

Docker 组件间标准输入输出复制的 DoS 攻击分析

周天昱^{1,2}, 申文博², 杨男子³, 李金库³, 秦承刚⁴, 喻望⁴

(1. 浙江大学 NGICS 大平台, 浙江 杭州 310027; 2. 浙江大学网络空间安全学院, 浙江 杭州 310027;
3. 西安电子科技大学网络与信息安全学院, 陕西 西安 710071; 4. 蚂蚁科技集团股份有限公司, 浙江 杭州 310000)

摘要: 近年来, Docker 技术因其部署灵活、可扩展性强, 获得了大规模应用。Docker 采用模块化设计, 在降低开发和维护的复杂性的同时引入了针对组件间通信的拒绝服务 (DoS) 攻击。在 Docker 容器内频繁进行 stdout 输出会引起 Docker 组件消耗大量 CPU, 造成 DoS 攻击。经过分析, 可发现容器实例中的 stdout 输出会触发 Docker 各个组件的 goroutine, 进行频繁输出复制。为系统化地找出可被 DoS 攻击的 goroutine 创建的路径, 提出使用静态分析的方法来分析 Docker 各组件, 设计并实现了 Docker 组件静态分析框架, 最后在 Docker 上进行了测试, 成功分析得到了 34 条此类路径, 其中 22 条路径经验证, 可成功被动态触发。

关键词: 容器; Docker 组件; DoS 攻击; 静态分析

中图分类号: TP393

文献标识码: A

doi: 10.11959/j.issn.2096-109x.2020074

Analysis of DoS attacks on Docker inter-component stdio copy

ZHOU Tianyu^{1,2}, SHEN Wenbo², YANG Nanzi³, LI Jinku³, QIN Chenggang⁴, YU Wang⁴

1. Zhejiang University NGICS Platform, Hangzhou 310027, China
2. School of Cyber Science and Technology, Zhejiang University, Hangzhou 310027, China
3. School of Cyber Engineering, Xidian University, Xi'an 710071, China
4. Ant Financial Services Group, Hangzhou 310000, China

Abstract: In recent years, Docker has been widely deployed due to its flexibility and high scalability. However, its modular design leads to the DoS attacks on inter-component communication. A new DoS attack that outputs to stdout, causing high CPU usages among different Docker components. Analysis shows that the stdout output triggers the goroutines of Docker components. To find all goroutines setup paths, using the static analysis method to analyze the Docker components systematically was proposed. A static analysis framework was designed and implemented,

收稿日期: 2020-07-01; 修回日期: 2020-09-24

通信作者: 申文博, shenwenbo@zju.edu.cn

基金项目: 浙江省引进培育领军型创新创业团队 (2018R01005); 陕西省重点研发计划基金 (2019ZDLGY12-06); 中央高校基本科研业务费专项基金 (浙江大学 NGICS 大平台)

Foundation Items: Leading Innovative and Entrepreneur Team Introduction Program of Zhejiang Province (2018R01005), The Key R&D Program of Shaanxi Province (2019ZDLGY12-06), Fundamental Research Funds for the Central Universities (Zhejiang University NGICS Platform)

论文引用格式: 周天昱, 申文博, 杨男子, 等. Docker 组件间标准输入输出复制的 DoS 攻击分析[J]. 网络与信息安全学报, 2020, 6(6): 45-56.

ZHOU T Y, SHEN W B, YANG N Z, et al. The analysis of DoS attacks on Docker inter-component stdio copy[J]. Chinese Journal of Network and Information Security, 2020, 6(6): 45-56.

and evaluated on Docker source code. The results show that static analysis framework finds 34 paths successfully, while 22 of them are confirmed by runtime verification.

Key words: container, Docker components, DoS attack, static analysis

1 引言

近年来,以 Docker 为代表的容器技术因其部署灵活,可扩展性强而获得了广泛的应用。在具体实现上, Docker 采用模块化设计,根据功能分为不同的模块(又称组件),包含 Docker 命令行、Docker 守护进程、容器守护进程以及 runc 等组件。这些组件实现了功能上的紧密聚合,各个组件间采用文件或者 socket 通信,实现了容器的快速创建和灵活管理。Docker 的模块化设计使各个模块可以独立开发和设计,降低了开发和维护的复杂性。此外,模块化设计提升了 Docker 的可扩展性,如 gVisor 使用 runsc (sand-boxed container) 替换 runc,提升容器安全性的同时,与上层 Docker 的接口实现了无缝结合。

但是, Docker 的模块化设计引入了利用组件间通信进行拒绝服务(DoS)攻击的可攻击面。本文发现使用 printf 在容器实例内持续输出数据到 stdout,可以引发 Docker 组件的高 CPU 消耗,造成对宿主机 CPU 资源的 DoS 攻击。其原因是 Docker 各个组件间通过自动触发的 goroutine 不断对 stdio 进行复制,消耗大量 CPU,导致 DoS 攻击。然而此类 DoS 攻击引起的 CPU 消耗属于不同的 Docker 组件进程,而这些组件进程,被多个容器实例所共享,不属于单个容器实例,因此无法利用 cgroups 进行限制。

为了找出所有可被 DoS 攻击的路径,本文提出使用静态分析来系统化地找出所有具有 stdio 复制的 goroutine,用 LLVM 编译器分析 Docker 各组件,以负责容器实例的 stdio 复制的函数为特征,找到各组件内所有能创建 goroutine 并调用特征函数的路径,然后跨组件将这些路径连接起来,最终找到所有能触发这些路径的 Docker 命令(用户可通过在终端输入 Docker 命令和 docker-cli 交互并触发这些路径)。

要实现 Docker 组件静态分析框架需要解决两个技术挑战。

第一,在分析 Docker 组件 LLVM IR 中的特征函数调用路径时,由于组件使用 Go 语言编写并大量使用了接口(interface),而在 IR 中一般以间接调用的方式来实现接口成员函数的调用,因此无法通过分析调用语句获得接口成员函数的目标函数,从而导致大量路径分析在接口成员函数处断裂,严重影响分析效果。为了解决该问题,本文采用基于数据流分析和函数类型匹配的方法,找到了接口成员函数的具体实现,并建立了接口成员函数调用表辅助路径分析。

第二, LLVM 编译器只能分析单个组件内的函数调用路径,不同组件之间的路径无法通过组件 LLVM IR 的分析得到,但用户只能通过 Docker 命令和 docker-cli 以及后续的各个组件进行交互,因此最终路径应为从 Docker 命令开始到组件内创建 goroutine 结束的跨组件路径,然而 LLVM 只能分析单个组件内的调用路径,不同组件之间的路径无法通过组件 IR 分析得到。为了解决该问题,本文提出了跨组件路径连接技术,通过分析 Docker 组件间通信的协议(如 HTTP、gRPC 和 ttrpc)和相关函数,利用特征字段和函数的匹配成功连接了不同组件的路径。

本文的贡献如下。

1) 本文针对 Docker 组件间 stdio 复制设计了一种新的 DoS 攻击方法,并对该攻击进行了分析,分析了该攻击成功的原因,发现负责 stdio 复制的 goroutine 会被该攻击不断触发从而消耗大量 CPU。

2) 本文提出并实现了 Docker 组件静态分析框架,系统化地找出所有可能被 DoS 攻击的 goroutine 创建路径。该分析框架包括接口成员函数解析,使用 Pass 分析各组件内调用路径和跨组件连接路径等新技术。

3) 本文在 Docker 上测试了静态分析框架,成功找出了各组件内创建 goroutine 的路径,并通过路径连接分析出可被 Docker 命令触发的路径总计 34 条,其中 22 条通过动态验证确定可被触发。

2 相关工作

Dockers 使用 cgroups (control groups) 进行硬件资源的限制^[1]。cgroups 是 Linux 内核提供的用于监控和限制进程资源的工具^[2]。cgroups 使用控制器 (controller) 来控制具体的计算资源, 包括用于限制 CPU 使用量的 CPU 控制器, 限制内存使用的 memory 控制器, 限制块设备访问速率的 blkio 控制器等。用户通过访问 cgroups 提供的虚拟文件系统 cgroupfs, 以文件操作的方式来访问和创建 cgroup。每个 cgroup 都有对应的资源控制器设置, 将进程添加进具体的 cgroup 即可限制该进程以及它的后续进程的资源访问。

然而, Docker 虽然可以限制容器实例对计算资源的访问, 恶意攻击者仍然可以在容器实例内部对宿主操作系统上的计算资源进行 DoS 攻击。Gao 等^[3]在 2019 年提出可以在容器实例内部触发宿主操作系统中进程或线程对 CPU、I/O 的消耗, 从而达到 DoS 的攻击效果。这些被触发的进程或线程并非由容器实例直接创建, 不受容器实例所在 cgroup 的限制, 且它们所在的 cgroup 通常不设置资源使用上限, 因此可以消耗大量计算资源。本文将这种间接对计算资源进行 DoS 攻击的方法分为 4 类: ①触发内核线程的工作流, 使其消耗大量资源; ②使内核线程创建子进程, 该子进程与内核线程同属一个 cgroup, 由于子线程消耗大量资源; ③触发当前操作系统中的服务进程消耗资源; ④利用中断上下文来消耗资源。文献[3]对每一种类型的攻击都列举了攻击实例, 但并未阐述是如何找到这些实例的, 同时缺乏系统性查找全部 DoS 攻击的方案。

Docker 的安全问题一直是学术界和工业界关注的热点。Bui^[4]分析了 Docker 容器的隔离性, 以及对 Linux 安全特性的利用。Gupta^[5]比较了 Docker 依赖的 Linux Container 技术和虚拟机技术在安全性上的差别。Bacis 等^[6]对 Dockerfile 进行扩展, 使用户在能够生成镜像时指定 SELinux^[7]规则来加强容器安全性。Combe 等^[8]总结了不正确的 Docker 配置可能导致的安全漏洞。Chelladhurai 等^[9]探讨了 Docker 中 DoS 的攻击和防护措施。Shu 等^[10]探究了 Docker Hub 存在的安

全性漏洞。Yasrab^[11]总结了目前 Docker 容器存在的安全风险, 并提出安全性更强的 Docker 容器部署方案。Jian 等^[12]总结了 Docker 容器逃逸攻击的方法和特点, 并提出了一种基于 namespaces^[13]状态检查的防御方法。王鹃等^[14]利用可信计算的相关技术设计了 Docker 安全防护模块, 保护容器和镜像不被篡改并监控容器内部进程。Lin 等^[15]筛选了 223 个能够在 Linux 操作系统上进行攻击的有效攻击方法, 并从中挑选出 88 个典型的攻击样例在 Docker 容器当中进行测验, 最终发现有 50 个攻击方法可以进行有效的攻击, 包括信息窃取、远程控制、DoS 和提权攻击。

现有工作基本集中在 Docker 容器本身的安全性研究上, 并未对 Docker 各组件通信过程中存在的漏洞进行分析和防护。

3 Docker 组件介绍

Docker 运行容器实例需要依赖多个组件的协同工作, 各个组件通过 HTTP、gRPC 和 ttrpc 技术进行通信, 发送请求与回复, 如图 1 所示。

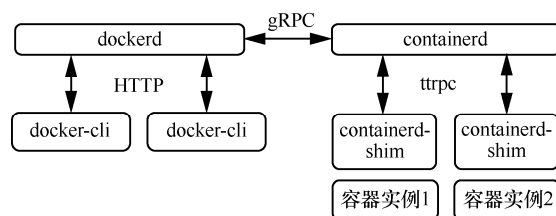


图 1 Docker 各组件关系
Figure 1 Relationships of Docker components

1) docker-cli 是 Docker 为用户提供的可以和 dockerd 进行交互的工具。用户可以在终端输入 Docker 命令, 每次执行命令都会创建一个 docker-cli 与之对应, docker-cli 会根据用户输入的命令内容, 将参数打包成 REST 格式的 HTTP 请求, 发送到 dockerd。

2) dockerd 是 Docker 的管理进程, 负责接收所有符合 Docker Engine API 的请求 (如 docker-cli 发送的 HTTP 请求), 并将容器相关的操作打包为 gRPC 请求发送给 containerd。宿主操作系统中同一时间只能运行一个 dockerd 进程, 它在初始化 Docker 服务的时候被创建, 并一直运行在后台。dockerd 进程与 docker-cli 进程是一对多的关系, 同一个 dockerd 进

程同时为所有的 docker-cli 进程提供服务。

3) containerd 负责容器实例的创建、监控和结束, 容器的网络和命名空间, 以及容器镜像的传输和存储。containerd 对外开放 gRPC API, 默认通过监听套接字来和 dockerd 进行交互, 在接收到 dockerd 的 gRPC 请求后, containerd 分析请求内容, 根据内容的不同 (如镜像处理或者容器实例创建) 将其分配到对应的服务中。containerd 进程是在初始化 containerd 服务的时候被创建并一直在后台运行, 该进程与 dockerd 进程是一对一的关系。containerd 与 containerd-shim 是一对多的关系, 负责所有 containerd-shim 进程的创建和管理, 并使用 ttrpc 协议进行通信。

4) containerd-shim 负责使用 runc 创建并启动容器实例, 并在启动完成 runc 退出后接管容器实例, 同时负责容器实例的标准输入输出以及打开文件, 从而在 containerd 或 dockerd 意外退出的时候保证容器实例的正常运行。containerd-shim 是容器实例进程的父进程, 每个容器实例都与一个 containerd-shim 进程一一对应。

4 DoS 攻击分析

本文设计了一种针对 Docker 组件的 DoS 攻击方法。受到 Gao 等^[3]利用宿主操作系统的 journald 进程进行 DoS 攻击的启发, 本文尝试了多种在容器实例内可能会触发宿主操作系统进行日志记录的操作, 发现当使用 printf 在容器实例内持续输出数据到 stdout 时, 会引发 Docker 各组件消耗大量 CPU, 从而造成 DoS 攻击。本节首先介绍 DoS 攻击的设置, 然后给出攻击结果, 最后对攻击成功的原因进行详细分析。

4.1 攻击设置

本文在 Docker 上测试了该 DoS 攻击。测试使用的 Docker 版本为 19.03.9, 宿主机操作系统为 Ubuntu 18.04.4 LTS, 内核版本为 4.15.0, 物理主机型号为 Dell Opti-Plex-7060, CPU 型号为 Intel i7-8700 (6 核 12 线程), 内存大小为 16 GB。

本文使用 docker run 命令创建并启动容器实例, 命令的参数包括: ①-it 参数, 用于开放容器实例的 stdin 并分配 tty 设备用于终端交互; ②--cpuset-cpus="0"参数, 用于限制容器实例运行

在单个 CPU 核上, 从而该容器实例中运行的所有进程占用 CPU 至多为单核 100% (宿主机 CPU 包含 12 核, 总 CPU 资源为 1200%); ③容器镜像名称, 本文使用 Docker Hub 中的 Ubuntu 官方镜像来创建容器; ④在容器中运行的命令, 本文指定在容器中运行攻击程序 (使用 printf 向 stdout 持续输出数据)。除了以上参数, 本文还通过 --log-driver 参数来控制 Docker 容器日志记录其是否启用, 并测试了在启用日志和禁用日志的配置下 DoS 攻击的效果。

4.2 攻击结果

测试结果显示该 DoS 攻击可以成功触发 Docker 组件消耗大量 CPU。图 2 展示了在 DoS 攻击中, Docker 各组件的 CPU 使用率, 可以看出在启用日志和禁用日志两种配置下的 Docker 各个组件总的 CPU 使用率分别为 607.1% 和 703.5% (宿主机总 CPU 资源为 1200%), 远高于 100% 的容器实例 CPU 限制。在启用日志记录的配置下, dockerd 进程的 CPU 使用率最高 (240.7%), 这是因为 dockerd 不仅将容器的 stdout 传递给 docker-cli, 还将这些数据记录到日志文件中。禁用日志记录后, 由于不再进行日志文件的读写, 反而加快了 Docker 组件对容器 stdout 复制的速度, 从而使各组件的 CPU 使用率总和升高。

4.3 原理分析

为了分析在 DoS 攻击测试中 Docker 组件大量消耗 CPU 的具体原因, 本文使用 pprof 动态分析了 Docker 各组件内函数在被攻击时的 CPU 使用量, 结果显示 docker-cli、dockerd 和 containerd-shim 内 CPU 占用率最高的函数分别为 io.Copy、io.CopyBuffer 和 io.CopyBuffer, 本文称其为特征函数, 进一步分析之后发现这些特征函数都位于特殊的、可被反复触发的 goroutine 中。goroutine 是一种轻量级的线程^[16], 与其他线程并发运行。一个 Go 应用程序通常同时运行多个执行不同任务的 goroutine, 这些 goroutine 可以被自动触发。

根据 pprof 中统计函数的 CPU 占用率和调用关系, 发现在基于 printf 的 DoS 攻击中, Docker 不同组件中的 goroutine 被攻击程序不断触发, 其内部的特征函数 (docker-cli 中的 io.Copy, dockerd

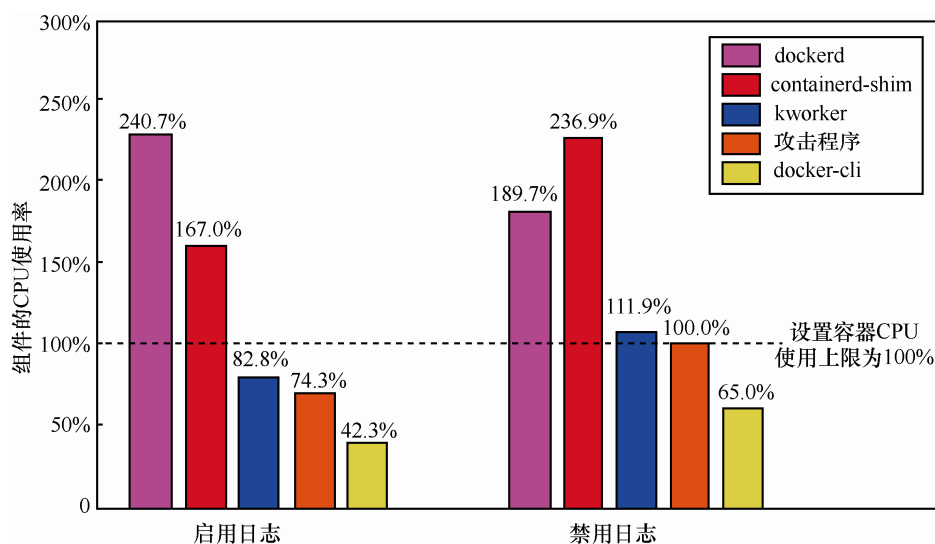


图2 DoS 攻击中各组件 CPU 使用率
Figure 2 The CPU utilization of components in the DoS attack

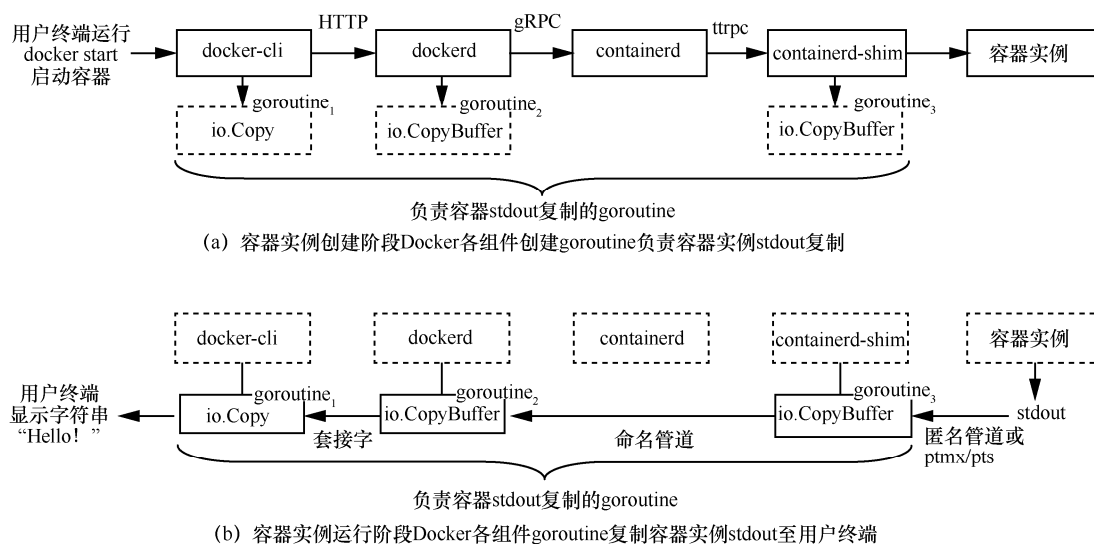


图3 Docker 组件复制容器实例 stdio 原理分析 (以 stdout 为例)
Figure 3 Docker components copy container instance's stdio (stdout for example)

和 containerd-shim 中的 io.CopyBuffer) 频繁地进行数据读写, 导致消耗大量 CPU, 使 DoS 攻击成功。在此基础上, 本文对该 goroutine 进行了分析, 发现该 goroutine 中的特征函数被用于复制容器实例的 stdout, 一旦容器实例产生 stdout 输出, 该 goroutine 中的特征函数就会被触发, 对 stdout 中的输出进行复制, 因此攻击者可通过在容器实例内部持续输出 stdout 触发各组件 goroutine 中的特征函数频繁地进行数据读写, 达到对组件的 DoS 攻击。由此可见, 利用组件间 stdio 复制进行 DoS 攻击需满足两个条件: ①有 goroutine 被创建; ②goroutine 中使用特征

函数 (io.Copy 或 io.CopyBuffer) 进行 stdio 复制。正是 Docker 组件进程中存在此类 goroutine, 攻击者才能对其进行 DoS 攻击。

通过分析 Docker 各组件, 本文发现若要复制容器实例的 stdout, 就需要在容器实例初始化阶段创建新的 goroutine, 并在其内部调用特征函数进行容器实例 stdout 输出的复制。图 3 分别展示了 Docker 各组件在容器实例创建阶段和容器实例运行阶段对 goroutine 和 stdout 复制的操作。当用户在终端执行 docker start 命令时, Docker 各个组件协同创建新的容器实例, 同时创建 goroutine

并在其内部调用特征函数负责复制容器实例的 stdout，本文将 docker-cli、dockerd 和 containerd-shim 创建的用于复制容器实例 stdout 的 goroutine 分别称为 goroutine₁、goroutine₂ 和 goroutine₃（goroutine₁ 内调用 io.Copy，goroutine₂ 和 goroutine₃ 内调用 io.CopyBuffer），如图 3(a)所示。

容器实例运行阶段，Docker 各组件采用套接字、命名管道、匿名管道和 ptmx/pts 设备对容器实例的 stdout 进行中继，如图 3(b)所示。containerd-shim 根据容器实例初始化的配置参数（如 -it 或 -d 等）来决定和容器实例的通信方法：如果使用 -it 参数，则使用 ptmx/pts 设备进行交互，容器实例将 stdout 写入 pts 设备后，containerd-shim 从 ptmx 设备中读出数据；若使用 -d 参数，containerd-shim 与容器实例通过匿名管道进行通信。containerd-shim 把容器实例的 stdout 通过命名管道传递给 dockerd，dockerd 将获得的数据通过套接字转发给 docker-cli。

容器实例 stdout 未产生数据时，特征函数处于阻塞状态。在运行阶段，容器实例内部产生 stdout 后，如 printf DoS 攻击中的 Hello 输出，containerd-shim 内 goroutine₃ 中的 io.CopyBuffer 函数被触发，先读出容器实例的 stdout（通过 ptmx 设备或匿名管道），再写入和 dockerd 共享的命名管道，如图 3(b)所示，进而触发 dockerd 中的 goroutine₂ 将数据写入和 docker-cli 通信的套接字中，最终触发 docker-cli 中负责复制 stdout 的 goroutine₁，把数据输出到用户终端上，用户即可观察到该输出，即字符串“Hello！”。综上可知，如果在容器实例创建阶段，Docker 组件创建了负责复制 stdio 的 goroutine，攻击者可以在容器运行阶段，通过不断输出到 stdout，触发各组件 goroutine 中的特征函数不断地进行数据读写，从而大幅提高各组件的 CPU 使用率，造成对于宿主机 CPU 资源的 DoS 攻击。

目前常用的资源限制技术 cgroups 无法通过限制 CPU 使用量抵御针对 Docker 组件间 stdio 的 DoS 攻击。cgroups 限制 CPU 使用量往往只针对容器实例进程，对其他 Docker 组件则不作限制，因此组件内 goroutine 可消耗的 CPU 资源并无上限。更大的问题是，所有的容器实例共享同一个 dockerd 进程，因此现有 cgroups 技术无法区分、

限制单个容器实例所引发的 dockerd 的 CPU 消耗。所以现有技术无法通过限制 CPU 来防护针对 Docker 组件间 stdio 的 DoS 攻击。

5 分析框架设计与实现

从攻击原理分析中，可知在容器实例创建阶段，Docker 各个组件创建的使用特征函数对 stdout 进行复制的 goroutine 会引发 DoS 攻击，而现有的 cgroups 技术并不能防护此类 DoS 攻击，因此本文提出一个 Docker 组件的静态分析框架，使用静态分析方法来找出所有能创建此类 goroutine 的路径，然后在不同组件间分析路径的连接，最终找到所有能触发此类路径的 Docker 命令。

本文提出的 Docker 组件调用路径静态分析框架如图 4 所示，使用 Docker 各个组件的源代码作为输入。分析框架首先使用 gollvm 将 Docker 各组件源代码编译为 LLVM IR，然后对每个组件 IR 中的接口成员函数进行解析，为所有接口成员函数的间接调用语句建立调用表，进而利用特征函数（docker-cli 对应 io.Copy，dockerd 和 containerd-shim 对应 io.CopyBuffer）逐个分析各个组件，得到每个组件内部创建 goroutine 并调用特征函数的路径，最后使用 Docker 组件间通信协议对不同组件内的路径进行连接，得到从 Docker 命令输入该组件创建 goroutine 的完整路径。

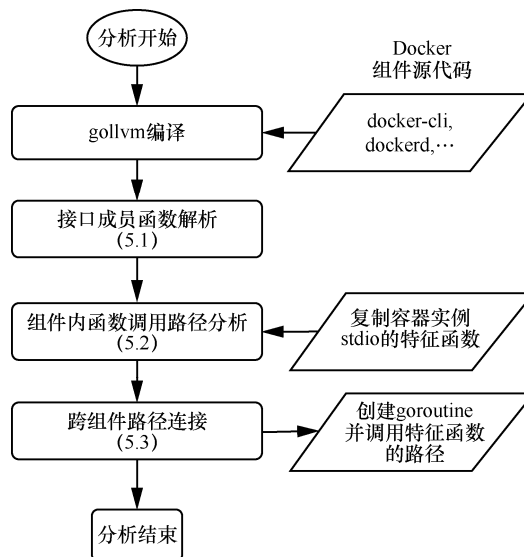


图 4 Docker 组件函数调用路径静态分析框架
Figure 4 Static analysis framework for function call paths in Docker components

虽然基本思路简单,但实现该分析框架需要解决两个技术挑战。

第一,在分析 Docker 调用路径时,需要构建完整的程序控制流图(CFG, control flow graph),然而由于 Docker 组件使用 Go 语言编写,其内大量使用了接口,对接口成员函数的调用最终通过间接调用来实现,因此无法由调用语句分析得到目标函数,导致无法形成完整的调用路径,无法构建完整的程序控制流图。为了解决该问题,本文设计了接口成员函数解析方法,利用数据流分析和函数类型匹配找到了接口成员函数的具体实现,进而建立接口成员函数调用表,使在进行组件分析时可以得到完整路径。

第二,由于用户只能通过 Docker 命令和 docker-cli 以及后续的各个组件进行交互,因此最终路径应为从 Docker 命令开始到组件内创建 goroutine 结束的跨组件路径,然而 LLVM 只能分析单个组件内的调用路径,不同组件之间的路径无法通过组件 IR 分析得到。为了解决该问题,本文提出了跨组件路径连接方法,根据组件两两交互时使用的通信函数和对应的特征字段,通过特征字段和函数的匹配完成不同组件的路径连接。

5.1 接口成员函数解析

在对 Docker 各个组件进行分析时,需要通过分析函数的调用关系来构建完整的程序控制流图,后续可以在控制流图进行路径遍历,进而找

到完整的调用路径。因此完整的程序控制流图对后续分析至关重要。在分析 Docker 组件时,由于 Docker 组件使用 Go 语言编写,大量使用接口,而接口成员函数的调用往往通过间接调用实现。无法通过解析调用语句得到目标函数,这导致大量的路径分析在接口成员函数处断裂,程序控制流图不完整,严重影响分析效果。

为了解决这个问题,本文提出结合数据流分析和函数类型来对接口成员函数的目标函数进行解析。图 5 以一个简单的 Go 语言程序为例,介绍一个接口成员函数调用的全过程。图 5(a)中定义了接口 I (第 3~5 行),该接口内有两个成员函数 Get 和 Set;同时程序中的结构体 A 实现了接口 I,包含 Get 函数和 Set 函数(图 5(a)第 8~18 行)。main 函数(第 24 行)中创建了结构体 A 的实例 a,并将其作为参数传入函数 f(第 20 行)。函数 f 中调用了接口 I 的实例 i 的成员函数 Set,即 a.Set。图 5(b)展示了源代码编译出的 LLVM IR,本文将其对接口成员函数调用实现分为 3 步:首先,创建全局变量 @pimt...main.A 存放所有结构体 A 实现的接口 I 的成员函数,其中结构体 A 实现的 Get 函数(@main.A.Get)为 @pimt...main.A 中的第二个成员变量,对应偏移量为 1,Set 函数(@main.A.Set)对应的偏移量为 2;其次,在接口实例被创建的函数中,如本例子中 @main.main 函数,创建 %I.0 类型的变量 %i.addr,并将全局变量 @pimt...main.A 存入 %i.addr 中偏移量为 0 的成



图 5 Go 语言 interface 在 LLVM IR 中的实现
Figure 5 Implementation of Go interface in LLVM IR

员变量中；最后，在接口成员函数的调用时（图 5(b)第 18~25 行），先从%I.0 类型的变量%i.addr 中取出其偏移量为 0 的成员变量，再从该成员变量中取它的偏移量为 2 的成员变量，最终得到的变量%.field.ld.2 中存放的是结构体 A 实现的 Set 函数（@main.A.Set）的地址。

基于以上分析，本文提出使用数据流分析和函数类型匹配的方法寻找接口成员函数的目标函数。首先，找到所有存放接口成员函数的全局变量（如图 5(b)中的@pimt...main.A），由于这类全局变量具有统一的命名规则（变量名以 imt 或 pimt 开头），因此可以通过匹配 pimt 和 imt 关键字来找到它们。在找到所有这类全局变量后，根据全局变量的初始化语句确定每个接口成员函数对应的偏移量，如@main.A.Get 和@main.A.Set 分别对应偏移量 1 和 2。然后，利用 LLVM 编译器提供的“定义-调用”链来遍历这类全局变量被 store 指令使用的位置（如图 5(b)第 12 行），通过分析 store 指令中存放对象（图中为%.field.15）的来源可知，该全局变量被存放到了某类型（图中为%I.0）变量中的特定偏移量（图中为 0）处，从而明确了类型%I.0 与全局变量@pimt...main.A 的关系。最后，对所有间接调用语句（图 5(b)第 25 行）进行分析，查找被调用变量（图中为%.field.ld.2）的来源，如果是接口成员函数的调用，则会有类似图 5(b)中第 20~23 行的语句，从类型%I.0 的变量中连续取了两次成员变量，第一次使用偏移量 0 取出了全局变量@pimt...main.A，第二次使用偏移量 2 取出接口成员函数@main.A.Set，如果有多个结构体类型实现了接口 I，则认为此处调用点可调用到所有 Set 函数的实现。

5.2 组件内函数调用路径分析

在对接口成员函数目标函数解析后，本文在各个组件的控制流图上进行分析，找出各个组件内的包含 stdio 复制的 goroutine 创建路径。本文设计并实现了组件内函数调用路径分析 Pass^[17]。该 Pass 以 Docker 组件的 LLVM IR 和组件内负责容器实例 stdio 复制的特征函数（docker-cli 中使用 io.Copy，dockerd 和 containerd-shim 中使用 io.CopyBuffer）为输入，输出该组件内会创建 goroutine 并调用特征函数的路径。

最终的 Pass 流程如图 6 所示，具体步骤如下。
①在接收到特征函数名称和 Docker 组件 LLVM IR 后，首先根据特征函数名称，如“io.Copy”或“io.CopyBuffer”，遍历组件 IR，找到和该名称对

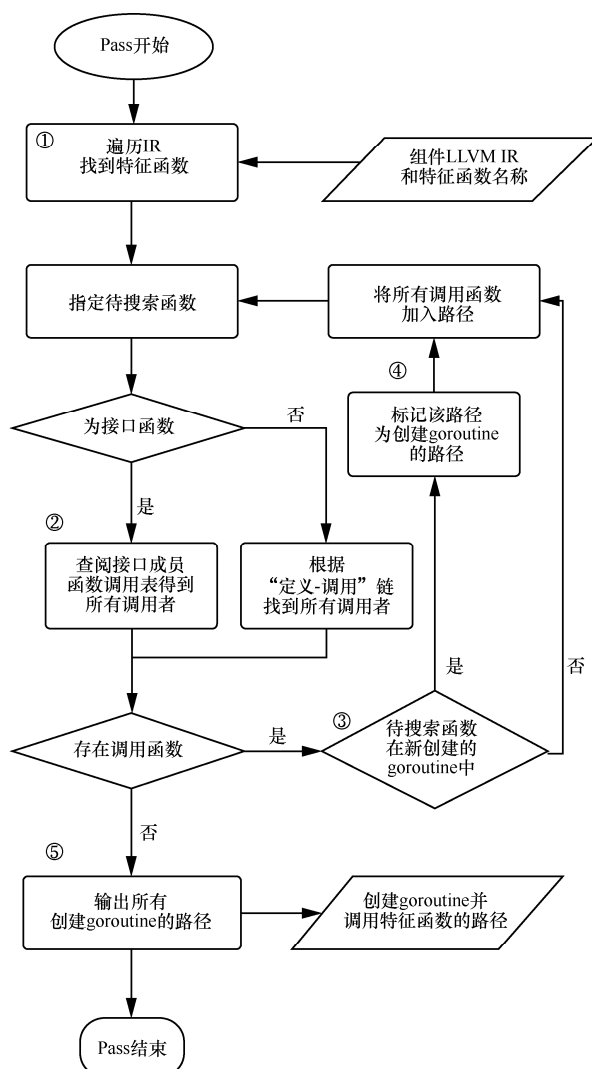


图 6 Docker 组件内函数调用路径分析 Pass 流程

Figure 6 Flow chart of Pass analysis for function call path in Docker components

应的函数（LLVM::Function），将其记录下来作为待搜索函数。②判断待搜索函数是否为接口成员函数，由于 5.1 节中解析了组件 IR 中所有的接口成员函数，因此可以由检查待搜索函数是否位于接口成员函数调用表中来判断，若在表中，说明待搜索函数为接口成员函数，直接从表中取出所有调用该待搜索函数的函数；若不在表中，则说明待搜索函数不是接口成员函数，此时使用

LLVM “定义-调用”链找到所有待搜索函数的调用处,由调用点可推断出调用函数。③若第②步没有得到任何调用函数,说明目前找到的调用路径已经完整,即可跳到第⑤步进行路径输出;否则,对所有的调用函数进行分析,判断待搜索函数是否在新创建的 goroutine 中被调用。goroutine 的创建在 LLVM IR 中是以 `@__go_go` 函数调用实现的,因此要判断待搜索函数是否在新创建的 goroutine 中被调用,只要分析在调用函数内,待搜索函数的地址是否被存入一个变量并作为 `@__go_go` 函数的参数被调用即可。④若待搜索函数处于调用函数新创建的 goroutine 中,则将该路径标记为创建 goroutine 的路径,否则不做任何操作。接着对所有的调用函数,重复步骤②至步骤④,直到无法找到任何调用函数为止。⑤对路径进行判断,如果路径被标记为创建 goroutine 的路径,说明该路径中创建了新的 goroutine 并调用了特征函数,符合分析的目标,将其输出即可。按照以上步骤最终可以得到该组件内所有创建 goroutine 并调用特征函数的路径。

5.3 跨组件路径连接

5.2 节实现的 LLVM Pass 找到的是 Docker 单个组件内的路径,本节通过对 Docker 各组件的通信方法进行分析,整理出了每两个组件之间通信函数和交互特征,使用字段匹配、函数匹配等方式成功连接各组件相关的函数调用路径。

`docker-cli` 与 `dockerd` 的交互函数如图 7 所示。`docker-cli` 通过向 `dockerd` 发送 HTTP 请求来完成和 `dockerd` 的交互,而 `dockerd` 会根据接收到的 HTTP 请求字段来调用对应的处理函数。图 7(a)展示了 `docker-cli` 中发送启动容器实例请求的函数 `ContainerStart`,该函数将特定的字段“`/containers/<containerID>/start`”作为 HTTP 请求中 `path` 字段的值,并调用 `post` 函数将该 HTTP 请求发送至 `dockerd`。图 7(b)展示了 `dockerd` 中调用 `initRoutes` 初始化 HTTP 请求 Router 的过程,该 Router 根据 HTTP 请求的 `path` 字段来指定对应的处理函数,图中的 `postContainerStart` 函数是处理容器实例启动请求的函数,函数中对应的 `path` 字段的值为“`/containers/{name:.*}/start`”。该字段与 `docker-cli` 中发送的 HTTP 请求的 `path` 字段值相对

应。由此可得,`dockerd` 中 `postContainerStart` 可以被 `docker-cli` 中的 `ContainerStart` 函数触发,从而完成跨组件调用。因此,本文通过匹配 HTTP 请求中的 `path` 字段值,来完成 `docker-cli` 与 `dockerd` 的路径连接。

`dockerd` 与 `containerd`、`containerd` 与 `containerd-shim` 的路径连接与上文介绍的方法类似。`dockerd` 通过 gRPC 和 `containerd` 进行通信,`dockerd` 向 `containerd` 发送 gRPC 请求,如图 3(a)所示,通过请求中的 `method` 字段指定相应的 gRPC 响应,如 `dockerd` 中 `tasksClient.Start` 函数将“`/containerd.services.tasks.v1.Tasks/Start`”作为 gRPC 请求中的 `method` 字段,而 `containerd` 收到该请求后,调用 `_Tasks_Start_Handler` 函数对其进行响应,并且 `_Tasks_Start_Handler` 函数同样包含上述字段,从而可以通过匹配该字段来完成 `dockerd` 和 `containerd` 的路径连接。

```
1 func (cli *Client) ContainerStart(ctx context.Context,
   containerID string, options
   types.ContainerStartOptions) error {
2     ...
3     resp, err := cli.post(ctx,
4         "/containers/"+containerID+"/start", query, nil,
5         nil)
6     ...
7 }
```

(a) `docker-cli`调用`ContainerStart`发送启动容器实例请求

```
1 func (r *containerRouter) initRoutes() {
2     ...
3     router.NewPostRoute("/containers/{name:.*}/start",
4         r.postContainersStart),
5     ...
6 }
```

(b) `dockerd`调用`initRoutes`初始化HTTP请求的Router

图 7 `docker-cli` 与 `dockerd` 的交互函数

Figure 7 Interact functions between `docker-cli` and `dockerd`

`containerd` 通过 `tttrpc` 和 `containerd-shim` 进行通信,如图 3(a)所示。`containerd` 通过请求中的 `service` 字段和 `method` 字段来指定 `tttrpc` 响应函数,如 `containerd` 调用 `shimClient.Start` 函数将“`containerd.runtime.linux.shim.v1.Shim`”作为 `service` 字段、“`Start`”作为 `method` 字段组成 `tttrpc` 请求包发送给 `containerd-shim`。`containerd-shim` 在初始化阶段调用 `RegisterShimService` 函数注册所有 `tttrpc` 服务,同时按照 `service` 字段和 `method` 字段来指定响应函数,在这个例子中,`shimClient.Start` 的响应函数为 `local.Start` 函数。因此在连接 `containerd` 和 `containerd-shim` 的路径时,可以通过 `service` 字段和 `method` 字段来匹配对应

的请求和响应函数来完成跨组件路径连接。

通过匹配 HTTP、gRPC 和 ttrpc 请求包中的特定字段,可以建立组件间的函数调用关系,从而完成 docker-cli 与 dockerd、dockerd 与 containerd、containerd 与 containerd-shim 中的调用路径连接。通过路径连接可以得到 goroutine₁、goroutine₂、goroutine₃ 对应的完整的创建路径,以及触发这些 goroutine 创建的 Docker 命令。

6 实验

本节使用 Docker 测试实现的静态分析框架原型。在测试中,成功将 Docker 各组件编译为 LLVM IR,并且使用静态分析框架原型成功地分析出了各组件内 goroutine 的创建路径,包含 docker-cli 的 goroutine₁, dockerd 的 goroutine₂ 和 containerd-shim 的 goroutine₃,然后通过跨组件路径连接得到了这些 goroutine 的创建对应应在 docker-cli 中的命令,最后验证了这些路径能否成功被触发并创建相应的 goroutine 进行容器实例的 stdio 复制。

6.1 实验设置

本文分析的 Docker 版本为 Docker CE 19.03.9,对应的 Docker 各组件版本如表 1 第 2 列所示。使用的 LLVM 编译器版本为 11.0.0git, gollvm 版本为 go1.14.2。验证路径时使用的物理设备和系统环境与第 4.1 节中介绍的相同。

表 1 Docker 各组件对应的 LLVM IR
Table 1 LLVM IR for Docker components

组件 IR (.bc)	版本	源代码行数	IR 大小/MB
docker-cli	19.03.09	76 374	473
dockerd	19.03.09	192 415	533
containerd	1.2.13	98 593	236
containerd-shim	1.2.13	7 704	27

在源代码基础上,本文需要将各组件编译为 LLVM IR。首先,设置 gollvm 为默认 Go 编译器,使用 go build 命令编译组件源码,命令中需要使用 -x-work 参数记录编译过程中的所有编译指令;然后,将编译指令中的生成对象替换为 LLVM IR 文件,再次执行,得到该组件内所有 package 对应的 LLVM IR;最后,使用 llvm-link 将所有包的

LLVM IR 链接起来,得到该组件对应的 LLVM IR。本文使用代码行数统计工具 cloc 来统计各个组件的源代码行数。实验中 Docker 各个组件的版本、源代码行数、对应的 LLVM IR 大小如表 1 所示。

6.2 实验结果

实验结果主要分为两部分:各个组件内路径统计和跨组件路径统计。Docker 各个组件路径分析结果如表 2 所示。docker-cli、dockerd 以及 containerd-shim 可创建包含 stdio 复制的 goroutine 路径分别有 12 条、71 条和 30 条。另外,containerd 经过分析不存在可被 Docker 命令触发的容器实例的 stdio 复制路径。

表 2 Docker 各组件路径分析结果
Table 2 Results of Docker components' path analysis

组件 IR (.bc)	涵盖函数	组件内路径数目/条
docker-cli	47	12
dockerd	82	71
containerd-shim	25	30
总计	154	113

跨组件路径分析结果如表 3 所示,本文实现的静态分析框架原型分析出 docker-cli 中通过命令可以创建 goroutine₁、goroutine₂ 和 goroutine₃ 的路径分别有 12 条、13 条和 9 条,总计 34 条。本文同时对所有分析出来的跨组件路径进行了动态运行验证,能够成功被用户输入的 docker-cli 命令触发并创建对应 goroutine 的路径分别为 8 条、8 条和 6 条,总计 22 条,如表 3 第 3 列所示。对比第 3 列和第 2 列,可以得出静态分析的准确率分别为 66.7%、61.5%和 66.7%。

表 3 跨组件路径分析结果
Table 3 Results of linking paths analysis across components

goroutine	docker-cli 中对应路径	经过动态运行验证成功触发的路径	准确率
goroutine ₁	12	8	66.7%
goroutine ₂	13	8	61.5%
goroutine ₃	9	6	66.7%
总计	34	22	64.7%

在验证路径时,本文发现 Docker 存在 Healthcheck 机制,当创建容器镜像时使用的 Dockerfile 中包含 HEALTHCHECK 命令时,

Docker 会定时检查以此镜像启动的容器实例是否死锁, 而检查过程中会创建复制容器实例的 stdio 的 goroutine (goroutine₂ 和 goroutine₃), 但此处的 goroutine 会在检查结束之后立刻终止, 无法被用来做 DoS 攻击。因此, 所有和该机制有关的路径 (如 docker unpause 等命令触发的路径) 都不视为验证成功的路径, 因为它们并没有主动创建 goroutine 进行 stdio 复制, 即使触发了容器实例的 Healthcheck 机制, 也无法被用来做 DoS 攻击。

在动态运行验证时可被成功触发的路径中, 有部分路径可以由同一个 Docker 命令 (命令参数也相同) 触发, 运行该命令可同时触发多个 goroutine 的创建。其中, docker start -it 命令可以同时触发 docker-cli 中 goroutine₁、goroutine₂、goroutine₃ 对应的路径, 因而在运行该命令后可同时创建这 3 种 goroutine。图 8 展示了该命令可以触发的路径。图左侧为容器实例初始化阶段, docker-cli、dockerd 和 containerd-shim 分别创建了 goroutine₁、goroutine₂ 和 goroutine₃ 具体的函数调用链。图右侧展示了容器实例运行阶段, 当有 stdout 产生时, 各组件的 goroutine 会把数据复制最终写入用户终端, 因而容器实例内部的攻击者可通过在容器实例内持续输出数据到 stdout 来进行对 Docker 组件的 DoS 攻击。可以同时触发 3 种 goroutine 创建的 Docker 命令和参数组合, 共计 6 种。

值得注意的是, 在执行一些 Docker 命令后 (如 docker run -d), 仅有 goroutine₂ 和 goroutine₃ 被创建, 当攻击者在容器实例内使用 printf 进行 DoS 攻击时, goroutine₂ 和 goroutine₃ 仍可被成功触发并消耗大量 CPU, 但此时 Docker 组件的 CPU 消耗总和小于 3 种 goroutine 都被创建时进行 DoS 攻击后的 CPU 消耗。

7 结束语

本文提出了一种针对 Docker 组件的 DoS 攻击方法, 通过在容器实例内部使用 printf 频繁输出数据到 stdout 令 Docker 组件消耗大量 CPU 造成 DoS 攻击。经过对该攻击原理的分析, 本文发现 Docker 组件在容器实例初始化阶段会创建新的 goroutine, 并在其内部调用特征函数, 如 io.Copy 或 io.CopyBuffer, 对容器实例的 stdio 进行

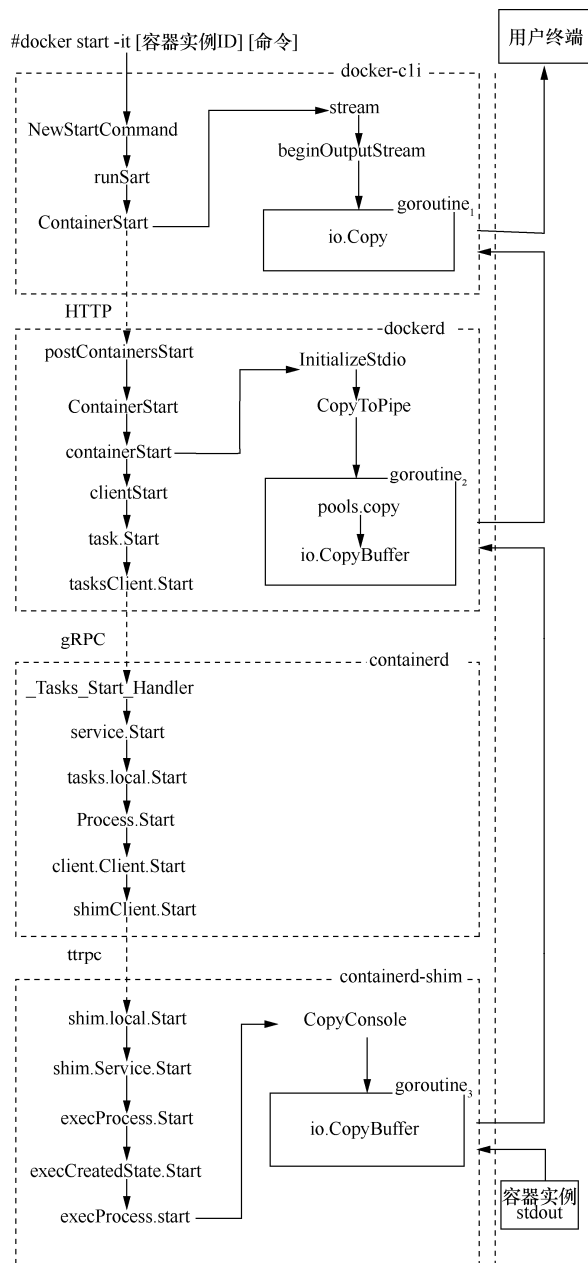


图8 Docker 组件 stdio 复制触发路径 (以 docker start 为例)
Figure 8 Docker components stdio copy trigger path (take docker start for example)

复制, 因此这些 goroutine 会被 printf DoS 攻击不断触发消耗大量 CPU 而使 DoS 攻击成功。当前 cgroups 技术并不能防御此类 DoS 攻击。

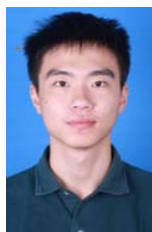
为了分析所有可能导致 DoS 攻击的路径, 本文提出使用静态分析的方法来系统化地分析 Docker 组件, 使用 LLVM 编译器分析 Docker 各组件创建 goroutine 并调用特征函数进行容器实例 stdio 复制的路径。本文设计并实现了接口成员函数解

析、组件内函数调用路径分析和跨组件路径连接等技术,实现了静态分析框架原型系统,并且在最新 Docker 版本上进行了测试。测试结果显示静态分析框架原型系统成功地找出了各组件内创建 goroutine 的路径,并通过路径连接分析出可被 Docker 命令触发的路径总计 34 条,经过动态验证,这些路径中总计 22 条可成功被 Docker 命令触发并创建对应 goroutine。

参考文献:

- [1] MERKELD. Docker: lightweight Linux containers for consistent development and deployment[J]. Linux Journal, 2014(239):2.
- [2] SEYFRIED S. Resource management in Linux with control groups[C]//Proceedings of the Linux-Kongress. 2010.
- [3] GAO X, GU Z S, LI Z F, et al. Houdini's escape: breaking the resource rein of Linux control groups[C]//Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. 2019: 1073-1086.
- [4] BUI T. Analysis of docker security[J]. arXiv preprint arXiv: 1501.02967, 2015.
- [5] GUPTA U. Comparison between security majors in virtual machine and Linux containers[J]. arXiv preprint arXiv:1507.07816, 2015.
- [6] BACIS E, MUTTI S, CAPELLI S, et al. Docker policy modules: mandatory access control for docker containers[C]//2015 IEEE Conference on Communications and Network Security (CNS). 2015: 749-750.
- [7] SMALLEY S, VANCE C, SALA-MON W. Implementing selinux as a linux security module[J]. NAI Labs Report, 2001, 1(43): 139.
- [8] COMBE T, MARTIN A, PIETRO R. To docker or not to docker: a security perspective[J]. IEEE Cloud Computing, 2016, 3(5): 54-62.
- [9] CHELLADHURAI J, CHELLIAH P R, KUMAR A. Securing Docker containers from denial of service (DoS) attacks[C]//2016 IEEE International Conference on Services Computing (SCC). 2016: 856-859.
- [10] SHU R, GU X H, ENCK W L. A study of security vulnerabilities on docker hub[C]//Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy. 2017: 269-280.
- [11] YASRAB R. Mitigating docker security issues[J]. arXiv preprint arXiv:1804.05039, 2018.
- [12] JIAN Z Q, CHEN L. A defense method against docker escape attack[C]//Proceedings of the 2017 International Conference on Cryptography, Security and Privacy. 2017: 142-146.
- [13] ROSEN R. Namespaces and cgroups, the basis of Linux containers[C]//Proceedings of the NetDev. 2016.
- [14] 王鹏, 胡威, 张雨菡, 等. 基于 Docker 的可信容器[J]. 武汉大学学报(理学版), 2017, 63(2): 102-108.
WANG J, HU W, ZHANG Y H, et al. Trusted container based on Docker[J]. Journal of Wuhan University (Natural Science Edition), 2017, 63(2): 102-108.
- [15] 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. 2018: 418-429.
- [16] PRABHAKARR, KUMARR. Concurrent programming with go[R]. 2011.
- [17] LOPES B C, AULERR. Getting started with LLVM core libraries[M]. Packt Publishing Ltd, 2014.

[作者简介]



周天昱 (1995-), 男, 浙江杭州人, 浙江大学硕士生, 主要研究方向为容器安全。



申文博 (1989-), 男, 浙江大学研究员、博士生导师, 主要研究方向为操作系统安全、容器安全、软件及系统攻防、硬件安全、程序分析。



杨男子 (1996-), 男, 陕西咸阳人, 西安电子科技大学硕士生, 主要研究方向为容器安全。



李金库 (1976-), 男, 河北保定人, 西安电子科技大学教授、博士生导师, 主要研究方向为系统与网络安全、移动安全、云计算及其安全。



秦承刚 (1983-), 男, 江苏徐州人, 主要研究方向为操作系统、系统安全。



喻望 (1986-), 男, 重庆人, 主要研究方向为 Linux 内核、安全容器技术。