**bpfbox: 基于 eBPF 的简单精确进程限制** 是一种利用 eBPF（扩展伯克利包过滤器）技术实现的轻量级、灵活且高效的 Linux 进程限制机制。本文将详细讲解 bpfbox 的设计思路和实现方法，包括其动机、架构、策略语言、执行机制、性能评估以及未来的改进方向。

## 1. 设计动机

在现代 Linux 系统中，进程限制和安全性通常通过多种工具和机制（如命名空间、cgroups、SELinux、AppArmor 等）组合实现。这些现有的安全机制虽然功能强大，但存在以下问题：

* **复杂性高**：配置和管理多个安全模块需要深入的专业知识，增加了运维难度。
* **灵活性不足**：难以实现细粒度的控制，尤其是在需要动态调整策略的场景下。
* **性能开销**：复杂的安全检查可能带来显著的性能损耗。

为了解决上述问题，bpfbox 利用了 eBPF 的能力，旨在提供一种更简单、灵活且高效的进程限制解决方案。

## 2. eBPF 技术概述

eBPF 是一种在 Linux 内核中运行用户定义代码的强大机制。它允许开发者在不修改内核源代码的情况下，动态加载和执行安全、高效的程序。eBPF 的关键优势包括：

* **高性能**：由于 eBPF 程序在内核空间执行，减少了用户态和内核态之间的上下文切换开销。
* **安全性**：内核在加载 eBPF 程序前会进行严格的验证，确保其安全性和稳定性。
* **灵活性**：eBPF 支持多种钩子（如系统调用、网络事件、文件操作等），使其能够覆盖广泛的应用场景。

## 3. bpfbox 的架构设计

bpfbox 的设计目标是通过 eBPF 提供细粒度的进程限制，同时保持高性能和易用性。其架构主要包括以下几个组件：

### 3.1 策略语言

bpfbox 定义了一种简洁且易于理解的策略语言，用于描述进程的允许和禁止操作。策略语言支持以下几种规则类型：

* **文件系统访问（fs）**：控制进程对特定文件或目录的读写权限。
* **网络操作（net）**：限制进程的网络通信能力，如绑定端口、发送和接收数据等。
* **信号操作（signal）**：控制进程可以发送或接收的信号类型。
* **ptrace 操作（ptrace）**：限制进程对其他进程的调试能力。

此外，bpfbox 还支持三种指令来定义规则的行为：

* **allow**：允许匹配的操作。
* **taint**：标记进程进入受限制状态，一旦被 taint，后续操作将受策略限制。
* **audit**：记录匹配的操作到审计日志中，便于事后分析。

### 3.2 内核空间组件

bpfbox 利用 eBPF 在内核空间注入多个程序，以拦截和处理特定的系统事件。这些 eBPF 程序主要负责：

* **拦截系统调用和 LSM 钩子**：监控进程的系统调用和安全模块事件，检查是否符合定义的策略。
* **策略执行**：根据策略语言定义的规则，决定是否允许或拒绝某个操作。
* **事件记录**：在启用审计模式时，记录相关的安全事件到用户空间日志。

### 3.3 用户空间组件

用户空间的 bpfbox 组件负责以下任务：

* **策略解析与加载**：解析用户定义的策略文件，将其转换为 eBPF 可理解的格式，并加载到内核中。
* **管理 eBPF 程序**：协调 eBPF 程序的加载、卸载和更新，确保策略的实时生效。
* **日志处理**：收集内核空间记录的审计日志，并进行存储或进一步处理。

## 4. 策略语言详解

bpfbox 的策略语言设计简洁，便于用户编写和理解。以下是策略语言的主要组成部分和语法示例：

### 4.1 文件系统规则（fs）

用于控制进程对文件或目录的访问权限。语法格式为：

fs("路径", 操作)

**操作**可以是多个权限的组合，用竖线（|）分隔，如 read|write。

**示例**：

fs("/var/log/httpd/\*log", getattr|append)

fs("/srv/html/\*\*", read|getattr)

### 4.2 网络规则（net）

用于限制进程的网络操作。语法格式为：

net("协议", 操作)

**协议**包括 inet（IPv4）、inet6（IPv6）、unix、netlink 等。

**操作**可以是 bind、connect、send、recv 等。

**示例**：

net(inet, any)

net(unix, create|connect|send|recv)

### 4.3 信号规则（signal）

用于控制进程发送或接收信号的能力。语法格式为：

signal("路径", 信号类型)

**信号类型**可以是具体的信号名（如 sigterm、sigint）或类别关键字（如 fatal、nohandle）。

**示例**：

signal("/bin/httpd", check|fatal)

signal("/usr/bin/ls", sigterm|sigint)

### 4.4 ptrace 规则（ptrace）

用于限制进程对其他进程的调试能力。语法格式为：

ptrace("路径", 访问模式)

**访问模式**可以是 read、attach 等。

**示例**：

ptrace("/usr/bin/ls", read)

ptrace(child, read|attach)

### 4.5 指令（Directive）

用于定义规则的行为，如 allow、taint、audit。使用 #[] 语法进行装饰。

**示例**：

#[taint] {

net(inet, bind)

net(inet6, bind)

}

#[audit]

fs("/etc/shadow", read)

### 4.6 函数和内核函数指令（func 和 kfunc）

用于在特定的用户空间或内核空间函数上下文中应用规则。

**示例**：

#[func "check\_password"]

fs("/etc/shadow", read)

#[kfunc "sys\_open"]

fs("/etc/passwd", read)

## 5. 实现方法

### 5.1 策略加载与解析

用户通过编写 bpfbox 的策略文件，定义进程的允许和禁止操作。用户空间的 bpfbox 组件负责解析这些策略，将其转换为内核空间的 eBPF 程序能够理解和执行的形式。

### 5.2 eBPF 程序注入与管理

bpfbox 使用 eBPF 将多个程序注入到内核中，这些程序分别负责拦截不同类型的系统事件（如文件操作、网络操作、信号发送等）。每个 eBPF 程序都包含了策略检查的逻辑，根据用户定义的策略决定是否允许或拒绝某个操作。

### 5.3 事件拦截与策略执行

当被限制的进程尝试执行某个操作时，相关的 eBPF 程序会被触发，执行以下步骤：

1. **事件拦截**：eBPF 程序拦截到系统调用或 LSM 钩子事件。
2. **策略匹配**：根据事件的属性（如路径、操作类型、信号类型等），匹配用户定义的策略规则。
3. **决策执行**：根据匹配结果，决定是允许该操作、拒绝该操作，还是记录到审计日志。
4. **反馈结果**：将决策结果反馈给内核，执行相应的操作（如阻止系统调用、发送信号等）。

### 5.4 日志记录与审计

当策略包含 #[audit] 指令时，匹配的操作将被记录到审计日志中。这些日志由用户空间的 bpfbox 组件收集，并存储或进一步处理，便于安全审计和分析。

## 6. 示例：Apache HTTPD 的 bpfbox 策略

为了说明 bpfbox 的实际应用，本文提供了一个针对 Apache HTTPD（httpd）服务器的示例策略。该策略旨在限制 httpd 进程的行为，仅允许其执行必要的操作，从而减少潜在的攻击面。

**策略文件内容（简化版）：**

#![ profile "/bin/httpd"]

/\* Taint on binding to an inet or inet6 socket \*/

#[taint] {

net(inet, bind)

net(inet6, bind)

}

/\* Specify allowed network access \*/

net(inet, any)

net(inet6, any)

net(unix, create|connect|send|recv)

net(netlink, create|bind|send|recv)

/\* Allows kill (2) to check for process existence and to send fatal signals \*/

signal("/bin/httpd", check|fatal)

/\* Write to logs \*/

fs("/var/log/httpd/\*log", getattr|append)

fs("/var/log/httpd", getattr)

/\* Create PID file \*/

fs("/run/httpd/", write)

/\* Delete or modify an existing PID file if necessary \*/

fs("/run/httpd/httpd.pid", getattr|rm|write)

/\* Serve files from /srv/html/ and all subdirectories \*/

fs("/srv/html/\*\*", read|getattr)

/\* Read configuration \*/

fs("/usr/share/httpd/\*\*", read|getattr)

fs("/etc/httpd/", getattr)

fs("/etc/httpd/conf/\*\*", read|getattr)

fs("/usr/share/zoneinfo/\*\*", read|getattr)

fs("/etc/resolv.conf", read|getattr)

fs("/etc/hosts", read|getattr)

fs("/proc/sys/kernel/random/boot\_id", read)

fs("/proc/sys/kernel/ngroups\_max", read)

fs("/usr/lib/httpd/modules/\*.so", getattr|read|exec)

fs("/usr/lib/libnss \*.so.\*", getattr|read|exec)

fs("/usr/lib/libgcc\_s.so.\*", getattr|read|exec)

/\* Transition to a separate suexec policy \*/

#[ transition]

fs("/usr/bin/suexec", getattr|read|exec)

**策略解析：**

1. **配置文件路径**：策略开始于指定 /bin/httpd 作为受限的进程。
2. **Taint 规则**：当 httpd 绑定到 IPv4 或 IPv6 网络套接字时，进程被标记为 tainted，此后所有操作将受到策略的约束。
3. **网络访问**：允许 httpd 进行广泛的网络操作，包括创建、连接、发送和接收数据。
4. **信号控制**：允许 httpd 检查进程存在性并发送致命信号。
5. **文件系统访问**：
   * 允许对日志文件的获取属性和追加写入。
   * 允许创建和修改 PID 文件。
   * 允许读取配置文件和静态内容。
   * 允许执行共享库和模块文件。
6. **策略转移**：当执行 suexec 程序时，进程将转移到一个独立的策略，以进一步限制 suexec 的行为。

**比较与优势：**

与现有的 AppArmor 和 seccomp-bpf 策略相比，bpfbox 提供了更简洁和可读的策略定义，同时在性能上表现出色。生成的策略文件更短、更易于维护，而不需要依赖复杂的工具链或多个安全模块的配置。

## 7. 性能评估

为了评估 bpfbox 的性能，研究人员进行了多项基准测试，并将其与 AppArmor 进行了比较。主要评估指标包括：

* **Phoronix OSBench**：测试文件创建、进程创建、线程创建、程序执行和内存分配等操作的性能。
* **内核编译基准**：模拟高负载下的系统编译任务，评估 bpfbox 在繁忙工作负载下的性能表现。
* **Phoronix Apache 基准**：测试 bpfbox 对网络操作的影响，特别是在处理大量 HTTP 请求时的性能表现。

**测试结果总结：**

* **Phoronix OSBench：**
  + 在不受限制的情况下，bpfbox 对系统性能的影响低于 4%，在某些测试中甚至略有提升。
  + 在审计模式下，性能开销增加到约 15%，主要影响文件创建和程序启动操作。
  + 与 AppArmor 相比，bpfbox 在大多数测试中表现相当，某些情况下表现更好，尤其是在文件创建和程序启动方面。
* **内核编译基准**：
  + 在用户态和内核态 CPU 时间上，bpfbox 与 AppArmor 表现相似。
  + 在整体耗时上，bpfbox 略逊于 AppArmor，主要原因是当前版本的 bpfbox 在用户空间日志记录时未进行缓冲，导致高频事件处理时 CPU 资源消耗较大。
* **Phoronix Apache 基准**：
  + bpfbox 对 HTTP 请求的处理性能影响较小，约在 6% 到 9% 之间，与 AppArmor 相当。

**结论**：

bpfbox 在大多数情况下表现出色，性能开销与现有的安全机制（如 AppArmor）相当，且在某些操作上具有更低的开销。此外，bpfbox 的策略定义更加简洁，便于管理和维护。

## 8. 限制与改进方向

### 8.1 当前限制

尽管 bpfbox 展示了显著的优势，但仍存在一些限制：

* **递归函数调用问题**：在使用 #[func] 指令时，递归函数调用可能导致状态管理混乱，进而引发意外的访问拒绝。
* **进程初始化限制**：bpfbox 目前无法对在其运行前启动的进程进行沙箱限制，因为这些进程未关联任何策略。
* **内核版本依赖**：bpfbox 依赖于特定的 eBPF 特性（如 KRSI LSM、ringbuf map type），需要 Linux 5.8 及以上版本，这可能限制了其在一些旧系统上的部署。
* **用户空间日志性能瓶颈**：在高频事件下，用户空间的日志记录可能成为性能瓶颈，特别是在并发度高的场景下。

### 8.2 潜在改进

为了解决上述限制，未来的改进方向包括：

* **改进递归调用处理**：通过使用更复杂的状态管理机制或用户空间追踪点，解决递归函数调用带来的问题。
* **支持现有进程的沙箱化**：在启动 bpfbox 后，扫描 /proc/{pid}/exe，为已存在的进程手动关联策略，实现对现有进程的沙箱限制。
* **优化用户空间日志记录**：引入缓冲机制或批处理方式，减少日志记录对 CPU 资源的消耗，提高高频事件下的性能表现。
* **扩展策略语言功能**：增加对系统调用级别的限制，提供更细粒度的控制，进一步提升安全性。
* **跨平台支持与兼容性提升**：通过迁移到更通用的 eBPF 框架（如 libbpf CO-RE），提高 bpfbox 的跨内核版本兼容性，便于在更多系统上部署。

### 8.3 未来发展方向

bpfbox 的设计展示了 eBPF 在系统安全方面的巨大潜力。未来的研究和开发可以探索以下方向：

* **集成入侵检测与防御系统**：将 bpfbox 与入侵检测系统（IDS）或入侵防御系统（IPS）结合，根据实时检测结果动态调整策略，实现自适应的安全防护。
* **容器级别的策略控制**：扩展 bpfbox 的功能，支持对容器进行细粒度的限制和监控，提升容器安全性。
* **自动策略生成**：利用 bpfbox 的详细事件日志，开发类似于 audit2allow 的工具，自动生成和优化策略，降低手动编写策略的复杂性。
* **用户界面与管理工具**：开发图形化界面或命令行工具，简化策略的编写、管理和监控，提升用户体验。

## 9. 总结

bpfbox 利用 eBPF 技术，实现了一种简洁、高效且灵活的进程限制机制。通过定义易于理解的策略语言，bpfbox 能够对进程的文件系统、网络、信号等操作进行细粒度的控制，同时保持较低的性能开销。尽管目前存在一些限制，但 bpfbox 展示了 eBPF 在系统安全中的巨大潜力，未来有望通过进一步优化和功能扩展，成为 Linux 系统中重要的安全工具之一。

## Q1：bpfbox 如何在用户空间中生成策略并在内核空间中存储和强制执行策略,如何管理进程状态以及如何在用户空间中记录每个事件的审计数据？

A1：

### 1. 用户空间生成策略并在内核空间存储和强制执行策略

bpfbox 采用了一个**用户空间守护进程**（bpfboxd），负责生成和管理策略，确保策略能够正确加载到内核空间，并与 eBPF 程序协同工作。整个过程分为以下几个步骤：

#### 1.1 用户空间策略生成

* **策略定义**：用户通过编写 bpfbox 的策略文件，定义进程可以执行哪些操作（如文件系统访问、网络操作、信号发送等）。这些策略使用 bpfbox 的策略语言来描述。
* **策略解析**：用户空间守护进程 bpfboxd 负责解析这些策略文件，将其转化为适合 eBPF 程序理解的格式。解析后的策略主要包括文件系统、网络、信号等不同类型的规则。

#### 1.2 将策略存储在内核空间

* **加载 eBPF 程序**：解析后的策略由 bpfboxd 转化为 eBPF 程序，通过 bpf(2) 系统调用将这些程序加载到内核空间。每个 eBPF 程序与特定的事件（如系统调用、LSM 钩子等）相关联，确保当事件发生时，相关的 eBPF 程序被触发执行。
* **使用 eBPF Maps 存储策略**：bpfbox 通过 eBPF 的 map 数据结构（内核与用户空间共享的映射表）来存储活动的策略。策略被存储为内核中的映射表，便于 eBPF 程序快速查找并应用这些策略。
  + 策略通常按进程的 inode 编号和文件系统设备编号进行键值映射，确保文件操作或系统调用发生时，能够通过 inode 和设备号来查找和验证策略。

#### 1.3 在内核空间强制执行策略

* **系统调用和 LSM 钩子拦截**：bpfbox 注入的 eBPF 程序会拦截系统调用或 LSM（Linux Security Module）钩子事件，例如文件访问、网络连接、信号发送等。
* **策略匹配与强制执行**：每当触发一个事件时，bpfbox 的 eBPF 程序会查询映射表中的策略并进行匹配。如果该操作符合策略规则，操作将被允许；如果不符合规则，操作将被拒绝。同时，某些操作会被审计（如果配置了 #[audit] 指令）。

### 2. 进程状态管理

为了有效跟踪进程的状态并执行相应的策略，bpfbox 需要管理进程的生命周期（如创建、执行和退出）。bpfbox 通过以下几个步骤实现对进程状态的管理：

#### 2.1 使用内核的 tracepoints 跟踪进程状态

bpfbox 利用 Linux 内核中的 sched tracepoints 来管理进程状态，主要使用以下三个 tracepoints：

* sched:process\_fork：当新进程通过 fork(2)、vfork(2) 或 clone(2) 系统调用创建时，该 tracepoint 被触发。bpfbox 会创建一个新的进程状态条目，并根据父进程的状态初始化子进程的状态。
* sched:process\_exec：当进程调用 execve(2) 加载一个新的程序时，该 tracepoint 被触发。bpfbox 会为该进程分配与新程序对应的策略，并更新进程的状态信息。
* sched:process\_exit：当进程退出时，该 tracepoint 被触发，bpfbox 会删除与该进程相关的状态条目，释放资源。

#### 2.2 跟踪进程状态的 eBPF maps

bpfbox 使用 eBPF maps（哈希表）来存储进程的状态信息。每个进程的状态条目以线程 ID（TID）为键，记录了进程关联的策略键（如进程的 inode 编号）以及进程当前的状态（如是否已 taint）。这允许 bpfbox 动态跟踪每个进程的状态，并根据进程的状态强制执行不同的策略。

#### 2.3 递归函数调用的处理

bpfbox 还可以使用用户空间的 uprobes 或内核空间的 kprobes 钩子来监控函数调用，并根据函数的上下文应用策略。例如，策略可以指定某个文件系统操作只能在特定函数调用的上下文中发生。每个函数调用会更新进程状态中的相应位掩码，标记函数的进入和退出。

### 3. 记录审计数据

bpfbox 支持对策略匹配的操作进行审计记录，具体方法如下：

#### 3.1 使用 ringbuf map 记录事件

**审计日志的存储**：bpfbox 使用 eBPF 的 ringbuf（环形缓冲区）类型 map 来存储事件的审计数据。这种数据结构在性能和可靠性方面具有优势，特别是在多个 CPU 并发访问的情况下。每当某个事件需要记录时，eBPF 程序将审计数据写入 ringbuf map 中。

#### 3.2 用户空间守护进程读取审计数据

**用户空间读取日志**：用户空间守护进程 bpfboxd 会通过 mmap(2) 映射 ringbuf map 中的内存，并通过轮询的方式定期检查是否有新的审计数据。当新的审计事件产生时，bpfboxd 会读取并处理这些事件，将其写入到用户空间的日志文件中。

#### 3.3 审计模式和日志管理

* **审计模式**：bpfbox 支持审计模式，即使某些操作未被拒绝，它们也会被记录下来。通过启用审计，用户可以在后期检查系统中哪些操作触发了策略。
* **日志处理**：审计数据的记录确保用户可以回溯系统中的安全事件，进行安全性分析。通过分析这些日志，用户可以发现异常行为，并根据日志生成新的策略。

### 4. 小结

bpfbox 的用户空间和内核空间协作紧密，用户空间负责解析策略并通过 eBPF 程序将策略加载到内核空间。内核空间通过 eBPF 拦截系统调用和 LSM 钩子事件，并根据用户定义的策略强制执行。同时，bpfbox 通过 eBPF maps 管理进程状态，确保策略能够在进程生命周期的不同阶段得到正确应用。此外，bpfbox 使用高效的 ringbuf 结构记录审计数据，用户空间进程 bpfboxd 定期读取并处理这些审计日志，为系统提供详细的安全事件记录。

## Q2：bpfbox 组件概述。 bpfbox 依赖于 eBPF 程序的三大类： USDT 探针，利用 libbpfbox 来加载和更新策略；跟踪点、kprobes 和 uprobes 用于检测应用程序行为并管理进程状态； LSM 探针根据策略和进程状态映射中存储的信息强制执行策略。这三者分别有什么用？

A2：

### 1. ****USDT 探针（User-Level Statically Defined Tracing Probes）****

USDT 探针主要用于在用户空间与内核空间之间桥接通信，帮助加载和更新策略。它的作用如下：

* **加载和更新策略**：USDT 探针通过 libbpfbox（bpfbox 的用户空间库）来加载和更新策略。策略是由用户空间的 bpfboxd 守护进程解析后生成的，这些策略通过 USDT 探针传递到内核空间，确保策略可以在内核中被应用。
* **用户空间与内核空间的通信**：USDT 探针允许在用户空间直接触发 eBPF 程序，以便更灵活地处理策略更新和事件监控。这使得用户空间守护进程能够直接与内核空间的 eBPF 程序进行交互，从而实现动态策略加载和更新，而无需重启系统或中断进程。

USDT（User-Level Statically Defined Tracing）探针允许在用户空间直接触发 eBPF 程序的原因在于，USDT 提供了一种在用户空间和内核空间之间进行高效交互的机制，使得用户空间的应用程序可以通过调用预定义的探针，动态触发 eBPF 程序。这一功能的实现依赖于以下几个关键机制：

1. USDT 探针的原理

USDT 探针是一种静态定义的用户级别探针，可以在用户空间应用程序的特定位置插入探针点。通过这些探针点，开发者可以在应用程序运行时，监控特定的函数或事件，并将相关信息传递给内核中的 eBPF 程序进行处理。

这些探针在应用程序编译时被嵌入到二进制文件中，类似于内核中的跟踪点（tracepoints）。当应用程序在运行时触发这些探针时，会通过系统调用与内核通信，进而启动 eBPF 程序执行。具体步骤如下：

* 定义和插入 USDT 探针：应用程序开发者可以在源代码中静态插入 USDT 探针点。这些探针点通常会定义在应用程序的关键位置（如函数的入口或特定事件发生时），并会在编译时嵌入到应用程序的二进制文件中。
* 探针激活：当应用程序运行时，如果有需要，USDT 探针会被激活。一旦激活，探针会通过内核提供的 tracing 机制来触发相应的 eBPF 程序。这意味着用户空间的应用程序可以通过探针直接与 eBPF 程序进行交互。
* 触发 eBPF 程序：当 USDT 探针被触发时，内核会通过 perf\_event\_open() 系统调用将该探针与 eBPF 程序关联。这时，预先加载到内核中的 eBPF 程序会立即执行，处理探针传递的数据并作出相应的决策。这个机制允许 eBPF 程序以极低的开销监控和响应用户空间事件。

2. USDT 如何在用户空间触发 eBPF 程序？

USDT 探针允许在用户空间直接触发 eBPF 程序，这种功能的实现依赖于内核和用户空间之间的协同工作。下面是具体的工作流程：

2.1 在用户空间定义和插入探针

在编写用户空间应用程序时，开发者可以通过定义 USDT 探针来插入观测点。例如，开发者可以在应用程序的某个函数调用中插入探针，以便在函数执行时触发 eBPF 程序。插入探针的语法通常类似于：DTRACE\_PROBE(application, probe\_name);这些探针被编译进二进制文件中，并在应用程序的特定位置埋下钩子。

2.2 关联 eBPF 程序

* 在运行时，用户可以使用 eBPF 工具链（如 bpftrace、libbpf 或 bcc 工具集）来将 eBPF 程序绑定到这些探针点。通过使用 perf\_event\_open() 系统调用，用户空间应用程序的 USDT 探针与内核中的 eBPF 程序进行关联。
* 当 USDT 探针被触发时，内核会通过 perf\_event 子系统捕捉到这一事件，并立即调用与该探针相关联的 eBPF 程序。

2.3 触发和执行 eBPF 程序

当应用程序执行到插入了 USDT 探针的代码段时，探针会被触发，内核会通过 perf\_event 子系统启动相应的 eBPF 程序。eBPF 程序此时会运行，处理探针传递的数据（如函数参数、进程信息等），并做出相应的操作。这种触发过程非常高效，几乎不会引入显著的性能开销。

3. USDT 和 eBPF 交互的优势

通过 USDT 探针，bpfbox 实现了从用户空间直接触发 eBPF 程序，这种机制提供了多个优势：

低开销：USDT 探针具有非常低的开销，它们只在被激活时才运行，并且通过高效的内核接口（如 perf\_event\_open()）与 eBPF 程序通信。相比传统的系统调用监控或调试工具，USDT 的开销可以忽略不计。

灵活性：用户空间应用程序可以在不修改内核或重新启动系统的情况下，动态地插入和激活探针点。开发者可以在应用程序的关键位置添加探针，动态观测和调试程序行为。

实时性：eBPF 程序可以在探针触发的瞬间执行，处理来自用户空间的实时数据。这种实时性对于安全监控和系统调试非常有用。

用户空间与内核空间的协作：USDT 探针提供了一个桥梁，允许用户空间应用程序动态地触发和控制内核中的 eBPF 程序，确保用户可以灵活地控制策略和事件处理过程。

### 2. ****跟踪点、kprobes 和 uprobes****

这些探针用于监控应用程序的行为并管理进程状态。它们在应用程序执行过程中动态拦截特定的系统调用和函数调用，作用如下：

#### 2.1 ****跟踪点（Tracepoints）****

* **监控内核事件**：跟踪点是内核中的预定义检测点，能够在系统运行时捕获特定事件（如进程创建、进程退出、文件访问等）。通过跟踪点，bpfbox 可以监控内核中的重要行为，并做出相应的策略决策。
* **管理进程状态**：bpfbox 使用跟踪点（如 sched:process\_fork、sched:process\_exec 和 sched:process\_exit）来跟踪进程的创建、执行和退出，从而管理每个进程的生命周期和状态。这确保了不同阶段的进程行为可以被正确监控和限制。

#### 2.2 ****kprobes 和 uprobes****

* **监控函数调用**：kprobes 用于监控内核空间的函数调用，uprobes 则用于监控用户空间的函数调用。bpfbox 可以通过这些探针来捕获特定函数的进入和退出，以便确定当前进程的执行上下文。
* **细粒度策略执行**：通过 kprobes 和 uprobes，bpfbox 可以实现更细粒度的策略控制。例如，策略可以指定某个文件操作只能在特定的函数上下文中发生，这样可以防止进程在错误的上下文中执行敏感操作。
* **管理函数调用栈信息**：这些探针还帮助 bpfbox 记录进程的函数调用栈信息，以便根据当前的函数上下文应用适当的策略。

总结来说，跟踪点和 kprobes/uprobes 主要用于实时检测应用程序行为，管理进程状态，并在特定条件下强制执行细粒度的访问控制策略。

### 3. ****LSM 探针（Linux Security Module Probes）****

LSM 探针用于根据策略和进程状态执行实际的安全控制。它们与内核中的 LSM 钩子配合，作用如下：

* **强制执行安全策略**：bpfbox 通过 LSM 探针将 eBPF 程序挂载到 LSM 钩子上，用于强制执行安全策略。LSM 钩子能够在关键的安全检查点上（如文件打开、进程间通信、网络访问等）触发 bpfbox 的 eBPF 程序，从而决定是否允许该操作。
* **动态控制权限**：LSM 探针不仅可以根据预定义策略强制执行操作，还可以根据进程的实时状态（如是否已被 taint）做出动态的权限控制决策。这使得 bpfbox 能够根据进程的行为实时调整安全策略。
* **覆盖系统层面的安全事件**：LSM 提供了覆盖文件系统、网络、进程间通信等多种安全领域的钩子。通过 LSM 探针，bpfbox 能够全面监控和控制这些系统层面的安全事件，确保进程行为符合最小权限原则。

总结，LSM 探针的主要作用是在内核关键安全点上进行策略检查和强制执行，确保系统的安全策略能够覆盖广泛的进程行为。

|  |  |  |
| --- | --- | --- |
| **特性** | **LSM 钩子** | **LSM 探针** |
| **定义** | 内核中的固定安全检查接口 | 利用 eBPF 动态附加到 LSM 钩子的程序 |
| **实现方式** | 编译时固定在内核中，由安全模块使用 | 动态加载，通过 eBPF 实现 |
| **灵活性** | 需要在编译时集成，不能动态更改 | 运行时动态加载和更新 |
| **性能** | 性能开销较大，影响较广 | eBPF 经过优化，开销较小 |
| **用途** | 为安全模块（如 SELinux）提供接口。这些钩子位于内核的多个关键位置，例如文件操作、网络操作、进程管理等，允许安全模块在必要时插入额外的访问控制检查。 | 通过 eBPF，LSM 探针可以在系统运行时加载，并附着到 LSM 钩子上，提供动态安全检查和访问控制。 |
| **典型应用** | SELinux、AppArmor 等传统 LSM 模块 | bpfbox、Landlock 等动态安全工具 |

### 总结

* **USDT 探针**：负责用户空间策略的加载和更新，确保策略从用户空间到内核空间的顺利传递。
* **跟踪点、kprobes 和 uprobes**：用于监控应用程序行为，管理进程状态，并实现基于函数上下文的精细访问控制。
* **LSM 探针**：负责根据策略和进程状态在系统层面强制执行安全策略，确保进程行为符合安全要求。