

Proj118: eBPF-based Monitor for Container 中期报告

项目描述

容器技术对系统资源实现了较好的隔离, 因而在云服务生产和开发环境中广泛应用. 然而, 默认的Docker环境需要赋予容器最高权限, 带来巨大的安全风险. SECCOMP是Linux内核的特性, 开发者可以通过SECCOMP限制容器的行为, 而Docker提供的默认SECCOMP配置允许了200余项系统调用的权限, 其中大多数是容器正常服务所不需要的, 由此极大地引入了容器逃逸的安全风险.

eBPF是Linux内核中的一个执行引擎, 可以在沙盒环境中安全高效地在内核中执行自定义程序, 且无需重新编译内核

. eBPF提供了若干内核和用户空间事件的hook, 可以对特定系统调用, socket数据包执行操作, 如记录, 转发, 丢弃等, 因此适合用于进行性能分析, 网络处理, 系统观测等工具.

现拟开发一套基于eBPF技术的容器权限监控和配置工具, 在开发阶段对容器的行为进行观测和统计, 输出报表, 并尝试为相应容器生成一个最小的系统权限配置.

设计思路

观测与记录

实现一个或一组eBPF程序, 并挂载到系统内核, 实现对特定系统调用的记录, 返回到用户空间.

实现一个用户空间的数据收集和统计工具, 处理eBPF程序回传的记录, 并进行筛选整合, 生成报表.

生成安全配置

实现一个基于容器行为数据生成对应SECCOMP配置文件的工具, 实现对容器行为的最小限制.

开发计划

调研与技术选型(4.10 - 4.25)

1. 了解eBPF程序原理, 语法和能力范围.
2. 调研eBPF用户空间工具链, 对比不同语言的BPF API和处理能力.
3. 了解容器技术在内核层面的实现原理, 思考如何与eBPF的能力进行配合.
4. 了解容器技术的业务范围, 初步思考如何找到准确的最小权限集合.

开发(4.25-5.31)

1. 开发eBPF C程序, 由内核向用户空间回传进程和系统调用事件信息.
2. 开发一个单容器单线程的最小demo, 实现基本的锁定进程和报告能力.
3. 拓展观测工具的能力到单容器多线程, 实现基于cgroup或进程树的多进程追踪和报告能力.

功能提升(5.31-6.31)

1. 拓展观测工具的能力到容器集群, 对真实生产环境进行还原.
2. 设计并开发容器权限配置生成的算法, 实现权限配置推荐工具.
3. 对算法进行调优和修改, 实现较准确的最小权限集合算法.

修改和完善 (7.1-8.1)

1. 继续完善工具组合

实现思路

eBPF C程序

如果为每个syscall事件编写一个BPF程序, 那么内核中的BPF JIT虚拟机将不堪重负(当前5.14版本的Linux内核已有超过300个系统调用, 且还在增加), 并且BPF程序代码依赖于内核版本, 不适合稳定使用.

经过调研, 发现内核中存在对系统调用进入和退出的抽象追踪点 `SYS_ENTER` 和 `SYS_RET`, 该追踪点可以在所有系统调用发起/返回时触发, 经过阅读内核源代码, 发现该追踪点可以报告系统调用函数的id或返回值, 因此只需实现一个eBPF程序, 且不依赖于内核版本, 可以在使用时传入任意对应版本的syscall-id映射表. 实现检测工具与内核版本的解耦合.

通过 `SYS_ENTER` 这一内核追踪点, 使用BPF的 `PERF_OUTPUT` 帮助函数, 我们可以获取到即将触发的syscall id, 并回传给用户空间.

进程追踪和统计

使用BCC将上述BPF C代码挂载到内核并准备读取, 加载当前内核版本的syscall-id映射, 准备将读取到的syscall id转换为syscall name. 由于BPF会回传所有进程的所有系统调用信息, 我们需要在用户空间进行筛选. 使用docker-py SDK提供的能力, 抓取容器的进程列表, 使用pid信息筛选BPF记录, 并生成报告, 存入文件.

最小权限推荐

读取上一步生成的系统调用统计, 取全部出现的系统调用的集合, 按照SECCOMP配置文件语法生成配置.

技术问题

1. 内核观测点的选取

- 已解决, 使用系统调用的抽象追踪点 `SYS_ENTER`, 极大减少内核中挂载的BPF程序数量, 并简化BPF逻辑, 将复杂的筛选和统计工作由内核转移到用户空间, 避免内核的过度负载.

2. 多线程容器的处理方法

- BPF有抓取并向User Space返回cgroupID的能力, 同时可以在user space抓取cgroup: 对于一个容器内的所有进程一定共享至少一个独有cgroup. 随后通过匹配cgroup对eBPF回传的数据进行过滤. 问题: 文档和搜索引擎都没有查询到cgroup path (在bash中使用 `cat /proc/$pid/cgroup` 获得) 和cgroup id的对应关系.
- docker-py SDK可以返回进程列表, eBPF可以报告pid, 通过比对pid是否在进程号列表进行过滤. 问题: 容器内的进程随时可能发生变化, 无法第一时间将这一变化同步到filter当中, 可能会漏掉一部分eBPF回传的数据.

3. syscall最小集的正确性

- 如果仅将docker容器启动并观测, 势必会缺失一部分syscall, 例如数据库通信, error handling, 内存交换等场景下的syscall. 尝试引入单元测试+压力测试的概念: 通过特定测试用例激发容器的不同表现 (响应用户请求, 处理错误, 应对高并发情况), 来尽可能获取准确的权限最小集. 通过编写测试用例, 记录容器使用的权限的变化, 生成不同场景的权限集合.

4. 集群上的容器行为观测

开发进度和演示

已完成适用于单进程的行为监测工具的demo开发, 效果如下:

```
^C76706.711141000    b'python3'      30604  b'1'
Traceback (most recent call last):
  File "/home/curtis/raw_tp.py", line 22, in <module>
    print("%-18.9f %-16s %-6d %s" % (ts, comm, pid, nr))
KeyboardInterrupt

[+] sudo python3 raw_tp.py [INT x] 12s
/virtual/main.c:9:1: warning: non-void function does not return a value [-Wreturn-type]
}
^
1 warning generated.
      ts              comm      pid nr
76706.711646000    b'gnome-terminal-' 30317 b'228'
76706.711647000    b'gnome-terminal-' 30317 b'228'
76706.711649000    b'gnome-terminal-' 30317 b'228'
76706.711650000    b'gnome-terminal-' 30317 b'228'
76706.711651000    b'gnome-terminal-' 30317 b'228'
76706.711653000    b'gnome-terminal-' 30317 b'228'
76706.711654000    b'gnome-terminal-' 30317 b'228'
76706.711655000    b'gnome-terminal-' 30317 b'228'
76706.711657000    b'gnome-terminal-' 30317 b'228'
76706.711657000    b'python3'        30604 b'1'
```

如图所示, demo工具可以汇报产生系统调用的时间, 程序名, pid和系统调用的代码. 使用BCC编写, python程序中预留了控制监控时间/次数/程序名筛选/pid筛选/调用种类筛选等能力.

已完成Docker-py SDK容器信息获取的开发, 能够获取容器所属的cgroup路径, 子进程pid等信息用于筛选.

导师信息

与企业导师和校内导师建立了微信群, 进行过两轮技术会议, 分别沟通了项目形态和若干具体技术问题, 从导师处获取了若干文档, 学术论文等调研材料.