 号进程对应值相等，主体 $\mathrm{T}_{S_{[i]}}$ 为其他值，则 $\mathrm{T}_{S_{[i]}}<\mathrm{H}_{S_{[i]}}$。本文将主体和客体之间的权限检查函数 F_{AC} 定义为

$$F_{\mathrm{AC}}=\sum_{i}^{m}\left(\Delta(\mathrm{T}_{S_{[i]}},\mathrm{H}_{S_{[i]}})\right) \tag{2}$$

其中，m 为本文选取的观测点列表 S 的长度，如 2.3 节所述，m=7。冗余技术在软件系统中可以极大提升系统的容错能力，从而提高系统的稳定性。本文使用具有冗余能力的多数表决算法[24]来判定当前操作是否存在权限提升。其基本思想是如果有超过半数观测点的检测标准 Δ 值为 1，则判定为当前主客体间的操作存在权限提升的行为，即如果 $F_{\mathrm{AC}}\geqslant\frac{m+1}{2}$（当 $m=7$ 时，$F_{\mathrm{AC}}\geqslant 4$）则判定当前的主客体间的操作存在权限提升，否则为正常操作。进程的观测链由多条边组成，只要有一条边（主客体间的操作）存在权限提升，本文就认为该进程发生容器逃逸。

4 仿真分析

4.1 实验设置

为了验证 HOC-Dectector 在防御利用内核漏洞的容器逃逸攻击时的有效性，本文选取了 4 个影响较大的、可通过不同方式实现容器逃逸的 CVE 内核漏洞。本文的实验环境主要由物理机和容器引擎 Docker 组成，物理机的配置为 Intel(R) Core(TM) i7-9750H，4 核处理器，10 GB 内存，200 GB 硬盘，具体的 CVE 信息和系统实验环境如表 1 所示。本文选择的 CVE 包括 CVE-2017-7308、CVE-2017-18344、CVE-2017-1000112 和 CVE-2016-5195。第一个 CVE 可通过调用内核函数的方式实现直接容器逃逸；第二个和第三个 CVE 一起使用，可通过调用内核函数实现间接容器逃逸；最后一个 CVE 可通过不调用内核函数的方式实现间接容器逃逸。

为检验 HOC-Detector 的有效性，本文复现了 Jian 等提出的方法 NS-Detector 并与之对比。NS-Detector 通过监测进程所属 namespaces 的状态变化来检测针对 Docker 容器的逃逸攻击。其主要思想是攻击者实现容器逃逸攻击后获得的 root 权限逃逸进程仍属于容器内进程的子进程，但其所属 Namespaces 已脱离容器的限制。本文将 NS-Detector 和 HOC-Detector 在前述 3 个容器逃逸的 CVE 实验中进行对比，实验结果如表 1 所示。其中，HOC-Detector 能检测出所有的容器逃逸，准确率达到 100%，而 NS-Detector 的准确率约为 33%。

4.2 CVE-2017-7308

AF_PACKET 是在 Linux 平台下的一种套接字（socket），用于在设备驱动层发送或者接收数据包。进程可以使用 send 和 recv 这 2 个系统调用在数据包套接字上发送和接收数据包。为了提升效率，套接字提供了一个环形缓冲区（ring buffer），能够使数据包的发送和接收更为高效，且这个环形缓冲区可以在内核态和用户态之间共享。在利用套接字收发数据时，每个数据包会存放在一个单独的帧（frame）中，多个帧会被分组形成内存块（block）。

表 1　容器逃逸检测的内核漏洞信息及实验环境和结果

实验	内核漏洞信息			实验环境				实验结果	
	CVE-ID	漏洞类型	影响版本	操作系统版本	内核版本	Docker 版本	容器逃逸方式	HOC-Detector	NS-Detector
1	CVE-2017-7308	整型溢出漏洞	~4.10.6	64 位 Ubuntu 16.04	4.8.0-34-generic	18.09.0	直接	√	√
2	CVE-2017-18344 CVE-2017-1000112	整型溢出漏洞 堆溢出漏洞	~4.14.8 ~4.13.9				间接	√	×
3	CVE-2016-5195	内核条件竞争漏洞	2.x~4.8.3		4.4.0-43-generic		间接	√	×

CVE-2017-7308 漏洞存在于内核处理版本为 TPACKET_V3 的环形缓冲区的代码中，由于内核在判断接收到数据的长度时存在整数溢出，使其通过长度的安全性检查而导致堆溢出漏洞。攻击者可以通过溢出控制函数指针来劫持内核的控制流。利用该内核漏洞进一步实现容器逃逸的攻击流程如下。

1) 通过/proc/kallsyms 或 syslog 的方式泄露内核函数地址来获得内核的加载基址，绕过内核 KASLR 保护机制。

2) 构造堆布局触发内核处理环形缓冲区代码 packet_set_ring()中的堆溢出漏洞。

3) 覆盖 packet_socket 结构体中的 retire_blk_timer 字段，使该字段中的函数指针 func 指向 native_write_cr4()。同时，覆盖该字段中的 data 为函数 native_write_cr4()的参数。若计时器超时，执行 native_write_cr4()来禁用 SMEP 和 SMAP。

4) 以同样的方式构造堆溢出，覆盖 packet_sock 中的 xmit 字段，使其指向攻击载荷函数 get_root_payload()。

5) 在 get_root_payload() 函数中首先执行 commit_creds()来提升容器内当前进程的权限；然后，执行 switch_task_namespaces()来切换容器内已提权进程的 namespaces，实现逃逸。由表 1 可知，HOC-Detector 和 NS-Detector 均能检测到利用漏洞 CVE-2017-7308 实现的逃逸行为。HOC-Detector 对利用漏洞 CVE-2017-7308 实现的容器逃逸过程进行监测所获取的观测链如图 8 所示。攻击者通过运行逃逸攻击程序 exploit 获得一个 root 权限的进程（pid=3046）。其中，进程（pid=3045）执行系统调用 clone 产生新进程（pid=3046）的操作存在权限提升。漏洞 CVE-2017-7308 权限提升操作处的观测点属性变化如表 2 所示。其中，容器内的进程（主体 T，pid=3045）复制出的子进程（客体 H，pid=3046）在所有观测点的值均与宿主机上 1 号进程的值相等（$\Delta(\mathrm{T}_{S_{[i]}},\mathrm{H}_{S_{[i]}})=1, 0\leqslant i\leqslant 6$），即 $F_{\mathrm{AC}}=7$，说明出现了权限提升，HOC-Detector 可以检测出利用该内核漏洞实现的直接逃逸行为。由于获得的逃逸进程（pid=3046）是容器中进程（pid=3045）的子进程，且两者的 namespaces 不一致，NS-Detector 也能检测到该逃逸行为。

4.3　CVE-2017-18344 和 CVE-2017-1000112

CVE-2017-18344 存在于 Linux 内核 4.14.8 之前的版本中，在系统调用 timer_create 的实现 kernel/time/posix-timers.c 中没有正确验证结构体 sigevent 中的字段 sigev_notify，存在整型溢出漏洞。当用户读取/proc/pid/timers 中的内容时，可以利用

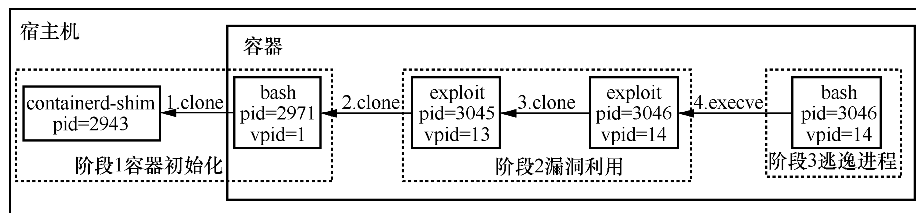


图 8　HOC-Detector 对利用漏洞 CVE-2017-7308 实现的容器逃逸过程进行监测所获取的观测链

表 2　漏洞 CVE-2017-7308 权限提升操作处的观测点属性变化

pid	fs_root	uid	gid	cap_permitted	mnt_ns	pid_ns	net_ns
1	host	0	0	0000003ffffffffff	4026531840	4026531836	4026531957
3045	container	5000	5000	0000000000000000	4026532688	4026532691	4026532693
3046	host	0	0	0000003ffffffffff	4026531840	4026531836	4026531957

该漏洞在 posix-timers.c 的函数 show_timer()中实现越界访问，进一步使用户态程序可以读取内核中任意地址的内容。CVE-2017-1000112 存在于 Linux 内核 4.13.9 之前的版本中，该漏洞位于/net/ipv4/ip_output.c 中的__ip_append_data()，漏洞形成的原因是内核通过标志 SO_NO_CHECK 来判断在处理数据包时是使用 UFO 机制还是 non-UFO 机制（UFO 机制是指用网卡辅助进行报文分片，用户层协议不进行分片；non-UFO 机制是指在用户层进行报文分片）。具体来说，第一次调用 send()函数发送数据包时，本文可以发送一个大小超过 MTU 的数据包，这将执行 UFO 的处理逻辑。在第二次调用 send()发送数据包之前，先执行系统调用 setsockopt 来设置 SO_NO_CHECK，使其执行 non-UFO 的处理逻辑，触发_ip_append_data()函数中的越界写漏洞。攻击者可以利用这 2 个内核漏洞实现间接容器逃逸，其攻击流程如下。

1) 构造堆布局触发 CVE-2017-18344，实现任意地址读，泄露中断描述符表（IDT, interrupt descriptor table）第一项（divide_error）的地址，并根据它的地址计算出内核的加载基址，绕过内核 KASLR 保护机制。

2) 利用 CVE-2017-1000112 实现任意地址写，劫持控制流至攻击载荷函数 get_root_payload()。

3) 在 get_root_payload()函数中首先执行 commit_creds()函数来提升容器内当前进程的权限；然后，执行_copy_fs_struct()函数来将当前进程的根目录切换为宿主机上 1 号进程的根目录。

4) 在宿主机目录/var/spool/cron/crontabs/下创建文件 root，向其中写入开启反弹 shell 的代码：echo '* * * * * bash -c " bash -i >& /dev/tcp/IP/PORT 0>&1 " ' >> root。

5) 攻击者在受控端执行 nc-vnlp 12345 等待容器所在宿主机反弹 shell 的连接，获取一个 root 权限的反弹 shell，实现逃逸。

由表 1 可知，HOC-Detector 可以检测出该逃逸行为，而 NS-Detector 则不可以。HOC-Detector 对利用漏洞 CVE-2017-18344 和 CVE-2017-1000112 实现的容器逃逸过程进行监测所获取的观测链如图 9 所示，在阶段 2 中，漏洞利用程序 exploit 首先执行 touch 命令，在宿主机目录/var/spool/cron/crontabs 下新建文件 root，随后执行 write 系统调用向其写入内容。宿主机上的进程（pid=1121）读取 root 文件中的内容，随后通过执行 connect 系统调用与远端（攻击者控制）建立 socket 连接，开启一个 root 权限的反弹 shell。漏洞 CVE-2017-18344 和 CVE-2017-10000112 权限提升操作处的观测点属性变化如表 3 所示，容器内进程（主体 T，pid=2874）复制出子进程（客体 H，pid=2798），子进程的 fs_root、uid、gid、cap_permitted 均变为宿主机 1 号进程对应的值（$\Delta(\mathrm{T}_{S_{[i]}},\mathrm{H}_{S_{[i]}})=1, 0\leqslant i\leqslant 3$），即 $F_{\mathrm{AC}}=4$，说明该操作存在权限提升。同时，容器内的进程（pid=2798）所执行的 5.create 和 6.write 均是对宿主机上的文件进行操作，也存在权限提升，HOC-Detector 可以检测到该间接逃逸攻击。

NS-Detecoor 认为逃逸进程属于容器进程的子进程，且其 Namespaces 与宿主机 1 号进程的 namespaces 一致。在本次实验中，虽然容器内进程的 namespcae 在图 9 中的操作（3.clone）前后发生变化，如表 3 所示，子进程（pid=2798）与父进程

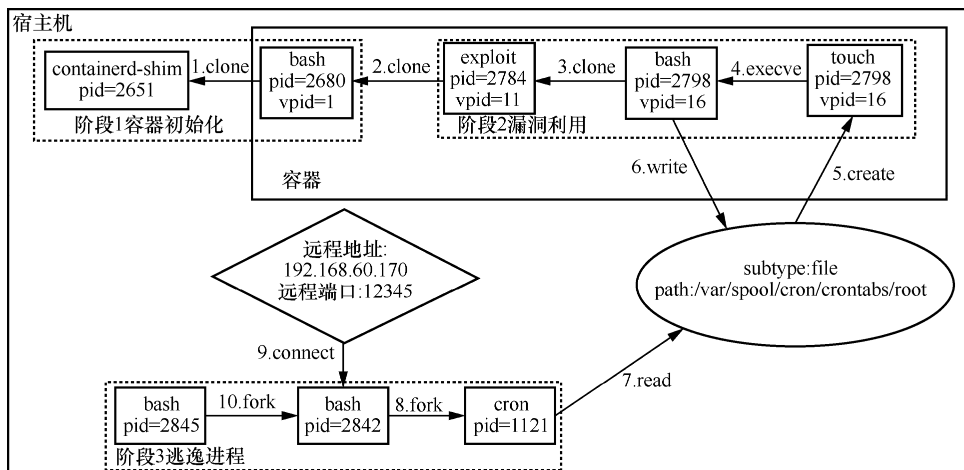


图 9　HOC-Detector 对利用漏洞 CVE-2017-18344 和 CVE-2017-1000112 实现的容器逃逸过程进行监测所获取的观测链

表 3 漏洞 CVE-2017-18344 和 CVE-2017-1000112 权限提升操作处的观测点属性变化

pid	fs_root	uid	gid	cap_permitted	mnt_ns	pid_ns	net_ns
1	host	0	0	0000003ffffffffff	4026531840	4026531836	4026531957
2784	container	5000	5000	0000000000000000	4026532462	4026532465	4026532467
2798	host	0	0	0000003ffffffffff	4026532462	4026532465	4026532523

（pid=2784）的 net_ns 不一致，但子进程（pid=2798）的 namespaces 与宿主机 1 号进程不一致。即在本次间接逃逸攻击中，容器内新创建进程的 namespaces 与宿主机 1 号进程的 namespaces 不一致，NS-Detector 不能检测出该间接逃逸行为。此外，图 9 中阶段 3 得到的逃逸进程（pid=2845）是宿主机上 cron 进程的子进程，而非容器进程的子进程。

4.4 CVE-2016-5195

CVE-2016-5195（脏牛漏洞）是一个写时复制的条件竞争漏洞，影响 Linux 内核 4.8.3 之前的版本。条件竞争[25]是指一个系统的运行结果依赖于不受控制的事件的先后顺序，通常发生在多个进程（线程）同时访问和操作相同的数据时。写时复制[26]允许不同进程中的虚拟内存映射到相同物理内存页面的技术。写时复制一般包含 3 个重要步骤：创建映射内存的副本；更新页表，使虚拟内存指向新创建的物理内存；写入内存。由于这 3 个步骤不是原子性的，一个进程在执行这 3 个步骤过程中可能被其他进程中断，从而触发写时复制的条件竞争。

攻击者可利用脏牛漏洞实现对 vDSO[16]的任意写，从而劫持控制流实现容器逃逸。vDSO 是内核提供的虚拟动态链接库（.so），当程序启动时，内核把 vDSO 映射入进程内存空间，程序将其当作普通动态库来调用其中的函数。在 Docker 容器中，本文通过利用写时复制的条件竞争漏洞（脏牛漏洞）实现对 vDSO 的任意写，将 vDSO 中的函数 clock_gettime()用 shellcode 覆盖，进而实现容器逃逸，具体流程如下。

1) 创建一个具有 capability SYS_PTRACE 的容器。这是因为漏洞利用代码使用 PTRACE 进行代码注入，但是 Docker 在容器启动时默认过滤了 SYS_PTRACE。

2) 运行漏洞利用代码，它将创建 2 个线程 ptrace_thread 和 madvise_thread。

3) 线程 ptrace_thread 利用 ptrace 不断向 vDSO 写入数据。

4) 线程 madvise_thread 不断执行 madvise 系统调用，将 vDSO 地址空间标记为 MADV_DONTNEED，内核将会释放 vDSO 所在内存区域，进程的页表会重新指向原始的物理内存。

5) 由线程 ptrace_thread 和 madvise_thread 触发写时复制的条件竞争漏洞，实现向只读内存区域写入数据的功能，即用 shellcode 覆盖 vDSO 中的函数 clock_gettime()。

6) shellcode 会检查是否是 root 权限进程在调用 clock_gettime 函数，若是，则开启一个 root 权限反弹 shell；若不是，则执行原来的 clock_gettime()函数。

本次实验结果和前一个间接逃逸的实验结果一致，HOC-Detector 可以检测出该逃逸行为，而 NS-Detector 则不可以。HOC-Detector 监控利用该漏洞进行逃逸的流程，得到如图 10 所示的观测链。攻击者执行漏洞利用代码 deadbeef，用 shellcode 覆盖 vDSO 中的函数 clock_gettime() 后，进程 containerd-shim 在某时刻执行函数 clock_gettime() 开启一个 root 权限的 shell（pid=1738）。漏洞 CVE-2016-5195 权限提升操作处的观测点属性变化如表 4 所示，由于该逃逸行为并未执行内核函数 commit_creds() 来进行提权，因此 uid、gid 和 cap_permitted 的值并未变化。在内核中读取子进程（pid=1739）的 fs_root 和 namespcae 相关属性时，其相关的指针为空，本文将其属性值标记为 None。本文认为属性值为 None 是仅低于 root 权限的属性值。容器内进程（主体 T，pid=21729）复制出子进程（客体 H，pid=1739），子进程的 fs_root、mnt_ns、pid_ns 、 net_ns 均比父进程对应的值大（$\Delta(\mathrm{T}_{S_{[i]}},\mathrm{H}_{S_{[i]}})=1$，$i\in[0,4,5,6]$），即 $F_{\mathrm{AC}}=4$，说明该操作存在权限提升，HOC-Detector 可以检测到该间接逃逸攻击。另外，containerd-shim 进程（pid=1670）的任务是用来启动一个容器，但它通过执行 3.fork 创建了一个 root 权限的 bash 进程（逃逸进程，pid=1738），超出其原有能力范围，从此角度看，该操作也存在权限提升。

在本次实验中，NS-detector 不能检测到容器逃逸攻击的原因与 4.3 节中的实验一样。容器内新创

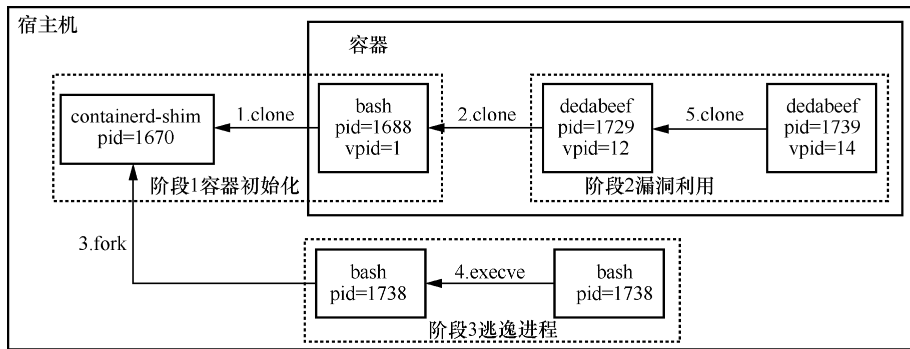


图 10　HOC-Detector 对利用漏洞 CVE-2016-5195 实现的容器逃逸过程进行监测所获取的观测链

表 4　　漏洞 CVE-2016-5195 权限提升操作处的观测点属性变化

pid	fs_root	uid	gid	cap_permitted	mnt_ns	pid_ns	net_ns
1	host	0	0	0000003fffffffff	4026531840	4026531836	4026531957
1729	container	5000	5000	0000000000000000	4026532458	4026532461	4026532463
1739	None	5000	5000	0000000000000000	None	None	None

建子进程的namespaces并不等于宿主机1号进程的namespaces，且最后获得的 root 权限逃逸进程（pid=1738）不是容器内进程的子进程。综合分析 4.3 节和 4.4 节中的 2 个实验，NS-Detector 无法检测到间接逃逸攻击的根本原因在于它对容器逃逸行为的画像不够全面，仅关注容器内的进程和它们的 namespaces。HOC-Detector 通过收集宿主机上所有进程的行为信息，提取与容器内进程所有相关的操作构建观测链。同时，本文经过大量复现内核逃逸后提炼出一些具代表性的观测点，实现对容器内进程全生命周期的异构观测，使其具备检测间接逃逸攻击的能力。

4.5　性能开销

4.5.1　实验设置

本文的性能评估方式与 CLARION[27]类似，使用基于容器的微服务数据集对 SPADE 和 HOC-Detector 进行性能开销的评估。本文使用谷歌提供的一个知名的微服务集 Online Boutique[28]作为本文的数据集。该数据集目前包含 11 个用不同编程语言编写的微服务，这些微服务通过 gRPC 相互通信。

4.5.2　运行时开销

为了评估 HOC-Detector 的运行开销，本文独立地启动每个微服务 50 次，并记录这 50 个容器微服务初始化的累计时间。首先，本文在没有启动 Linux Audit 的情况下执行这一流程，以获得一个基础测试时间。然后，本文分别用 Linux Audit、SPADE 和 HOC-Detector 重复这一流程。运行时开销比较如表 5 所示。其中，增量开销是通过对比 HOC-Detector 和 SPADE 的开销计算得到的，而总体开销则是 HOC-Detector 与基础测试的性能比较。本文系统

表 5　　运行时开销比较

微服务	基础测试/s	Linux Audit/s	SPADE/s	HOC-Detector/s	增量开销	总体开销
adservice	47	51	53	57	7.5%	21.3%
cartservice	31	35	43	48	11.6%	54.8%
checkoutservice	35	38	48	53	10.4%	51.4%
currencyservice	28	31	37	40	8.1%	42.9%
emailservice	35	40	44	49	11.4%	40.0%
frontend	33	36	45	49	8.9%	48.5%
loadgenerator	33	37	41	43	4.9%	30.3%
paymentservice	38	44	46	49	6.5%	28.9%
Productcatalog	37	40	45	50	11.1%	35.1%
Recommendation	33	37	41	44	7.3%	33.3%
shippingservice	30	33	35	37	5.7%	23.3%

HOC-Detector 基于 SPADE 实现，带来的额外增量开销低于 9%，本文认为这是可以接受的。

HOC-Detector 的总体开销包括 SPADE 原有的开销和 HOC-Detector 通过内核模块提取进程生命周期中观测点的开销。通过和 Base 的开销数值进行比较，HOC-Detector 引入的总体开销平均增加 37.3%，对容器的单次启动开销增加 0.75%。经过分析可发现，增加的开销主要来自 SPADE，而不是 HOC-Detector 通过内核模块获取进程运行时信息的开销。

5 结束语

针对利用内核漏洞的容器逃逸攻击，本文提出了一种容器逃逸检测技术 HOC-Detector，通过在内核中监视进程的行为和提取关键进程属性信息，增强了系统对攻击行为的实时检测能力，降低了信息捕捉时延。其次，HOC-Detector 基于进程行为信息构建进程起源图，将进程关键的属性信息作为观测点，对容器进程的全生命周期进行异构观测。最后，HOC-Detector 通过检测容器中的进程行为是否有权限提升来判断当前进程是否在实施容器逃逸攻击。实验结果表明，HOC-Detector 能成功检测到直接逃逸和间接逃逸攻击，并且能应对不同的攻击方式。同时，经过本文的性能开销测试，HOC-Detector 对容器单次启动增加的开销约为 0.75%，不会对整个系统造成较大影响。目前，HOC-Detector 还存在一些不足之处。首先，HOC-Detector 仅使用 Docker 容器下的内核漏洞逃逸，尚未对其他容器引擎进行测试。其次，针对提取进程行为、属性信息的内核模块，本文尚未涉及对其安全性的讨论。这些都将作为笔者未来的工作进一步深入研究。

参考文献：

[1] SOFIA Z. Container technologies[J]. Hypatia, 2000, 15(2): 181-201.

[2] BABU S A, HAREESH M J, MARTIN J P, et al. System performance evaluation of Para virtualization, container virtualization, and full virtualization using xen, OpenVZ, and XenServer[C]//Proceedings of 2014 Fourth International Conference on Advances in Computing and Communications. Piscataway: IEEE Press, 2014: 247-250.

[3] BERNSTEIN D. Containers and cloud: from LXC to docker to Kubernetes[J]. IEEE Cloud Computing, 2014, 1(3): 81-84.

[4] MARATHE N, GANDHI A, SHAH J M. Docker swarm and Kubernetes in cloud computing environment[C]//Proceedings of 2019 3rd International Conference on Trends in Electronics and Informatics (ICOEI). Piscataway: IEEE Press, 2019: 179-184.

[5] SULTAN S, AHMAD I, DIMITRIOU T. Container security: issues, challenges, and the road ahead[J]. IEEE Access, 2019, 7: 52976-52996.

[6] COMBE T, MARTIN A, DI PIETRO R. To docker or not to docker: a security perspective[J]. IEEE Cloud Computing, 2016, 3(5): 54-62.

[7] SALAMERO J. Kubernetes runtime security with falco and sysdig[R]. 2019.

[8] JIAN Z, CHEN L. A defense method against docker escape attack[C]//Proceedings of the 2017 International Conference on Cryptography, Security and Privacy. Piscataway: IEEE Press, 2017: 142-146.

[9] ROSEN R. Resource management: Linux kernel namespacess and cgroups[R]. 2013.

[10] BACARELLA M. Taking advantage of Linux capabilities[R]. 2002.

[11] ROSEN R. Namespacess and cgroups, the basis of Linux containers[R]. 2016.

[12] LIN X, LEI L G, WANG Y W, et al. A measurement study on linux container security: attacks and countermeasures[C]//Proceedings of the 34th Annual Computer Security Applications Conference. New York: ACM Press, 2018: 418-429.

[13] COOK K. Kernel address space layout randomization[R]. 2013.

[14] CORBET J. Supervisor mode access prevention[R]. 2012.

[15] WILFAHRT N. Dirtycow vulnerability details[R]. 2016.

[16] CORBET J. On vsyscalls and the vDSO[R]. 2011.

[17] MASON J, SMALL S, MONROSE F, et al. English shellcode[C]//Proceedings of the 16th ACM Conference on Computer and Communications Security. New York: ACM Press, 2009: 524-533.

[18] BONGARD M, ILLI D. Reverse shell via voice[D]. Zurich: HSR Hochschule für Technik Rapperswil, 2019.

[19] RANDAZZO A, TINNIRELLO I. Kata containers: an emerging architecture for enabling MEC services in fast and secure way[C]//Proceedings of 2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS). Piscataway: IEEE Press, 2019: 209-214.

[20] YOUNG E G, ZHU P, CARAZA-HARTER T, et al. The true cost of containing: a gVisor case study[C]//Proceedings of 11th USENIX Workshop on Hot Topics in Cloud Computing. Berkeley: USENIX Association, 2019: 1-16.

[21] MOREAU L, FREIRE J, FUTRELLE J, et al. The open provenance model: an overview[C]//Provenance and Annotation of Data and Processes. Berlin: Springer, 2008: 323-326.

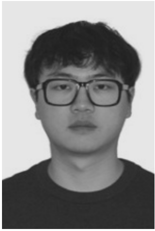
[22] MORISSON B. Analysis of the linux audit system[D]. England: Royal Holloway, University of London, 2015.

[23] GEHANI A, TARIQ D. SPADE: support for provenance auditing in distributed environments[C]//ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing. Berlin: Springer, 2012: 101-120.

[24] LORCZAK P R, CAGLAYAN A K, ECKHARDT D E. A theoretical

investigation of generalized voters for redundant systems[C]//Proceedings of the Nineteenth International Symposium on Fault-Tolerant Computing. Digest of Papers. Piscataway: IEEE Press, 1989: 444-451.
[25] NETZER R H B, MILLER B P. What are race conditions? [J]. ACM Letters on Programming Languages and Systems, 1992, 1(1): 74-88.
[26] FÁBREGA F J T, JAVIER F, GUTTMAN J D. Copy on write[R]. 1995.
[27] CHEN X, IRSHAD H, CHEN Y, et al. CLARION: sound and clear provenance tracking for microservice deployments[C]// Proceedings of 30th USENIX Security Symposium. Berkeley: USENIX Association, 2021: 3989-4006.
[28] GOOGLE. Google microservice demo: online boutique[R]. 2019.

[作者简介]



张云涛（1993− ），男，山西晋城人，北京邮电大学博士生，主要研究方向为网络安全、二进制脆弱性分析等。

方滨兴（1960− ），男，江西万年人，博士，中国工程院院士，北京邮电大学教授，主要研究方向为计算机体系结构、计算机网络、信息安全。

杜春来（1975− ），男，河北保定人，博士，北方工业大学副教授，主要研究方向为网络安全、恶意代码分析等。

王忠儒（1986− ），男，山东烟台人，博士，中国网络空间研究院高级工程师，主要研究方向为人工智能、网络安全。

崔志坚（1998− ），男，河北保定人，北方工业大学硕士生，主要研究方向为网络安全、漏洞挖掘。

宋首友（1989− ），男，云南昆明人，北京邮电大学博士生，主要研究方向为网络安全。