

# seccomp-jail 项目文档

---

- [seccomp-jail 项目文档](#)
  - [项目描述](#)
  - [赛题](#)
  - [赛题完成情况](#)
  - [开发日程](#)
  - [演示视频](#)
  - [原理介绍](#)
    - [Ptrace](#)
    - [Seccomp](#)
    - [Linux下的进（线）程](#)
    - [软件工作流程](#)
  - [代码目录索引](#)
  - [使用说明](#)
    - [环境/依赖](#)
    - [编译/安装](#)
    - [页面介绍](#)
    - [使用流程](#)
    - [注意事项](#)
  - [样例测试](#)
    - [Test1: mkdir test](#)
    - [Test2: mkdir test1](#)
    - [Test3: 测试动态改变规则](#)
    - [Test4: 更改系统调用号](#)
    - [Test5: 多进程程序1](#)
    - [Test6: 多进程程序2](#)
    - [Test7: 主动执行系统调用测试](#)
  - [开发过程中遇到的问题](#)
    - [seccomp的功能限制](#)
    - [不使用ptrace操作进程](#)
  - [目前已知问题](#)
  - [未来扩展功能](#)
  - [参考资料](#)
    - [Ptrace](#)
    - [Seccomp](#)
    - [Waitpid](#)
    - [Libseccomp](#)
    - [QML FluentUI](#)
    - [Stack Overflow](#)

## 项目描述

在deepin操作系统中，基于seccomp开发一个程序。其可以根据用户的选择动态地对该程序直接或间接启动的子进程产生的系统调用进行过滤。

## 赛题

能够根据固定的规则对直接/间接启动的进程的系统调用进行过滤;

能够动态地对直接/间接启动的进程的系统调用进行过滤, 指当系统调用发生以后弹出窗口展示程序的系统参数, 根据用户的选择对系统调用进行放行;

能够动态地根据用户输入生成对应系统调用的结果。

## 赛题完成情况

☑能够根据固定的规则对直接/间接启动的进程的系统调用进行过滤;

☑能够动态地对直接/间接启动的进程的系统调用进行过滤, 指当系统调用发生以后弹出窗口展示程序的系统参数, 根据用户的选择对系统调用进行放行;

☑能够动态地根据用户输入生成对应系统调用的结果。

## 开发日程

2024.3.9 确定选题, 开始阅读seccomp文档。

2024.3.11 开始编写demo, 使用seccomp-unotify模式

2024.3.31 demo编写完成, 能将系统调用通知给用户, 并由用户选择是否通过, 但无法获取返回值。意识到单seccomp的功能有限, 决定换用ptrace与seccomp结合的方式重构。开始阅读ptrace文档, 编写新的demo。

2024.4.28 demo编写完成, 基本实现了全部赛题内容, 临时编写的GUI界面的美观性和易用性存在较大问题, 决定寻找一个前端框架。

2024.5.6 决定使用基于qml的开源框架 FluentUI For QML。开始阅读并修改部分源码以适应需求; 尝试构建软件界面; 修改后端代码, 从原有的阻塞模式 (在一个系统调用显示到前端后, 在用户处理此事件之前, 其他进程的所有待处理系统调用都不能被处理) 改为非阻塞模式 (所有待处理的系统调用会同时列出, 用户可以任意选择处理); 开始将前后端进行适配。

2024.5.10 移植完毕, 继续优化交互性, 和修复一些小的美观问题。调整主动注入功能。

2024.5.16 调整完毕。主动注入功能从只能在程序处在用户态时注入改为在任何状态均可; 由原有的使用libc改为使用由LD\_PRELOAD提前加载的空白动态库libhookhere.so。好处: 减少了对正常代码的修改, 降低风险性; 可以支持同时对多个进程进行注入; 执行完毕后不需要恢复该动态库的代码段, 比较方便。项目基本完成。

2024.5.25 添加对Signal-delivery-stop和Group-stop两种暂停的处理。开始删除调试用代码和一些无效注释, 继续修复各种小bug。

2024.5.27 发布Release v1.0版本, 后发现较严重bug, 该bug导致不能为系统调用指定abort forever规则, 查明原因为适配原界面的遗留代码导致。检查并修复其他bug, 准备发布v1.1版本。

2024.5.28 发布Release v1.1版本。

2024.5.31 修复罕见情况下, 主动执行系统调用功能在检测暂停状态时可能会吞掉seccomp事件发出的暂停信号导致目标程序僵死的问题。

## 演示视频

与样例测试配套

\*因视频与文档不是同时制作，故同个测试在文档和视频中的PID和参数等可能会不同。

video: [度盘](#)

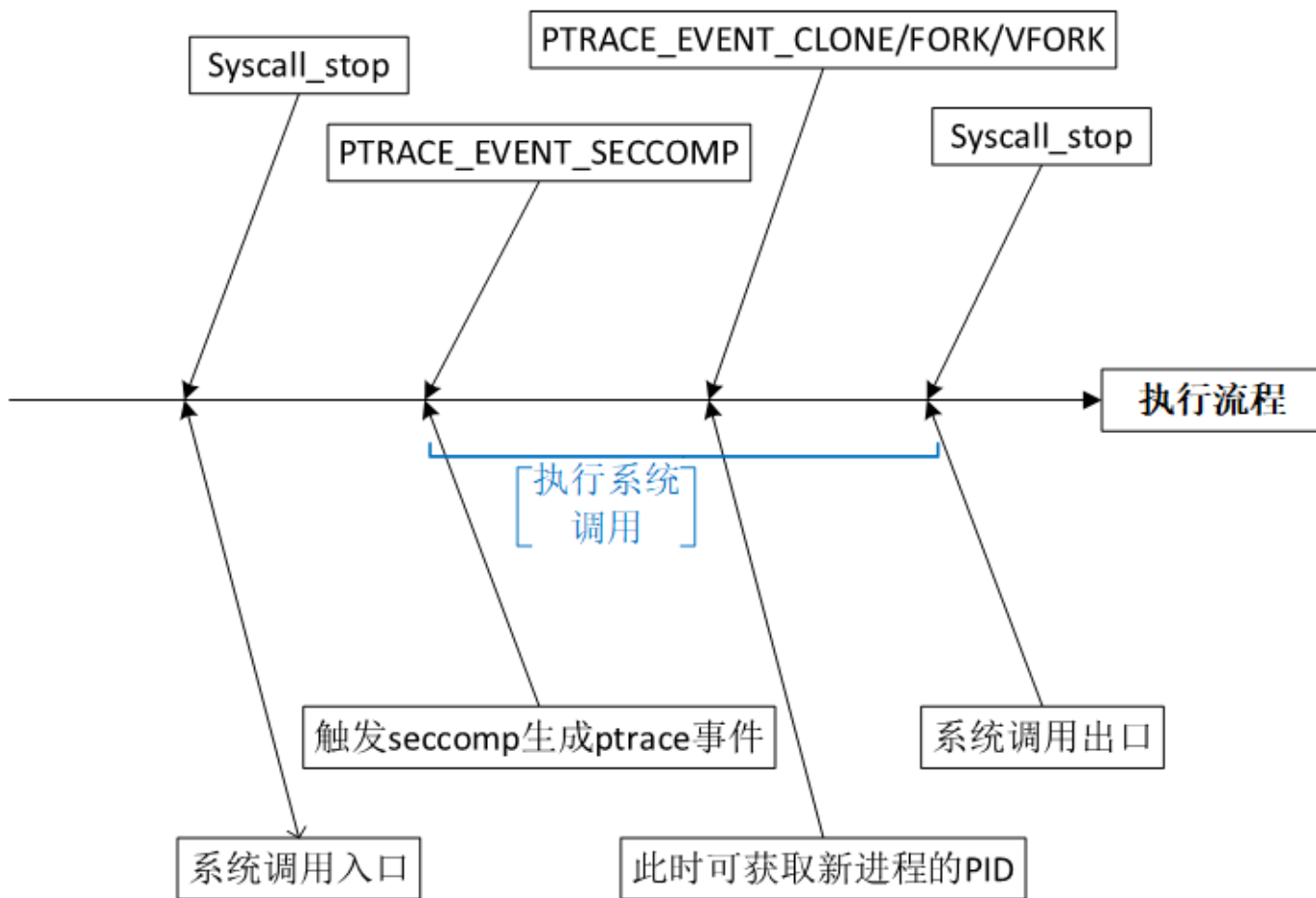
## 原理介绍

### PTRACE

ptrace是linux下一个强大的系统调用，可以通过它来调试其他进程。它提供了一种方法让一个进程（Tracer）可以观察和控制另一个进程的执行（Tracee），并检查和更改Tracee的存储和寄存器。它主要用于实现断点调试和系统调用跟踪。

#### ptrace的不同模式和暂停的不同种类

ptrace可以大致分为两种：traceme(attach其实是令目标进程执行traceme后进行追踪，属于同一种)和seize两种，使用seize模式附加到目标进程时不会停止目标进程，只有在触发了预先设置的事件时才会停止。且seize模式下的api比较友好。一开始使用attach模式，但后来因为上述原因换用seize模式。ptrace触发的暂停事件有很多种，本项目中主要用到了PTRACE\_EVENT\_SECCOMP和Syscall\_stops两种，对于clone, fork, vfork三个系统调用，还需考虑PTRACE\_EVENT\_CLONE/FORK/VFORK暂停的情况，经测试，几种暂停的顺序如下。



其中，Syscall-stops由PTRACE\_SYSCALL触发，tracee在用户态时使用该指令会使其停在下一个系统调用入口，在进入系统调用后使用则会使其停在该系统调用的出口。

对于clone, fork, vfork三个较特殊的系统调用，在设置了PTRACE\_O\_TRACECLONE/TRACEFORK/TRACEVFORK后，会在如图所示的时机触发PTRACE\_EVENT\_CLONE/FORK/VFORK暂停，此时系统调用已经执行，父进程被

暂停，以便tracer可以获取子进程的PID，刚创建好的子进程被立刻暂停并自动被tracer追踪。这样的机制可以防止tracee的子进程脱离tracer控制。

每次进程暂停后会发出信息，需要由监控者用waitpid函数接收，如在忽略该信号的情况下恢复该程序的执行，可能会导致监视者无法再收到此进程的消息(在此种情况下，若该进程再次暂停，会因无法通知监视者导致自己僵死)。

## 如何分辨不同种类的暂停

waitpid函数的定义如下：

```
pid_t waitpid(pid_t pid, int *_Nullable wstatus, int options);
```

使用时如：

tracer调用waitpid后，第二个参数处的变量会存有此次暂停的相关信息。通过一些系统自带的宏或者ptrace文档中提供的计算方式即可分辨暂停的类别。

## SECCOMP

seccomp是Linux内核提供了一种安全机制，用于在用户态应用程序执行系统调用时进行过滤和限制。其原理是基于对系统调用号的过滤和限制，以及对系统调用参数的校验。当一个应用程序启动时，它可以使用seccomp机制来加载一个过滤器，该过滤器指定了允许或禁止执行的系统调用。在应用程序执行系统调用时，seccomp会根据过滤器的规则进行匹配和处理。

seccomp有三种模式：

### strict

只允许read write exit rt\_sigreturn四种系统调用，尝试执行其他系统调用进程会被杀死。

### bpf

允许在程序运行前提前编写规则并加载到内核中，可以指定系统调用号和对应的操作。

### unotify

在bpf模式的基础上，若将规则设定为notify，即可将系统调用是否执行的决定权交由另一个进程处理，提供更强灵活性。

## linux下的进（线）程

linux下的线程其实是“轻量级”进程，在内核中都用task\_struct的结构表示。故ptrace和seccomp对进程和线程的行为大致是相同的。且二者都会在执行exec(进程替换)，clone一族(产生子进程)两种操作时保持自己对新进程的控制。

## 软件工作流程

软件由前端，controller，watcher三个部分组成。

## 前端

负责与用户交互，显示信息，接受用户操作等。

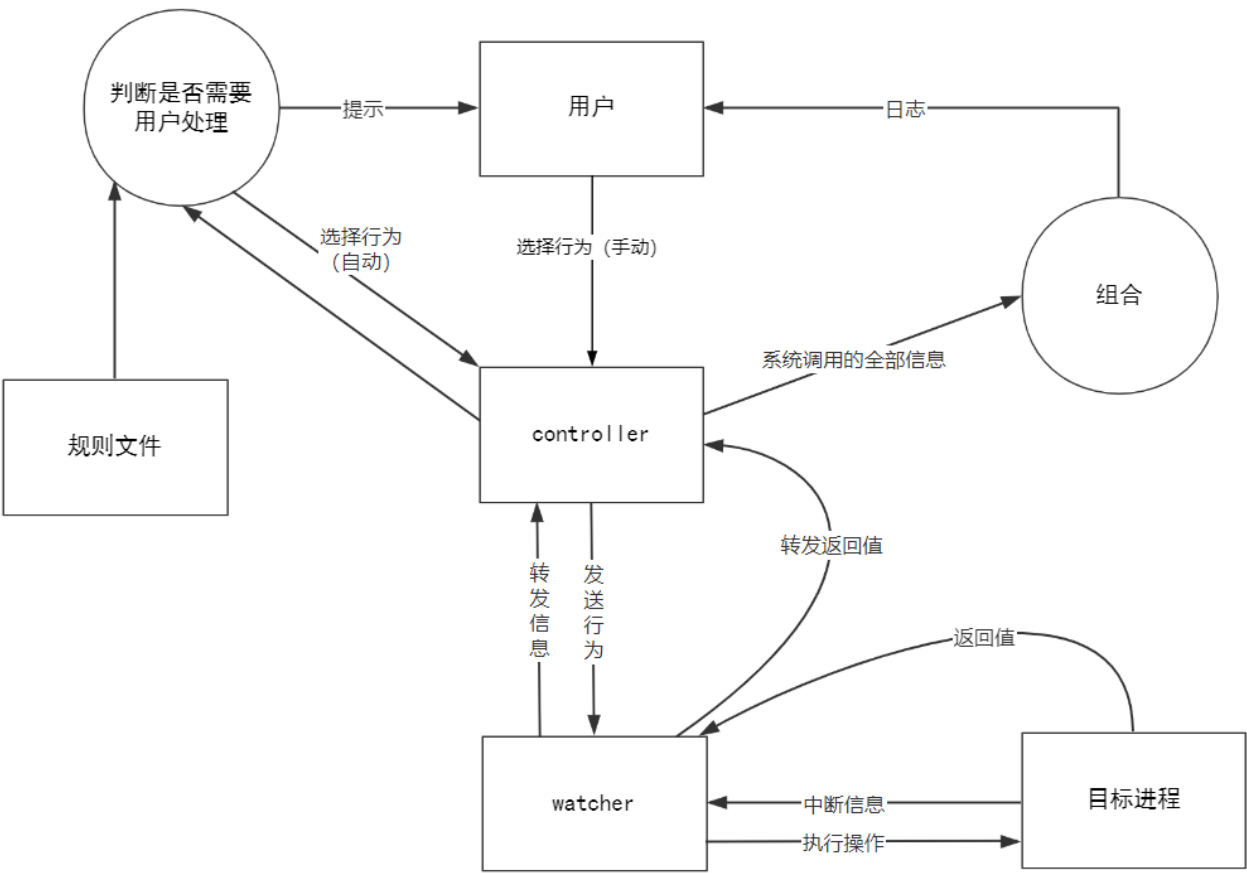
controller

前端和watcher间的桥梁，负责处理watcher发来的信息和要交给前端的信息、检查规则文件、指示watcher的行动等。完全使用信号槽机制构建。

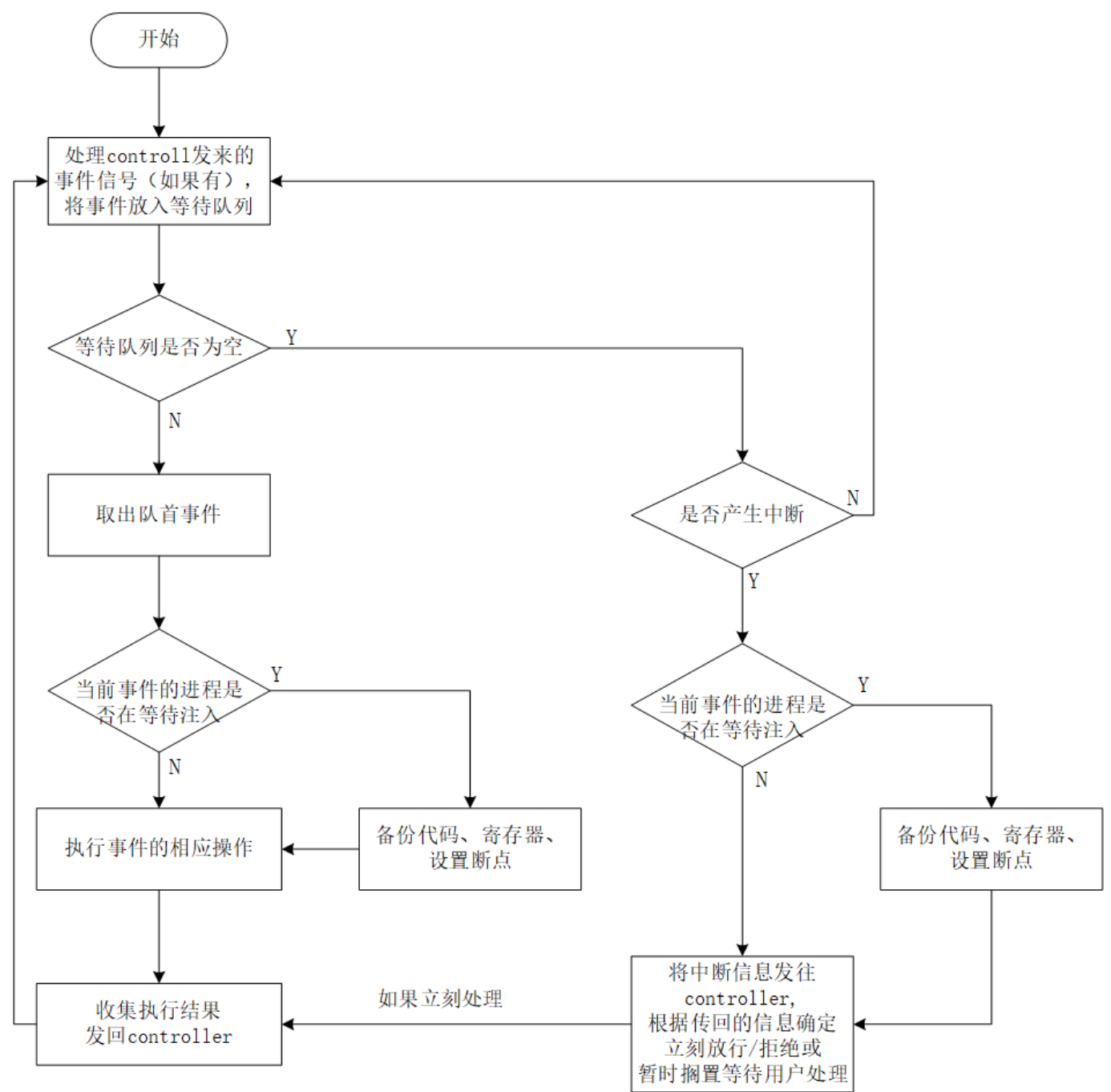
watcher

直接与目标程序交互，目标程序的实际控制者，接受controller的指令决定下一步操作。由于需要轮询进入暂停状态的进程，使用死循环。

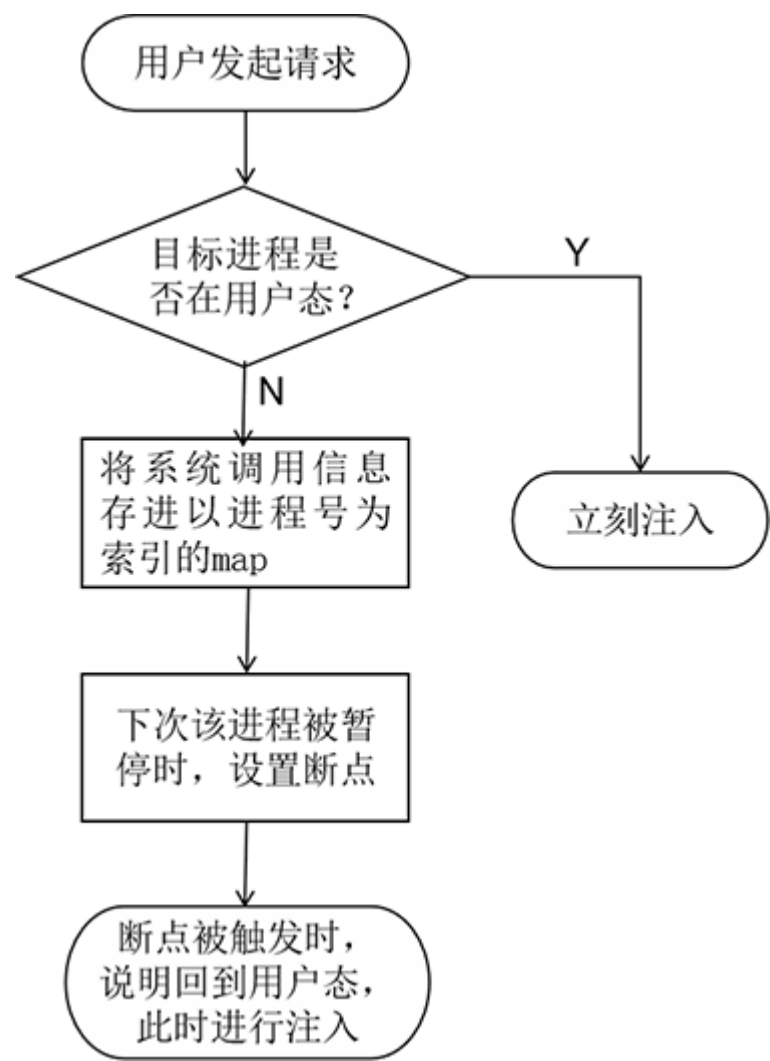
软件工作流程总图



watcher工作流程图

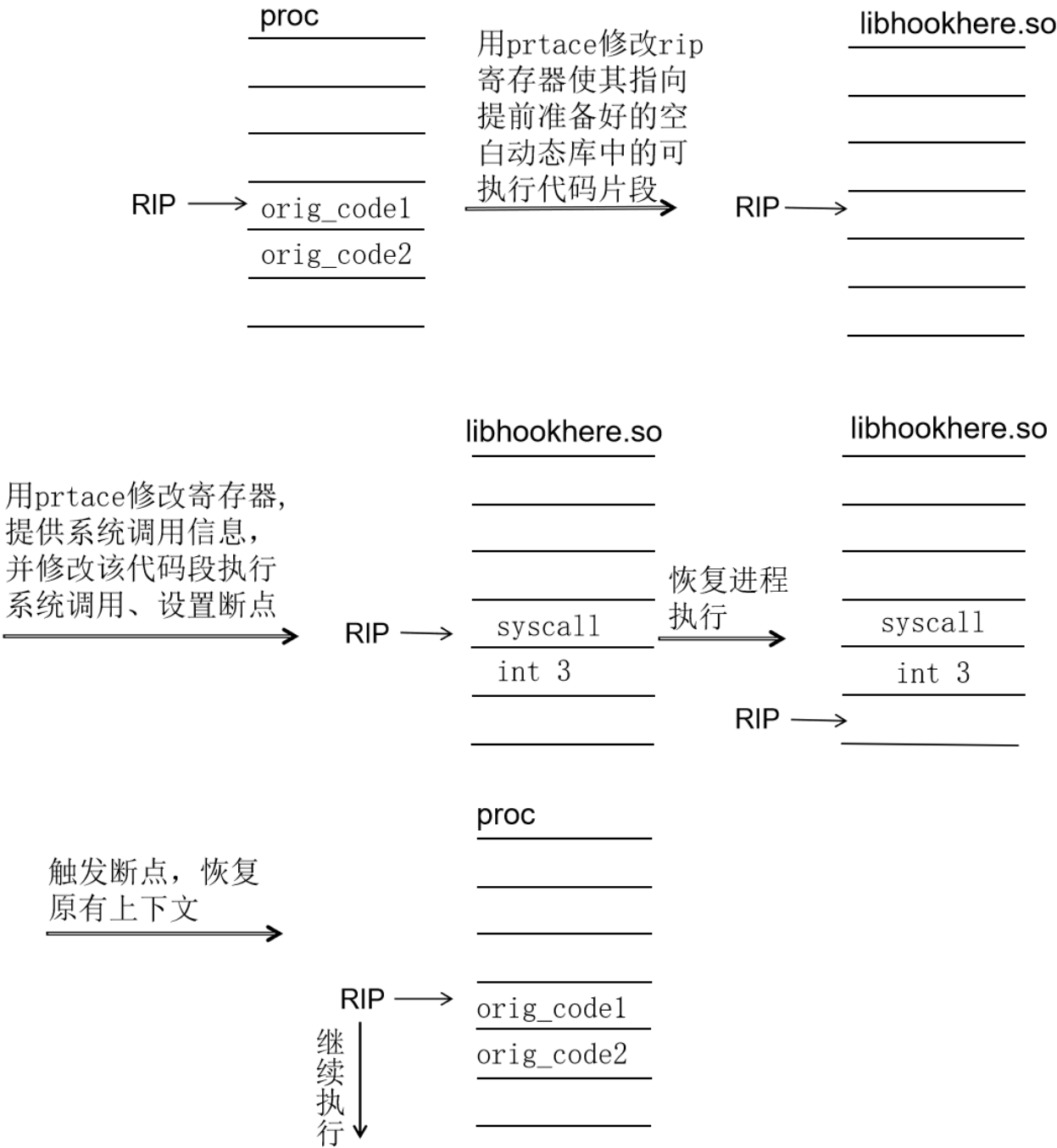


主动执行系统调用流程图



注入原理图

\*在tracer采用ptrace的seize模式附加到tracee之后，tracer可以使用PRACE\_INTERRUPT单方面暂停tracee，同时可以根据返回的信息判断tracee在暂停时的工作状态。此图为tracee暂停时工作在用户态的情况。若为其他情况，则需要额外在tracee代码段中设置一个int3断点，待进程返回用户态执行到该断点后，再由tracer捕获并开始下列流程。



## 代码目录索引

.	
├─ CMakeLists.txt	-----项目文件
├─ FluentUI	-----前端控件库
├─ LICENSE	
├─ README.md	
└─ src	-----源码

使用说明

环境/依赖

支持架构：x86\_64

开发环境：deepinV23, Qt 6.6.2

依赖：

[前端框架 QML FluentUI by zhuzichu](#)

[libseccomp](#)

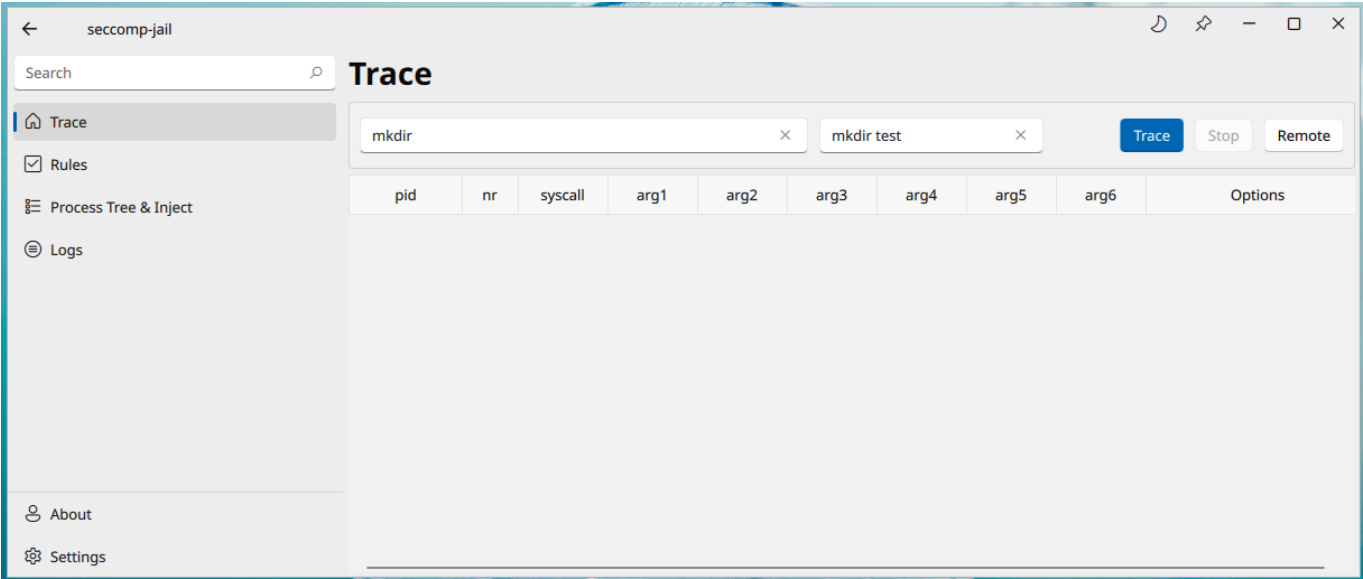
pstree(大部分linux发行版应该都有)

编译/安装

下载打包好的Release版本并解压，将ApplImage可执行文件和libhookhere.so放在同一目录,运行ApplImage即可使用；如需自行编译源码运行，且环境中原已有Fluentui控件库，请注意编译此项目时会导致其中一些控件的行为被改变。

页面介绍

总览



主界面

# Trace

mkdir

×

mkdir test

×

Trace

Stop

Remote

	pid	nr	syscall	arg1	arg2	arg3	arg4	arg5	arg6	Options
1	53410	83	__NR_mk dir	1407357 314016...	511	0	1404108 920117...	511	9461060 5329920	Select an action ▾

在此界面启动程序、手动处理系统调用

规则编辑

Select rule

Load rule

Create rule

Save

Save As

	nr	Name	status	Change to
1	0	__NR_read	Allow forever	<div>allow ▾</div> <div>Update</div>
2	1	__NR_write	Allow forever	<div>allow ▾</div> <div>Update</div>
3	2	__NR_open	Allow	<div>allow ▾</div> <div>Update</div>
4	3	__NR_close	Allow forever	<div>allow ▾</div> <div>Update</div>
5	4	__NR_stat	Allow	<div>allow ▾</div> <div>Update</div>
6	5	__NR_fstat	Allow	<div>allow ▾</div> <div>Update</div>

<Previous

1

2

3

4

5

6

...

10

Next>

在此界面创建、加载、修改、保存规则文件

主动执行

mkdir(53410)

Pid: 53410

×

+

syscall number and argc

nr

0

▼

Execute

Clear

arg1:

arg1

arg2:

arg2

arg3:

arg3

arg4:

arg4

arg5:

arg5

arg6:

arg6

在此界面上方显示目标程序进程/线程树，下方提供主动执行系统调用的接口

日志

Logs		
	Log	Options
1	Sun May 26 17:22:48 2024 pid: 141357 nr: 137(__NR_statfs) arg1: 140623164965488 arg2: 14073269598 0480 arg3: 140623165002196 arg4: 140623165106120 arg5: 0 arg6: 12 action: pass according to rules...	<div>Delete</div> <div>Copy</div>
2	Sun May 26 17:22:48 2024 pid: 141357 nr: 137(__NR_statfs) arg1: 140623164965495 arg2: 14073269598 0480 arg3: 18446744073709551264 arg4: 140623165106120 arg5: 0 arg6: 12 action: pass according t...	<div>Delete</div> <div>Copy</div>
3	Sun May 26 17:22:48 2024 pid: 141357 nr: 318(__NR_getrandom) arg1: 140623164789912 arg2: 8 arg3: 1 arg4: 140623165106120 arg5: 0 arg6: 12 action: pass according to rules returnval: 8	<div>Delete</div> <div>Copy</div>
4	Sun May 26 17:22:48 2024 pid: 141357 nr: 12(__NR_brk) arg1: 0 arg2: 140623164763264 arg3: 0 arg4: 4 arg5: 135168 arg6: 1 action: pass according to rules returnval: 94221040357376	<div>Delete</div> <div>Copy</div>
5	Sun May 26 17:22:48 2024 pid: 141357 nr: 12(__NR_brk) arg1: 94221040492544 arg2: 140623164763264 arg3: 140623164793656 arg4: 4 arg5: 135168 arg6: 1 action: pass according to rules returnval: 94221...	<div>Delete</div> <div>Copy</div>
6	Sun May 26 17:22:48 2024 pid: 141357 nr: 257(__NR_openat) arg1: 4294967196 arg2: 140623164965460 arg3: 524288 arg4: 0 arg5: 8 arg6: 1 action: pass according to rules returnval: 3	<div>Delete</div> <div>Copy</div>
7	Sun May 26 17:22:48 2024 pid: 141357 nr: 262(__NR_newfstatat) arg1: 3 arg2: 140623164509075 arg3: 1 40732695980160 arg4: 4096 arg5: 128 arg6: 1 action: pass according to rules returnval: 0	<div>Delete</div> <div>Copy</div>
8	Sun May 26 17:22:48 2024 pid: 141357 nr: 21(__NR_access) arg1: 140623164965528 arg2: 0 arg3: 23003	<div>Delete</div> <div>Copy</div>

此界面显示日志，包括自动审批的和用户手动审批的。

使用流程

编辑规则

如果目前没有规则文件，点击Create rule 可以选择一种预设规则创建一个默认规则文件。

规则有五种类型：

永远拒绝(Abort forever)

拒绝该系统调用，且在目标进程的此次运行中不可更改此条规则

### 永远允许(Allow forever)

允许该系统调用，且在目标进程的此次运行中不可更改此条规则

设置上面两条永久规则主要是为了提高运行效率，永久规则不会经过用户态,适合一些规则已经确定的系统调用或程序运行必不可少，不需要监控的系统调用。

### 提示(Notify)

将该系统调用交由用户处理，用户可以自由选择是否允许此系统调用并可以将该行为加入规则，以后再次遇到相同系统调用会自动审批。

### 允许(Allow)

允许该系统调用，且在目标进程的此次运行中可以更改此条规则。

### 拒绝(Abort)

拒绝该系统调用，且在目标进程的此次运行中可以更改此条规则。

创建规则文件后，点击load加载规则，此时可以查看并修改规则，可以按系统调用名查找对应规则。目标程序运行途中，也可以在此界面更改规则，即时生效。

### 编辑目标程序路径和参数

在主界面输入程序路径（相对路径/绝对路径/命令如mkdir、ls等）和参数后，点击Trace即可开始监控目标程序，可以点击Stop强制停止目标程序运行。

### 监测系统调用

设置了notify规则的系统调用会显示详细信息在主界面等待用户处理，双击对应的栏位，可修改对应的参数（系统调用号和六个参数）。直到用户选择一种行动前，请求该系统调用的进程处于暂停状态。可选择的行动有五种：

#### 允许

允许执行该系统调用

#### 拒绝

拒绝执行该系统调用

#### 允许并添加规则

允许执行该系统调用，后续该系统调用自动审批通过。

#### 拒绝并添加规则

拒绝执行该系统调用，后续该系统调用自动审批拒绝。

查看详细信息

弹出一个窗口，在此窗口中可以将系统调用的六个参数作为地址，查看指定个字长的数据。（\*此选项仅作查看用，仍需选择上方四个选项的一种来审批该系统调用）

主动执行系统调用

在主动执行界面上方会显示目标程序的进程树。点击下方区域的加号，可以打开一个进程树中已有的进程的注入选项卡，在选项卡中输入系统调用号和六个参数后，按Execute即可执行该系统调用。支持同时注入多个进程（理论上同时注入多个线程也支持，但因为线程共享代码段，可能出现预料之外的问题，十分不推荐）

查看日志

此界面会显示系统调用日志，除两条永久规则的系统调用外，用户手动、系统自动审批的系统调用和主动执行的系统调用都会在此显示，可以点击右侧clear清除条目或copy复制该条目的文本。日志内容包括日期时间，进程号，系统调用号和调用名，六个参数，采取的操作，返回值。

注意事项

1. 对clone、fork、vfork三个系统调用而言，将他们的规则设为allow forever将使seccomp-jail不追踪他们创建的子进程。
2. 在测试环境中(deepinV23 x86\_64 内核版本 6.1.32)，在调用clone()、fork()、vfork()时，seccomp会将其都归为clone系统调用，但ptrace可以检出三种不同系统调用。结合第一条，例如：若要追踪fork及子进程，建议(或者必须)将clone和fork两条规则全部打开。clone规则影响实际的行为，fork规则只要不置于allow forever或abort forever即可。视系统或内核的不同，情况可能会有变化，但将三者保持为同一规则总是没错的。

样例测试

Test1: mkdir test

说明：

试图创建一个名为test的文件夹。

规则：

系统调用名	规则	用户操作	备注
mkdir	notify	允许	
其他	allow forever		

过程：

Trace

mkdir

×

mkdir test

×

Trace

Stop

Remote

	pid	nr	syscall	arg1	arg2	arg3	arg4	arg5	arg6	Options
1	11072	83	__NR_mkdir	1407257584854...	511	0	1396406596426...	511	94155916209664	Select an action ▾

成功捕获。

Detail

arg1

▾

4

×

Peek

5498687339275445000(test)

8375935897724802000(ORTERM=t)

8245928668403430000(ruecolor)

6431505650740184000()

4993734732483149000(=:0)

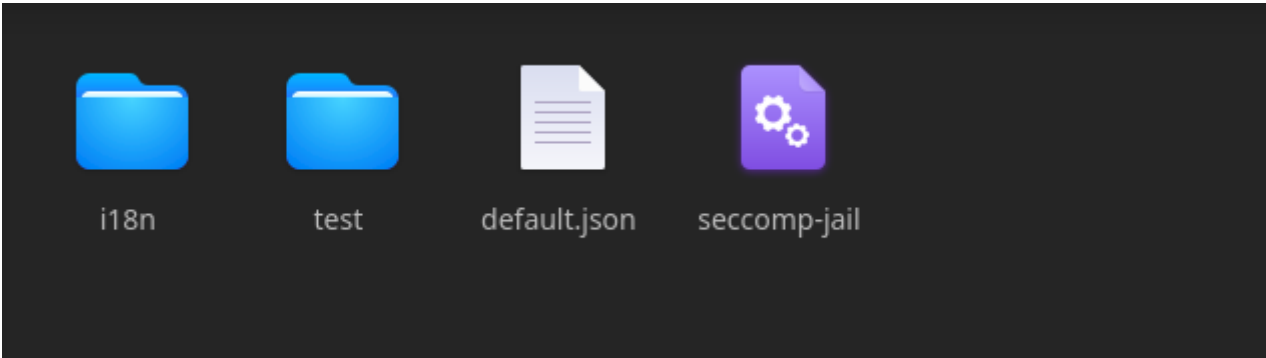
右侧选择More..进入查看详细信息，将第一个参数作为地址查看四个字长的数据，可以发现显示出了文件夹名。

\*因为会将读到的数据全部转为字符串，所以后面会有一些无意义/作其他用途的字符。

Logs

	Log	Options
1	Mon May 27 19:59:05 2024 pid: 11072 nr: 83(__NR_mkdir) arg1: 140725758485415 arg2: 511 arg3: 0 arg4: 139640659642624 arg5: 511 arg6: 94155916209664 action: pass by user returnval: 0	<div>DeleteCopy</div>

选择允许此系统调用，转到日志界面可以发现此条记录。



可以发现成功生成了test文件夹。

Test2: mkdir test1

规则：

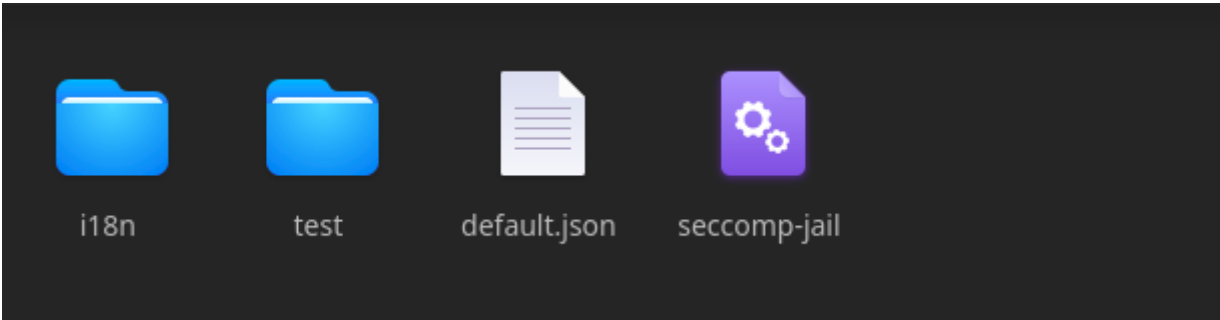
系统调用名	规则	用户操作	备注
mkdir	notify	拒绝	
其他	allow forever		

过程：

前同Test1



在选择禁止执行该系统调用后，转到日志界面可以发现此条记录。



没有生成test1文件夹。

Test3: 测试动态改变规则

说明:

在一个死循环中反复调用getpid(),在运行过程中改变规则，观察软件行为。

目标程序主要源码:

```
int main()
{
    while(1)
    {
        getpid();
        sleep(1);
    }
}
```

规则:

系统调用名	规则	用户操作	备注
getpid	allow -> abort -> notify	允许并添加规则	
其他	allow forever		

过程:

将getpid的规则设为allow后启动程序。

Logs		
	Log	Options
1	Tue May 28 15:34:57 2024 pid: 4296 nr: 39(__NR_getpid) arg1: 1 arg2: 140732659932872 arg3: 140732659932888 arg4: 139816003264952 arg5: 0 arg6: 139816005278432 action: pass according to rules ret...	Delete Copy
2	Tue May 28 15:34:58 2024 pid: 4296 nr: 39(__NR_getpid) arg1: 0 arg2: 0 arg3: 0 arg4: 140732659932528 arg5: 0 arg6: 139816005278432 action: pass according to rules returnval: 4296	Delete Copy
3	Tue May 28 15:34:59 2024 pid: 4296 nr: 39(__NR_getpid) arg1: 0 arg2: 0 arg3: 0 arg4: 140732659932528 arg5: 0 arg6: 139816005278432 action: pass according to rules returnval: 4296	Delete Copy
4	Tue May 28 15:35:00 2024 pid: 4296 nr: 39(__NR_getpid) arg1: 0 arg2: 0 arg3: 0 arg4: 140732659932528 arg5: 0 arg6: 139816005278432 action: pass according to rules returnval: 4296	Delete Copy

查看日志，发现系统正在自动通过getpid系统调用。

Logs		
	Log	Options
32	arg5: 0 arg6: 139816005278432 action: pass according to rules returnval: 4296	Delete Copy
33	Tue May 28 15:35:29 2024 pid: 4296 nr: 39(__NR_getpid) arg1: 0 arg2: 0 arg3: 0 arg4: 140732659932528 arg5: 0 arg6: 139816005278432 action: pass according to rules returnval: 4296	Delete Copy
34	Tue May 28 15:35:30 2024 pid: 4296 nr: 39(__NR_getpid) arg1: 0 arg2: 0 arg3: 0 arg4: 140732659932528 arg5: 0 arg6: 139816005278432 action: pass according to rules returnval: 4296	Delete Copy
35	Tue May 28 15:35:31 2024 pid: 4296 nr: 39(__NR_getpid) arg1: 0 arg2: 0 arg3: 0 arg4: 140732659932528 arg5: 0 arg6: 139816005278432 action: pass according to rules returnval: 4296	Delete Copy
36	Tue May 28 15:35:32 2024 pid: 4296 nr: 39(__NR_getpid) arg1: 0 arg2: 0 arg3: 0 arg4: 140732659932528 arg5: 0 arg6: 139816005278432 action: abort according to rules	Delete Copy
37	Tue May 28 15:35:33 2024 pid: 4296 nr: 39(__NR_getpid) arg1: 0 arg2: 0 arg3: 0 arg4: 140732659932528 arg5: 0 arg6: 139816005278432 action: abort according to rules	Delete Copy

修改规则为abort后，发现系统改为自动拒绝该系统调用。

Trace

/home/user/playground/Test3

Args

Trace

Stop

Remote

	pid	nr	syscall	arg1	arg2	arg3	arg4	arg5	arg6	Options
1	4296	39	__NR_getpid	0	0	0	1407326599325...	0	1398160052784...	Select an action ▾

修改规则为notify后，系统调用信息在主界面等待用户处理。

Logs

	Log	Options
	139816005278432 action: abort according to rules	
67	Tue May 28 15:36:03 2024 pid: 4296 nr: 39(__NR_getpid) arg1: 0 arg2: 0 arg3: 0 arg4: 140732659932528 arg5: 0 arg6: 139816005278432 action: abort according to rules	Delete Copy
68	Tue May 28 15:36:04 2024 pid: 4296 nr: 39(__NR_getpid) arg1: 0 arg2: 0 arg3: 0 arg4: 140732659932528 arg5: 0 arg6: 139816005278432 action: abort according to rules	Delete Copy
69	Tue May 28 15:36:05 2024 pid: 4296 nr: 39(__NR_getpid) arg1: 0 arg2: 0 arg3: 0 arg4: 140732659932528 arg5: 0 arg6: 139816005278432 action: abort according to rules	Delete Copy
70	Tue May 28 15:37:09 2024 pid: 4296 nr: 39(__NR_getpid) arg1: 0 arg2: 0 arg3: 0 arg4: 140732659932528 arg5: 0 arg6: 139816005278432 action: pass by user and add rule returnval: 4296	Delete Copy
71	Tue May 28 15:37:10 2024 pid: 4296 nr: 39(__NR_getpid) arg1: 0 arg2: 0 arg3: 0 arg4: 140732659932528 arg5: 0 arg6: 139816005278432 action: pass according to rules returnval: 4296	Delete Copy
72	Tue May 28 15:37:11 2024 pid: 4296 nr: 39(__NR_getpid) arg1: 0 arg2: 0 arg3: 0 arg4: 140732659932528 arg5: 0 arg6: 139816005278432 action: pass according to rules returnval: 4296	Delete Copy
73	Tue May 28 15:37:12 2024 pid: 4296 nr: 39(__NR_getpid) arg1: 0 arg2: 0 arg3: 0 arg4: 140732659932528 arg5: 0 arg6: 139816005278432 action: pass according to rules returnval: 4296	Delete Copy

选择允许并添加规则后，系统再次开始自动通过该调用。

Test4: 更改系统调用号

说明：

更改Test3中程序的某次getpid为getuid。

目标程序主要源码：

同Test3

规则：

系统调用名	规则	用户操作	备注
getpid	notify	允许	更改系统调用号为getuid对应的系统调用号(102)
其他	allow forever		

过程：

Logs

	Log	Options
1	Tue May 28 16:08:31 2024 pid: 25634 nr: 39(__NR_getpid) arg1: 1 arg2: 140731515089112 arg3: 140731515089128 a rg4: 139780486017464 arg5: 0 arg6: 139780488030944 action: pass by user returnval: 25634	<div>DeleteCopy</div>
2	Tue May 28 16:08:50 2024 pid: 25634 nr: 39(__NR_getpid) arg1: 0 arg2: 0 arg3: 0 arg4: 140731515088768 arg5: 0 arg 6: 139780488030944 action: pass by user returnval: 25634	<div>DeleteCopy</div>

先正常通过几次，程序正常工作。

Trace

/home/user/playground/Test3

×

Args

Trace

Stop

Remote

	pid	nr	syscall	arg1	arg2	arg3	arg4	arg5	arg6	Options
1	25634	102 ×	__NR_get pid	0	0	0	1407315 150887...	0	1397804 880309...	<div>Select an action ▾</div>

在主界面双击系统调用号，修改为102，接着允许该调用。

Logs

	Log	Options
1	Tue May 28 16:08:31 2024 pid: 25634 nr: 39(__NR_getpid) arg1: 1 arg2: 140731515089112 arg3: 140731515089128 a rg4: 139780486017464 arg5: 0 arg6: 139780488030944 action: pass by user returnval: 25634	<div>DeleteCopy</div>
2	Tue May 28 16:08:50 2024 pid: 25634 nr: 39(__NR_getpid) arg1: 0 arg2: 0 arg3: 0 arg4: 140731515088768 arg5: 0 arg 6: 139780488030944 action: pass by user returnval: 25634	<div>DeleteCopy</div>
3	Tue May 28 16:11:44 2024 pid: 25634 nr: 102(__NR_getuid) arg1: 0 arg2: 0 arg3: 0 arg4: 140731515088768 arg5: 0 ar g6: 139780488030944 action: pass by user returnval: 0	<div>DeleteCopy</div>

检查日志界面，发现成功将getpid调用改为getuid调用。

\*开发环境使用了root用户，故返回值为0，实际使用时软件并不需要root权限。在对应视频中使用普通用户演示。

Test5: 多进程程序1

说明：

程序分为两个进程，父进程死循环执行getuid，子进程死循环执行getpid。

目标程序主要源码：

```
int main()
{
    pid_t pid = fork();
    if(pid == 0)//child
    {
        while(1)
        {
            getpid();
            sleep(1);
        }
    }
    else
    {
        while(1)
        {
            getuid();
            sleep(1);
        }
    }
}
```

规则：

系统调用名	规则	用户操作	备注
getpid	notify	任意	
getuid	notify	任意	
clone	allow		见 <a href="#">注意事项</a>
fork	allow		见 <a href="#">注意事项</a>
vfork	allow		见 <a href="#">注意事项</a>
其他	allow forever		

过程：

Trace										
/home/user/playground/Test5				Args				Trace	Stop	Remote
	pid	nr	syscall	arg1	arg2	arg3	arg4	arg5	arg6	Options
1	50915	102	__NR_getuid	0	0	0	1396851652738...	0	1396851672583...	Select an action ▾
2	50921	39	__NR_getpid	0	0	0	1396851652448...	0	1396851651643...	Select an action ▾

启动程序，发现正确监控了父子进程各自的系统调用

Logs										
	Log									Options
1	Tue May 28 16:48:53 2024 pid: 50915 nr: 56(__NR_clone) arg1: 18874385 arg2: 0 arg3: 0 arg4: 139685165165072 arg5: 0 arg6: 139685167258336 action: pass according to rules returnval: 50921									Delete Copy
2	Tue May 28 16:49:22 2024 pid: 50915 nr: 102(__NR_getuid) arg1: 0 arg2: 0 arg3: 0 arg4: 139685165273848 arg5: 0 arg6: 139685167258336 action: pass by user returnval: 0									Delete Copy
3	Tue May 28 16:49:26 2024 pid: 50921 nr: 39(__NR_getpid) arg1: 0 arg2: 0 arg3: 0 arg4: 139685165244856 arg5: 0 arg6: 139685165164352 action: abort by user									Delete Copy
4	Tue May 28 16:49:30 2024 pid: 50921 nr: 39(__NR_getpid) arg1: 0 arg2: 0 arg3: 0 arg4: 140725288676960 arg5: 0 arg6: 139685165164352 action: pass by user returnval: 50921									Delete Copy
5	Tue May 28 16:49:33 2024 pid: 50921 nr: 39(__NR_getpid) arg1: 0 arg2: 0 arg3: 0 arg4: 140725288676960 arg5: 0 arg6: 139685165164352 action: abort by user									Delete Copy
6	Tue May 28 16:49:36 2024 pid: 50915 nr: 102(__NR_getuid) arg1: 0 arg2: 0 arg3: 0 arg4: 140725288676960 arg5: 0 arg6: 139685167258336 action: pass by user returnval: 0									Delete Copy

任意通过/拒绝各进程的系统调用后，查看日志，可以发现各进程行为正常。

Test6: 多进程程序2

说明:

子进程再创建子进程，观察运行情况

目标程序主要源码:

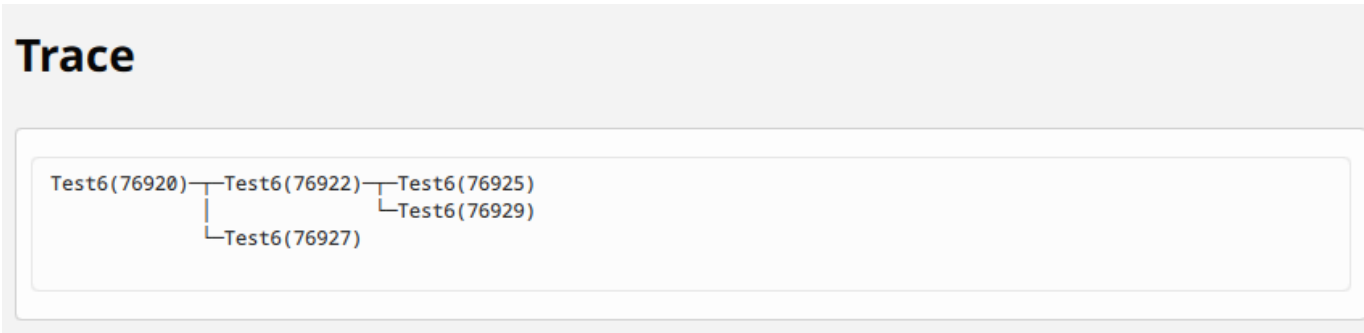
```
int main()
{
    int pid = fork();
    if(pid == 0)
    {
        int pid1 = fork();
```

```
        if(pid1 == 0)
        {
            getpid();
            while(1)sleep(1);
        }
        else
        {
            int pid2 = fork();
            if(pid2 == 0)
            {
                getpid();
                while(1)sleep(1);
            }
            else
            {
                getpid();
                while(1)sleep(1);
            }
        }
    }
    else
    {
        int pid3 = fork();
        if(pid3 == 0)
        {
            getpid();
            while(1)sleep(1);
        }
        else
        {
            getpid();
            while(1)sleep(1);
        }
    }
    return 0;
}
```

规则：

系统调用名	规则	用户操作	备注
getpid	allow		
clone	allow		见 <a href="#">注意事项</a>
fork	allow		见 <a href="#">注意事项</a>
vfork	allow		见 <a href="#">注意事项</a>
clock_nanosleep	abort forever		见 <a href="#">目前已知问题</a>
其他	allow forever		

过程:



启动程序，可以看到进程结构。

### Logs

	Log	Options
1	Tue May 28 17:31:48 2024 pid: 76920 nr: 56(__NR_clone) arg1: 18874385 arg2: 0 arg3: 0 arg4: 140235949767184 arg5: 0 arg6: 140235951860448 action: pass according to rules returnval: 76922	<div>DeleteCopy</div>
2	Tue May 28 17:31:48 2024 pid: 76922 nr: 56(__NR_clone) arg1: 18874385 arg2: 0 arg3: 0 arg4: 140235949767184 arg5: 0 arg6: 140235949766464 action: pass according to rules returnval: 76925	<div>DeleteCopy</div>
3	Tue May 28 17:31:48 2024 pid: 76920 nr: 56(__NR_clone) arg1: 18874385 arg2: 0 arg3: 0 arg4: 140235949767184 arg5: 0 arg6: 140235951860448 action: pass according to rules returnval: 76927	<div>DeleteCopy</div>
4	Tue May 28 17:31:48 2024 pid: 76925 nr: 39(__NR_getpid) arg1: 0 arg2: 0 arg3: 0 arg4: 140235949846968 arg5: 0 arg6: 140235949766464 action: pass according to rules returnval: 76925	<div>DeleteCopy</div>
5	Tue May 28 17:31:48 2024 pid: 76920 nr: 39(__NR_getpid) arg1: 0 arg2: 0 arg3: 0 arg4: 140235949846968 arg5: 0 arg6: 140235951860448 action: pass according to rules returnval: 76920	<div>DeleteCopy</div>
6	Tue May 28 17:31:48 2024 pid: 76927 nr: 39(__NR_getpid) arg1: 0 arg2: 0 arg3: 0 arg4: 140235949846968 arg5: 0 arg6: 140235949766464 action: pass according to rules returnval: 76927	<div>DeleteCopy</div>
7	Tue May 28 17:31:48 2024 pid: 76922 nr: 56(__NR_clone) arg1: 18874385 arg2: 0 arg3: 0 arg4: 140235949767184 arg5: 0 arg6: 140235949766464 action: pass according to rules returnval: 76929	<div>DeleteCopy</div>
8	Tue May 28 17:31:48 2024 pid: 76929 nr: 39(__NR_getpid) arg1: 0 arg2: 0 arg3: 0 arg4: 140235949846968 arg5: 0 arg6: 140235949766464 action: pass according to rules returnval: 76929	<div>DeleteCopy</div>

在日志界面可以看到正确追踪了各个进程的系统调用。

Test7: 主动执行系统调用测试

说明:

在目标程序运行在用户态时，插入一个系统调用

目标程序主要源代码:

```
int main()
{
    int a;
    while(1)
    {
        a = !a;
    }
}
```

```
}  
}
```

规则：

系统调用名	规则	用户操作	备注
getpid	allow		
其他	allow forever		

过程：

### Logs

Log	Options
-----	---------

启动程序后，进入死循环，没有任何系统调用。

### Trace

Test7(11942)

Pid: 11942 × +

syscall number and argc

39 × 0 ▾

Execute Clear

arg1: arg1

arg2: arg2

arg3: arg3

arg4: arg4

arg5: arg5

arg6: arg6

在主动执行界面输入39，点击execute执行该系统调用。

Logs		
	Log	Options
1	Tue May 28 18:24:57 2024 pid: 11942 nr: 39(__NR_getpid) arg1: 1 arg2: 140735645245752 arg3: 140735645245768 arg4: 139662602524760 arg5: 0 arg6: 139662602541792 action: pass according to rules re...	<div>DeleteCopy</div>

转到日志界面，发现出现了getpid的信息。

Logs		
	Log	Options
1	Tue May 28 18:24:57 2024 pid: 11942 nr: 39(__NR_getpid) arg1: 1 arg2: 140735645245752 arg3: 140735645245768 arg4: 139662602524760 arg5: 0 arg6: 139662602541792 action: pass according to rules re...	<div>DeleteCopy</div>
2	Tue May 28 18:27:20 2024 pid: 11942 nr: 39(__NR_getpid) arg1: 1 arg2: 140735645245752 arg3: 140735645245768 arg4: 139662602524760 arg5: 0 arg6: 139662602541792 action: pass according to rules re...	<div>DeleteCopy</div>
3	Tue May 28 18:27:21 2024 pid: 11942 nr: 39(__NR_getpid) arg1: 1 arg2: 140735645245752 arg3: 140735645245768 arg4: 139662602524760 arg5: 0 arg6: 139662602541792 action: pass according to rules re...	<div>DeleteCopy</div>
4	Tue May 28 18:27:23 2024 pid: 11942 nr: 39(__NR_getpid) arg1: 1 arg2: 140735645245752 arg3: 140735645245768 arg4: 139662602524760 arg5: 0 arg6: 139662602541792 action: pass according to rules re...	<div>DeleteCopy</div>

多次点击execute，发现日志界面出现多个getpid信息，说明主动执行系统调用后，正常恢复了上下文，程序仍在死循环内正常运行。

## 开发过程中遇到的问题

### seccomp的功能限制

读文档时发现了seccomp的unotify机制，该机制可以把seccomp拦截到的系统调用转发至其他进程处理，并且在执行fork、exec等操作时，seccomp的过滤器会被内核自动拷贝进新进程，所以在编写初版程序时，便采用了单seccomp的方式，并没有使用ptrace。但在后续开发过程中发现，由于seccomp只在系统调用入口暂停进程，所以无法读取到系统调用的返回值，所以改用seccomp生成ptrace事件，再用ptrace追踪。保证功能的同时，运行效率高于单独使用ptrace，也简化了编码难度。

### 不使用ptrace操作进程

在决定换用ptrace之前，为了获取返回值，试着使用过内核模块的方式在系统调用出口暂停进程，即阅读ptrace源码，找到其真实调用的内核函数，在自己的内核模块中劫持一个系统调用并使用这些函数。结果是一些只读的函数能正常工作（如获取寄存器的值），其他则会引起程序崩溃。究其原因，是因为在操作进程之前的各种加锁操作在自己的内核模块中没有实现。再继续往这个方向走下去，有些南辕北辙了，故放弃。不过在探索的过程中，也学到了诸如编译内核模块、劫持系统调用、如何寻找并使用内核没有导出的符号（函数）等等很有价值的内容。

## 目前已知问题

目标程序退出后，前端界面的重置没有做好。

小部分按钮按下后给用户的反馈不明确。

如果子进程再次fork后并立刻sleep可能导致捕获不到子进程的子进程。

## 未来扩展功能

□启动并监控远程计算机上的程序

□可以对每个子进程（线程）应用不同的规则文件

↓

□细化过滤规则。可以为系统调用创建一段脚本实现自定义规则(如判断呼叫系统调用的pid、参数的值、解引用参数后的值等等)来确定执行的操作

□中文支持

□设置页面，能够调整一些参数；关于页面，显示项目相关信息

□美化日志界面

## 参考资料

[ptrace](#)

[seccomp](#)

[waitpid](#)

[libseccomp](#)

为seccomp提供了更易用，可读性更高的api。项目贡献者Paul Moore先生也非常友善，多次耐心解答我的疑问。

[QML FluentUI](#)

基于QML实现，非常美观，功能强大的控件库。

[Stack Overflow](#)

许多邪门问题在上面找到了答案