

大作业-可行性报告

陈思睿 梁恒宇 吕泓涛 汤力宇
中国科学技术大学 安徽合肥

2021 年 4 月 18 日

目录

1	项目介绍	2
2	理论依据	2
2.1	恶意进程	2
2.1.1	linux 下的恶意进程	2
2.1.2	经典攻击方法	2
2.1.3	恶意进程行为	3
2.1.4	小结	3
2.2	docker 安全	3
2.3	macOS 沙盒分析	4
2.3.1	macOS 沙盒的权限设置	4
2.3.2	应用程序沙盒化	5
2.3.3	沙盒机制	5
2.3.4	macOS 沙盒优缺点	7
2.3.5	小结	7
3	技术依据	7
3.1	BPF 应用实现	7
3.1.1	一个简单的 BPF 程序	7
3.1.2	探针	9
3.1.3	seccomp 实现控制系统调用	11
3.1.4	小结	13
3.2	Linux 内核 BPF 源码的调研	13
3.2.1	修改 linux 中 BPF 源码的意义	13
3.2.2	linux 下 BPF verifier 部分的源码结构分析	13
3.2.3	linux 下 BPF helper 函数与 BPF program type 的相关分析	14
3.2.4	对内核代码的重编译和加载	14
4	技术路线	15
4.1	路线一：轻量化 bpf 沙盒实现方案	15
4.2	路线二：bpf 内实现进程虚拟化	15

4.3 路线三：bpf 优化用户态沙盒	15
---------------------------	----

1 项目介绍

使用新兴的 eBPF 架构，实现兼有安全性和性能的通用沙箱。

任何操作系统都或多或少的潜藏着安全漏洞，近年来各大主流 OS 都被爆出过存在重大安全隐患。当恶意程序侵入用户的系统，则可能破坏、窃取宝贵的用户数据，造成不可估量的损失。而为了防止此类事件发生，沙盒技术正在不断发展。沙盒技术通过对可疑的进程进行隔离与监控，防止其对系统其它部分造成损害。

然而随着恶意程序的攻击策略不断扩展，传统沙盒的安全性也难以长期保持，因此一个理想的沙盒应当在高效、安全同时拥有便于升级维护的特点，这一理想在现有的沙盒中难以实现。用户态的沙盒存在着大量到内核态的状态切换，带来了严重的性能损失，而内核态的沙盒则在损失了升级的灵活性的同时带来了更多可能被攻击的安全漏洞。

面对这一困境，本项目希望使用新兴的 eBPF 架构，实现一个可以从用户态灵活对其升级的内核态沙盒。由于 BPF 架构的设计特征，这一沙盒将不会给内核多带来额外的安全漏洞，并且将可以实现和内核态相仿的性能。

2 理论依据

2.1 恶意进程

2.1.1 linux 下的恶意进程

Linux 环境下的经典病毒种类有：[1]

- BillGates 基于僵尸网络的 DDOS 攻击
- DDG 蠕虫式挖矿
- SystemdMiner 蠕虫式挖矿
- StartMiner 蠕虫式挖矿
- WatchdogsMiner 蠕虫式挖矿
- XorDDos 基于僵尸网络的 DDOS 攻击
- RainbowMiner 蠕虫式挖矿

2.1.2 经典攻击方法

在这几篇文章 [2] [3] 中介绍了几种经典的恶意进程攻击手段：

- 操作系统中的一个用户态组件——动态装载器，负责装载二进制文件以及它们依赖的库文件到内存中。二进制文件使用动态装载器来支持导入符号的解析功能。有趣的是，这恰好就是一个面对加固应用的攻击者通过泄漏库地址与内容尝试“重塑”一个符号的表现。windows 下的 svchost.exe 攻击或者 linux 下的 elf 攻击都是利用了这个组件进行的攻击。

- 早期的栈溢出利用依赖于向缓冲区中注入二进制代码 (称为 shellcode) 的能力, 并需要覆盖在栈上的一个返回地址使其指向这个缓冲区。随后, 当程序从当前函数返回时, 执行流就会被重定向到攻击者的 shellcode, 接着攻击者就能取得程序的控制权。

2.1.3 恶意进程行为

卡巴斯基安全实验室在对一个新型 zeus 变种木马的报告中给出了如下的分析: [4]

- 此木马功能非常丰富, 包括使用 VNC 远程桌面控制电脑, 截屏并发送, 读取本地数据并发送, 读取用户所有操作并发送, 通过注册表开机自动启动, 监测是否处于沙盒环境并且在沙盒中自动停止活动, 监测注册表以防止自己的自动启动被清除。
- 此木马的传播主要通过邮件, 在恶意邮件中包含一个 doc 文件作为附件, 打开后 office 会提示需要启用宏来查看完整信息, 一旦用户点击启用宏后, 此木马会自动解码并且将自己复制进 svchost.exe, 然后其遍可以开始运行各种功能, 包括剩余模块的下载, 劫持浏览器, 窃取本地数据等。
- 此木马通过读取进程列表以判断是否有正在打开的浏览器, 然后当存在浏览器的时其会通过浏览器的安全漏洞劫持网页, 修改用户正在访问的银行网页, 截取其输入的密码、账号、pin、等内容并且发送给服务器。
- 此木马对沙盒有特殊的监测机制, 运行在沙盒内的时候系统中会出现若干特征性的监控类设备驱动, 当发现自己运行在沙盒中时, 木马将停止活动以防止自己被安全人员监测出来。

2.1.4 小结

此分析给我们带来了如下的提示:

- 针对某个具体的漏洞来设计沙盒是不切实际的, 随着系统体量的膨胀, 漏洞的存在是不可避免的, 至今 linux 和 windows 也无法完全杜绝对系统代码的恶意篡改。
- 各类资源的隔离对于沙盒的安全性至关重要, 必须确保沙盒内的进程只能访问有限的文件服务或者系统服务。
- 任何需要与被隔离进程直接交互的服务都有被污染的可能, 因此多层的隔离或者使用类似于 BPF 的无法污染的实现方式可以有效的提高对恶意进程的控制力度。

2.2 docker 安全

下面是 docker 安全性的具体实现方式 [5] [6]

1. 通过 namespace, 不同 docker 容器无法访问其他的进程, 在容器位置向系统请求进程列表会只能看到少数几个局部的容器内进程, 无法发现主机的其他进程。通过这种技术, 类似于 zeus 的劫持浏览器进程的木马难以危害主机安全。
2. 通过 namespace, 每个 docker 会被置入隔离的网络环境中, 对外的网络功能是通过在每个 docker 上运行虚拟的网卡并且以桥接模式 (默认) 与主机网卡链接来实现的。在这种情况下可以通过网络安全策略的方式直接控制容器进程的非法网络访问。

libnetwork: docker 的网络功能实现的具体技术

3. 利用 libcontainer (以及 namespace) 来实现了对文件系统的保护, libcontainer 中的 chroot 技术可以限制某个子系统对应的根目录 (rootFS), 即在容器内的进程来看, 当前 FS 的 root 就是实际所在的子目录, 因而其无法读取或访问主机上的其他文件。
4. cgroup (控制组) 是用于限制进程对 CPU、内存、网络带宽等运行资源的占用强度的, 其也可以用来限制容器内程序对设备的访问。不同的进程被组合成一个 cgroup, 作为一个整体参与资源的调度, 并且可以通过 cgroup 组策略来限制当前 group 可以占用多少资源。且 cgroup 可以嵌套, 一个 cgroup 里面可以包含多个子 cgroup。如整个 docker 可能被放在一个 cgroup 中以限制总资源使用量, 然后 docker 里面的每个容器中的进程也各自建立 cgroup, 参与划分 docekr-group 分配到的总的资源。
5. 联合文件系统 (Unionfs), 实质上概念很简单, 此文件系统不管理物理存储, 只是依赖于某一个通用的文件系统, 并且把不同文件夹的内容映射到同一个文件目录内。似乎是 docker 的重要组成部分。

在这篇文章中, 分析了 docker 结构的安全性: [7]

- 文章基于的模型如下: 当 hostOS 中运行的 docker 容器中有一部分被恶意进程完全控制了, 其可以对系统进行如 Denial-of-Service 和 Privilege escalation 的攻击。
- 为了在这种情况下保护系统安全, 容器应当做到如下几点:
 - process isolation 进程间的隔离
 - filesystem isolation 文件系统的隔离
 - device isolation 设备的隔离
 - IPC isolation 进程间通讯的隔离
 - network isolation 网络的隔离
 - limiting of resources 限制资源的使用量

对于路线一 (见4.1) 或路线三 (见4.3), 我们可以参考 docker 的安全策略, 此策略在各个角度上都有较好的安全性, 而且性能相当的高。

2.3 macOS 沙盒分析

2.3.1 macOS 沙盒的权限设置

macOS 沙盒中有如下的权限设置 [8]:

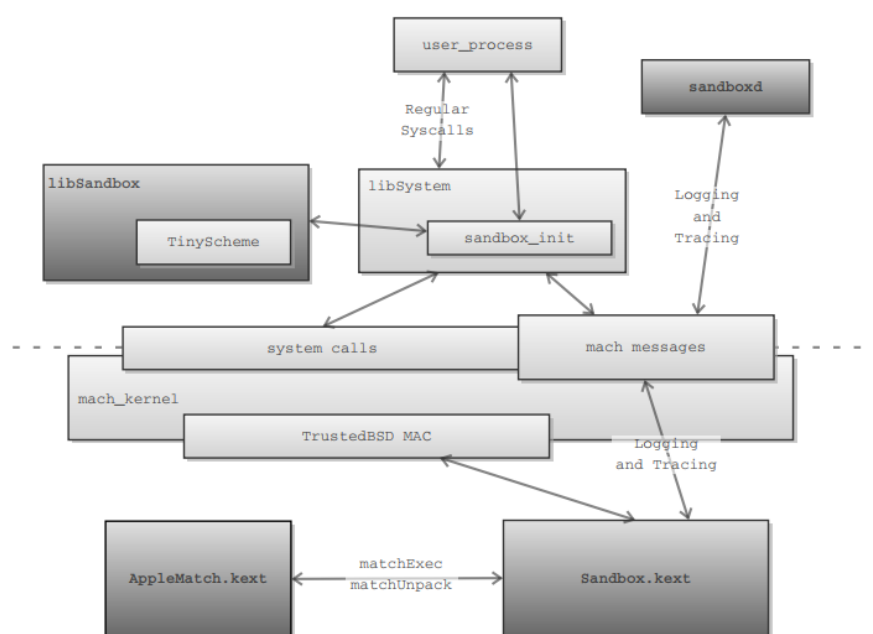
- Essentials: 是否使用访问控制技术
- Network: 接收和发送网络数据的权限
- Hardware: 硬件权限, 如照相机
- App Data: 访问地址簿, 位置, 日历的权限
- File Access: 其中设置是否拥有访问系统全部文件的权限已被剥离。读写限制十分严格, 只有用户主动操作才能实现一般文件的读写 (通过脱离程序语境的 Powerbox 机制——一个选择文件的对话框), 否则只有应用程序自己的目录是可见的。下载内容、图片、音乐、电影各有特殊限制规则。

2.3.2 应用程序沙盒化

- 在使用苹果的开发软件（如 Xcode）进行 build 时启用沙盒并进行沙盒权限确认即可。
- 建立于 MacOS 沙盒不是程序的义务，但却是上架 Mac App Store 的必要条件。就算上架 MAS，沙盒机制也是可以绕过去的。开发者在商店之外提供一个程序外挂供用户自由安装。由上架 MAS 的程序指挥外挂进行沙盒外操作即可。
- iOS 是一个从一开始就彻底应用沙盒机制的系统，正常情况下，所有 iOS 设备都只能安装 App Store 中的应用程序，而所有这些应用程序都彻底采用了沙盒机制，不能访问除自己目录之外的资源。虽然保证了安全性，但是也牺牲了一些便利性，如他们的应用程序无法调用其它程序配合完成任务，导致必须重复实现各种已有的功能（如 pdf 阅读功能）。

2.3.3 沙盒机制

沙盒机制见下图：



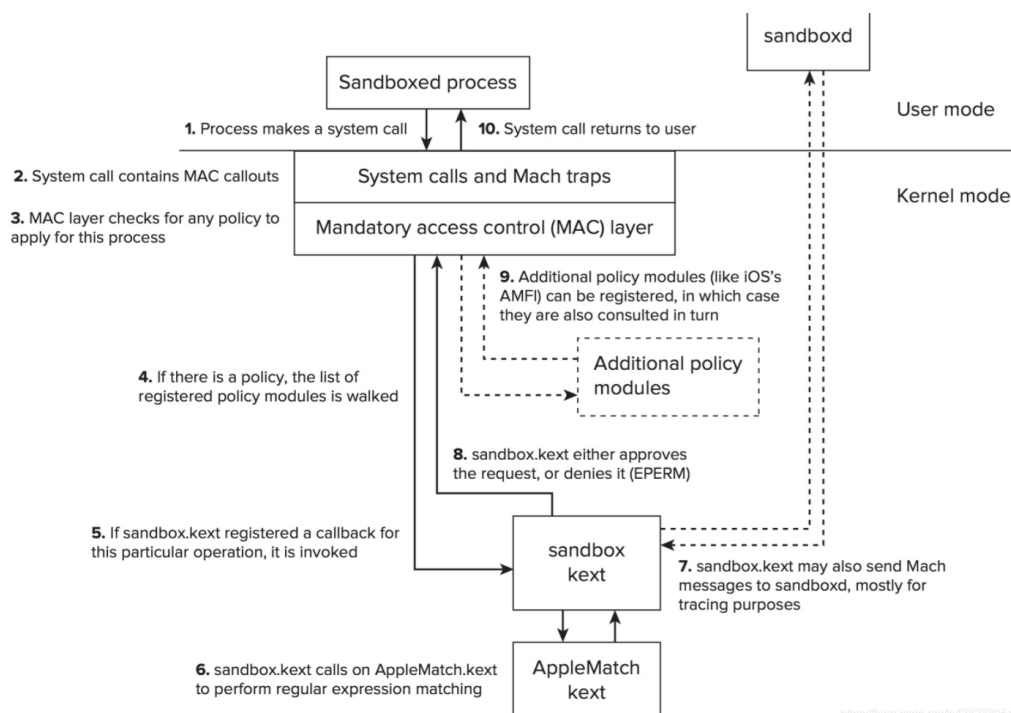


图 1: 沙盒机制 [9]

- 新的应用程序需要经过初始化。原先的、还未运行过的程序没有被 sandbox 约束，但是在初始化过程中，如果动态链接器确认了程序启用 sandbox，就会根据程序中的权限设置 (entitlements) 编译出 profile。在每次程序运行时，动态链接器都会在应用程序的语境中，根据这个 profile 初始化沙盒，然后才把控制权递交给程序。
- 流程
 - 1 进程尝试进行一次系统调用 (system call)，调用内核功能。
 - 2、3 MAC 层需要根据该进程的安全策略判断此次系统调用是否可以执行。
 - 4、5、6、7、8、9 如果存在策略的话，通过 sandbox.kext (提供 system call 的 hook 函数) 和 AppleMatch.kext (解析沙盒的 profile) 两个内核扩展实现权限的检查。
 - 10 返回调用结果
- XPC 服务
 - 是 MacOS 下的一种 IPC(进程间通信) 技术, 它实现了权限隔离, 使得 App Sandbox 更加完备。
 - 将程序隔离成多块能提高稳定性, 在某部分 crash 时, 主程序依然运作。
 - 沙盒中应用无法让子进程拥有比自身更高的权限, 但是开发者可以设定 XPC helpers, 它们各自运行在自己的沙盒中并拥有刚好供功能实现所需的权限。实现权限分割, 安全性就进一步提高了。
- 和 linux 下的 seccomp 有相似之处, 即在内核中对 system call 进行过滤。沙盒初始化就像 seccomp 程序启用。

2.3.4 macOS 沙盒优缺点

- 优点

1. 安全性高。不影响性能。
2. 开发者使用方便。
3. 对于用户需求的敏感访问有系统接管办法，如 Powerbox。

- 缺点

1. 自由度低。非 MAS 平台的应用，大部分没有采用 Sandbox 机制。
2. 沙盒有不可避免的漏洞：
 - 沙盒在应用程序的语境下初始化，如果程序偶然在沙盒初始化完成前就运行了代码，就存在保护失效的可能性。事实上 bug 确实发生了。[10]
 - 在规则内做出背离沙箱的行为。[11] 尽管 Apple 阻止应用程序注册全局接收分布式通知，但没有任何机制能够阻止沙箱应用程序注册接收任何通知。因此，如果按照名称，注册所有的分布式通知，恶意应用程序可以轻而易举地避开 Apple 针对沙箱的限制。尽管这需要一些额外的代码来实现，但其带来的影响是造成任何应用程序都可以通过遍历注册，来接收（捕获）到所有分布式通知，包括在沙箱中也是如此。

2.3.5 小结

Apple 系统的安全性很大程度上都无法离开一种强大的闭环生态，想在设备上运行就必须得遵守相应的约定。而 Linux 是一个非常开放的环境，就此而言，我们不可能让应用在开发的时候就强制开启沙盒，所以我们几乎是没办法实现一个兼顾安全性、性能、通用性的沙盒的。

对 macOS 沙盒结构的研究，对后续路线一（4.1）和路线三（4.3）均有启发。拦截系统调用是目前大多数沙盒的做法，在 Linux 下，我们可以借助 BPF 实现系统调用的拦截。同时，我们可以用编写安全策略加额外策略模块的方式来认证每次系统调用是否安全，若安全则允许调用，若不安全则拒绝调用。

3 技术依据

3.1 BPF 应用实现

3.1.1 一个简单的 BPF 程序

运行 BPF 程序需要两个部分，一个是 BPF 内核程序，一个是用户态的加载程序。BPF 内核程序需要编译成 BPF 内核字节码，用户态程序只需要编译成可执行文件。[12]

加载进内核的程序可以使用 C 代码编写，示例 C 代码如下：

```
1 #include <linux/bpf.h>
2 #define SEC(NAME) __attribute__((section(NAME), used))
3
4 static int (*bpf_trace_printk)(const char *fmt, int fmt_size,
5                                ...) = (void *)BPF_FUNC_trace_printk;
```



```

6
7 SEC("tracepoint/syscalls/sys_enter_execve")
8 int bpf_prog(void *ctx) {
9     char msg[] = "Hello, World!";
10    bpf_trace_printk(msg, sizeof(msg));
11    return 0;
12 }
13
14 char _license[] SEC("license") = "GPL";

```

代码 1: BPF 内核程序代码

程序被设置为在其他程序开始执行时被调用，并输出“Hello World!”。
它将被编译成字节码：

```

1 0000000000000000 bpf_prog:
2      0:      b7 01 00 00 21 00 00 00 r1 = 33
3      1:      6b 1a fc ff 00 00 00 00 *(u16 *) (r10 - 4) = r1
4      2:      b7 01 00 00 6f 72 6c 64 r1 = 1684828783
5      3:      63 1a f8 ff 00 00 00 00 *(u32 *) (r10 - 8) = r1
6      4:      18 01 00 00 48 65 6c 6c 00 00 00 00 6f 2c 20 57 r1 = 6278066737626506568 11
7      6:      7b 1a f0 ff 00 00 00 00 *(u64 *) (r10 - 16) = r1
8      7:      bf a1 00 00 00 00 00 00 r1 = r10
9      8:      07 01 00 00 f0 ff ff ff r1 += -16
10     9:      b7 02 00 00 0e 00 00 00 r2 = 14
11    10:      85 00 00 00 06 00 00 00 call 6
12    11:      b7 00 00 00 00 00 00 00 r0 = 0
13    12:      95 00 00 00 00 00 00 00 exit

```

代码 2: BPF 字节码

用户态的加载程序如下，它基本上只起一个加载 BPF 程序的作用：

```

1 #include "bpf_load.h"
2 #include <stdio.h>
3
4 int main(int argc, char **argv) {
5     if (load_bpf_file("bpf_program.o") != 0) {
6         printf("The kernel didn't load the BPF program\n");
7         return -1;
8     }
9
10    read_trace_pipe();
11
12    return 0;
13 }

```

代码 3: BPF 加载程序

执行 BPF 程序，需要以管理员身份调用该加载程序。程序的执行效果如下图所示：


```

<...>-325497 [001] .... 13609.129484: 0: Hello, World! <..
.->-325498 [002] .... 13609.129564: 0: Hello, World!
<...>-325499 [003] .... 13610.195734: 0: Hello, World!
<...>-325500 [000] .... 13610.197371: 0: Hello, World!
<...>-325501 [002] .... 13610.197623: 0: Hello, World!
<...>-325502 [002] .... 13611.266022: 0: Hello, World!
<...>-325503 [003] .... 13611.267731: 0: Hello, World!
<...>-325504 [000] .... 13611.267816: 0: Hello, World!
<...>-325505 [001] .... 13612.336308: 0: Hello, World!
<...>-325506 [002] .... 13612.338039: 0: Hello, World!
<...>-325507 [003] .... 13612.338171: 0: Hello, World!
<...>-325508 [001] .... 13613.405017: 0: Hello, World!
<...>-325509 [003] .... 13613.406619: 0: Hello, World!
<...>-325510 [000] .... 13613.406740: 0: Hello, World!
<...>-325511 [003] .... 13614.476889: 0: Hello, World!
<...>-325512 [000] .... 13614.478580: 0: Hello, World! <..
.->-325513 [001] .... 13614.478669: 0: Hello, World!
<...>-325514 [001] .... 13615.598697: 0: Hello, World!
<...>-325515 [002] .... 13615.600209: 0: Hello, World!
<...>-325516 [003] .... 13615.600332: 0: Hello, World!
<...>-325517 [001] .... 13616.668419: 0: Hello, World!
<...>-325518 [002] .... 13616.670007: 0: Hello, World!
<...>-325519 [003] .... 13616.670134: 0: Hello, World!

```

图 2: 示例程序执行效果

可以看到，每当一个系统中有新的程序开始执行时，该 BPF 程序就会被调用，并输出一个 Hello World!。

3.1.2 探针

BPF 程序可用于跟踪可执行文件中的某个符号是否被执行。

例如，我们使用这样一个简单的 Hello World! 程序进行测试：

```

1 #include<stdio.h>
2
3 int main(){
4     printf("Hello World!\n");
5     return 0;
6 }

```

代码 4: Hello World! 程序

使用 nm 工具可看到可执行文件中包含的符号（有省略）：

```

1          U abort@@GLIBC_2.17
2 0000000000011018 B __bss_end__
3 0000000000011018 B __bss_end__
4 0000000000011010 B __bss_start
5 0000000000011010 B __bss_start__
6 ...
7 0000000000000790 T __libc_csu_init
8          U __libc_start_main@@GLIBC_2.17
9 000000000000076c T main
10          U puts@@GLIBC_2.17
11 00000000000006e0 t register_tm_clones
12 0000000000000660 T _start
13 0000000000011010 D __TMC_END__

```

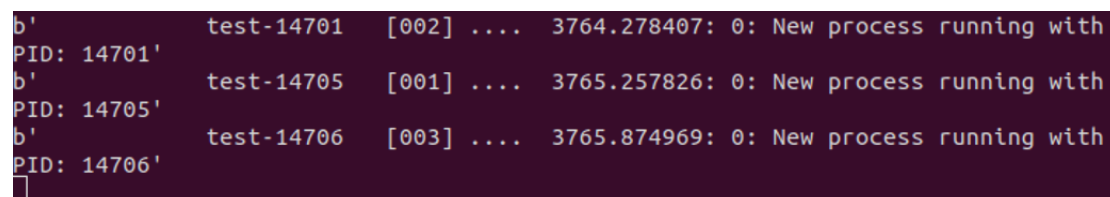
代码 5: 符号表

Hello World! 程序被命名为“test”，我们跟踪它的 main 符号，跟踪程序如下：

```
1 from bcc import BPF
2
3 bpf_source = """
4 int trace_go_main(struct pt_regs *ctx) {
5     u64 pid = bpf_get_current_pid_tgid();
6     bpf_trace_printk("New process running with PID: %d\\n", pid);
7     return 0;
8 }
9 """
10
11 bpf = BPF(text = bpf_source)
12 bpf.attach_uprobe(name = "./test", sym = "main", fn_name = "trace_go_main")
13 bpf.trace_print()
```

代码 6: 符号跟踪程序

bpf.attach_uprobe 中的 name 和 sym 即为要跟踪的程序和要跟踪的符号。运行程序三次，每次运行都会有相应的跟踪结果出现：



```
b' test-14701 [002] .... 3764.278407: 0: New process running with
PID: 14701'
b' test-14705 [001] .... 3765.257826: 0: New process running with
PID: 14705'
b' test-14706 [003] .... 3765.874969: 0: New process running with
PID: 14706'
```

图 3: 符号跟踪结果

对 test 程序进行修改，让程序根据输入的数字决定执行的函数：

```
1 #include<stdio.h>
2
3 void odd(){
4     printf("odd number!\\n");
5 }
6
7 void even(){
8     printf("even number!\\n");
9 }
10
11 int main(){
12     int i;
13     scanf("%d", &i);
14     if (i%2 == 0){
15         even();
16     } else {
17         odd();
18     }
19     return 0;
20 }
```

代码 7: 修改后程序

此时该程序具有 main、even 和 odd 三个符号。

```

1      U abort@@GLIBC_2.17
2 0000000000011018 B __bss_end__
3 0000000000011018 B __bss_end__
4 ...
5 0000000000011018 B _end
6 00000000000008ac T even
7 00000000000009d4 T _fini
8 0000000000000888 t frame_dummy
9 ...
10      U __libc_start_main@@GLIBC_2.17
11 00000000000008cc T main
12 000000000000088c T odd
13      U puts@@GLIBC_2.17
14 0000000000000800 t register_tm_clones
15      U __stack_chk_fail@@GLIBC_2.17
16      U __stack_chk_guard@@GLIBC_2.17
17 0000000000000780 T _start
18 0000000000011010 D __TMC_END__

```

代码 8: 符号表

我们对 even 符号进行跟踪, 此时, 当且仅当输入偶数时, BPF 才会被触发, 显示如下结果:

```

b' test-16412 [002] .... 4351.014742: 0: New process running with
PID: 16412'

```

图 4: 符号跟踪结果

3.1.3 seccomp 实现控制系统调用

我们编写的 seccomp 程序实际上是一种“过滤器”, 类似“正则表达式”。它会利用 BPF 将过滤器程序加载到内核当中监测被监控程序的系统调用, 一旦该程序的系统调用匹配上了你编写的格式, 对该被监控程序的一些行为就会被触发。

seccomp 在头文件 <linux/seccomp.h> 中定义了有限的一些控制被监控的进程的操作。

```

1 #define SECCOMP_RET_KILL_PROCESS 0x80000000U /* kill the process */
2 #define SECCOMP_RET_KILL_THREAD 0x00000000U /* kill the thread */
3 #define SECCOMP_RET_KILL SECCOMP_RET_KILL_THREAD
4 #define SECCOMP_RET_TRAP 0x00030000U /* disallow and force a SIGSYS */
5 #define SECCOMP_RET_ERRNO 0x00050000U /* returns an errno */
6 #define SECCOMP_RET_USER_NOTIF 0x7fc00000U /* notifies userspace */
7 #define SECCOMP_RET_TRACE 0x7ff00000U /* pass to a tracer or disallow */
8 #define SECCOMP_RET_LOG 0x7ffc0000U /* allow after logging */
9 #define SECCOMP_RET_ALLOW 0x7fff0000U /* allow */

```

代码 9: seccomp 头文件

例如, 使用 `SECCOMP_RET_KILL_PROCESS` 会关闭该程序, 使用 `SECCOMP_RET_ERRNO` 会拒绝该程序相关的系统调用。

这里有如下的一个示例程序, 这个程序会拒绝被调用程序的所有和写相关的系统调用。

```
1  #include <errno.h>
2  #include <linux/audit.h>
3  #include <linux/bpf.h>
4  #include <linux/filter.h>
5  #include <linux/seccomp.h>
6  #include <linux/unistd.h>
7  #include <stddef.h>
8  #include <stdio.h>
9  #include <stdlib.h>
10 #include <sys/prctl.h>
11 #include <unistd.h>
12
13 static int install_filter(int nr, int arch, int error) {
14     struct sock_filter filter[] = {
15         BPF_STMT(BPF_LD + BPF_W + BPF_ABS, (offsetof(struct seccomp_data, arch))),
16         BPF_JUMP(BPF_JMP + BPF_JEQ + BPF_K, arch, 0, 3),
17         BPF_STMT(BPF_LD + BPF_W + BPF_ABS, (offsetof(struct seccomp_data, nr))),
18         BPF_JUMP(BPF_JMP + BPF_JEQ + BPF_K, nr, 0, 1),
19         BPF_STMT(BPF_RET + BPF_K, SECCOMP_RET_ERRNO | (error & SECCOMP_RET_DATA)),
20         BPF_STMT(BPF_RET + BPF_K, SECCOMP_RET_ALLOW),
21     };
22     struct sock_fprog prog = {
23         .len = (unsigned short)(sizeof(filter) / sizeof(filter[0])),
24         .filter = filter,
25     };
26     if (prctl(PR_SET_SECCOMP, SECCOMP_MODE_FILTER, &prog)) {
27         perror("prctl(PR_SET_SECCOMP)");
28         return 1;
29     }
30     return 0;
31 }
32
33 int main(int argc, char const *argv[]) {
34     if (prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0)) {
35         perror("prctl(NO_NEW_PRIVS)");
36         return 1;
37     }
38     install_filter(__NR_write, AUDIT_ARCH_AARCH64, EPERM);
39     return system(argv[1]);
40 }
```

代码 10: seccomp 程序

编译好这个程序, 我们使用这个程序调用一个需要进行写操作的程序 `ls`, 同时我们还需要使用 `strace` 来跟踪 `ls` 所有的系统调用。执行的部分结果如下:

```

1 [pid 319933] write(1, "total 32\ndrwxrwxr-x 2 lhy lhy 40"... , 339) = -1 EPERM (Operation not
   permitted)
2 [pid 319933] close(1)                                = 0
3 [pid 319933] write(2, "ls: ", 4)                      = -1 EPERM (Operation not permitted)
4 [pid 319933] write(2, "write error", 11) = -1 EPERM (Operation not permitted)
5 ...
6 [pid 319933] write(2, ": Operation not permitted", 25) = -1 EPERM (Operation not permitted)
7 [pid 319933] write(2, "\n", 1)                        = -1 EPERM (Operation not permitted)

```

代码 11: seccomp 程序执行结果

为了节省空间，大部分程序的输出都被省略掉了，完整结果可见代码??。

可以看到，ls 正确执行出了通常的运行结果："total 32\ndrwxrwxr-x 2 lhy lhy 40"，但是执行的 write 系统调用均被拒绝了，所以在终端并不会看到任何的输出。此外，程序并没有被 kill 掉，调用了不允许的系统调用后，还在正常运行。

3.1.4 小结

BPF 提供了非常多的跟踪功能，它能高效成为一个内核中的 profiler，这是我们接下来将引出的路线一（见4.1）或路线三（见4.3）的主要技术依据，在后文中会对技术依据进行更详细的解释。

但是我们也能注意到，类似 seccomp 的 BPF 安全应用有非常大的局限性。如果我们的过滤机制不够完善，它在阻止病毒程序的运行的同时，还阻止了正常程序的运行。这也是我们改进想法由来的原因之一，同时，我们据此才有了另一个思路。下一小节中我们会尝试解除当前 BPF 的一些限制，让 BPF 有更完善的功能，支持我们沙盒的运行。

3.2 Linux 内核 BPF 源码的调研

路线一（4.1）或者路线二（4.2）中希望实现的沙盒程序的复杂度可能难以直接载入现有的 linux eBPF，为了实现这些功能，我们将有必要对 linux 的 eBPF 代码进行修改或者需要定制自己的 eBPF 内核模块。为此，我们对 linux 内核中的 BPF 源码进行了简单的调研。具体的，我们研究了如何修改 verifier 模块，如何添加更多的 helper 函数，

3.2.1 修改 linux 中 BPF 源码的意义

BPF 的程序的功能丰富度是和内核对 BPF 程序的支持直接相关的，例如每个 BPF 程序的输入输出规范都是直接以模板形式定义在内核源码中的，而 BPF 程序可以使用的系统调用也是直接在 linux/kernel/bpf/helper.c 中直接定义的，因此实现诸如劫持系统调用等的功能讲不可避免的会涉及到对内核中 BPF 源码的修改或重写。我们也有可能会在后续的工作中发现我们的程序在现有的 verifier 体系下难以得到认可，例如我们在 bpf 中实现的文件系统管理服务可能会被 verifier 认定为不会终止，于是被拒绝载入到内核中，这种情况下我们便会需要修改 verifier 对软件安全性的认证过程，用更加智能的检测手段来检查 bpf 程序的安全性，从而防止我们的程序被错误的认定为不安全。

3.2.2 linux 下 BPF verifier 部分的源码结构分析

与 linux 中大部分源码相似，BPFverifier 的头文件位于 linux/include/linux/bpf_verifier.h，原文件位于 linux/kernel/bpf/verifier.c。头文件中最重要的数据结

构为 `struct bpf_verifier_env`，这个数据结构作为 `verify` 过程中最核心的结构出现在了源文件相关函数中。此数据结构中包括了程序源码，各个验证流程的状态（通过与否）等，后文中的 `check` 函数中主要通过传递 `env` 变量来进行各级的验证流。而原文件中最重要的函数为 `bpf_check()`，承载了验证 `bpf` 安全性的主要功能，包括代码的载入，对代码规模、运行路线、最坏运行时间的分析等各项检查分别在此函数中被调用。其中涉及到的最大代码长度、最长运行时间等重要常数通过宏定义的方式定义在此文件或 `bpf.c` 中。

```

10883 int bpf_check(struct bpf_prog **prog, union bpf_attr *attr,
10884               union bpf_attr __user *uattr)
10885 {
10886     u64 start_time = ktime_get_ns();
10887     struct bpf_verifier_env *env;
10888     struct bpf_verifier_log *log;
10889     int i, len, ret = -EINVAL;
10890     bool is_priv;
10891     .
10892     /* no program is valid */
10893     if (ARRAY_SIZE(bpf_verifier_ops) == 0)
10894         return -EINVAL;
10895
10896     /* 'struct bpf_verifier_env' can be global, but since it's not small,
10897      * allocate/free it every time bpf_check() is called
10898      */
10899     env = kzalloc(sizeof(struct bpf_verifier_env), GFP_KERNEL);
10900     if (!env)
10901         return -ENOMEM;
10902     log = &env->log;
10903 }

```

图 5: `bpf_check` 函数

3.2.3 linux 下 BPF helper 函数与 BPF program type 的相关分析

linux 下有关 helper 函数的主要定义都位于 `linux/kernel/bpf/helpers.c` 中，其中对 helper 函数的格式、类型以及具体的实例有着详细的定义。我们在这段源码中发现了诸如 `__bpf_spin_lock`，和 `mmap` 抽象存储相关的各种 helper 函数的具体实现，在后续的工作中，我们可以直接将我们需要的 helper 函数添加在此处来实现我们需要的功能。而有关 BPF program type 的主要代码都位于 `linux/include/bpf_types.h` 中，其中定义了常用的诸如 `BPF_PROG_TYPE_SOCKET_FILTER`，`BPF_PROG_TYPE_XDP` 等 BPF 程序及其相关输入输出参数类型，这些程序类型的声明方式对我们的代码编写有着重要的指导意义。

```

5 BPF_PROG_TYPE(BPF_PROG_TYPE_SOCKET_FILTER, sk_filter,
6               struct __sk_buff, struct sk_buff)
7 BPF_PROG_TYPE(BPF_PROG_TYPE_SCHED_CLS, tc_cls_act,
8               struct __sk_buff, struct sk_buff)
9 BPF_PROG_TYPE(BPF_PROG_TYPE_SCHED_ACT, tc_cls_act,
10              struct __sk_buff, struct sk_buff)
11 BPF_PROG_TYPE(BPF_PROG_TYPE_XDP, xdp,
12              struct xdp_md, struct xdp_buff)

```

图 6: `bpf_types.h`

3.2.4 对内核代码的重编译和加载

与一般的模块化加载方式不同，`bpf` 相关代码只能在编译之初就集成于内核中，因此修改 BPF 源码后，有必要对内核进行整体重新编译。linux 对内核更新的支持非常友好，其允许用户在系统中完成新内核的编译和加载，然后直接通过重启以切换到新内核。具体的，

完成源码的修改后,通过正常的编译指令可以将内核编译完成,依次完成新版本模块的装载和新版本内核的装载后,完成 boot 目录下 grub 文件的修改,最后通过 reboot 重启以进入新版本的内核。

4 技术路线

在现阶段,我们提出了三种具体的实现本项目的路线。我们后续会选择最优的路线。

4.1 路线一:轻量化 bpf 沙盒实现方案

这一路线中,我们希望借鉴 docker 的隔离结构,尽可能使沙盒本体实现轻量化和高效。这一路线中,bpf 的运行模式与 seccomp 中拦截系统调用的做法类似,举例来说我们可以通过使 bpf 程序拦截用户进程的文件类系统调用,将其中的具体目录信息篡改后转发给系统,如将用户进程请求的 read C:/root/ 的访问请求拦截篡改改为 D:/user/sandbox/ins1/C/root 来进行实际的执行,通过这种手段可以实现 docker 中的 fakeroot 技术。我们希望使用类似的方式在 bpf 环境下实现 docker engine 中的各项隔离技术,以实现和 docker 同等的隔离性,并且可以有望实现更高的安全性和性能。

优点:实现较为简单,沙盒本体较为轻量化。

缺点:沙盒层数较少,故一旦被恶意进程攻破会存在着危害主机系统的可能。

4.2 路线二:bpf 内实现进程虚拟化

这一路线中,我们希望借鉴 gVisor 的多层沙盒结构,尽可能使沙盒本体实现最大化的安全性和相对 gVisor 更高的运行效率 gVisor 维护了一套虚拟化的内核服务,例如沙盒内的局部文件系统。这种结构设计比路线一可以实现更好的安全性,恶意进程将难以同时捕捉到局部文件系统和主机文件系统的漏洞,从而增大其污染主机系统的难度。本路线中我们希望在 bpf 中实现局部的文件系统或局部的动态装载器等系统服务,这种方式能够实现更加完备的隔离性,减少沙盒与主机系统的实际交互。使用 bpf 结构可以比 gvisor 更加容易和高效的捕捉拦截转发沙盒进程的系统调用,同时 bpf 结构几乎没有被篡改代码内容的危险,从而实现了比 gvisor 更进一步的安全性。

优点:最大化的安全性,比 gVisor 等同类原理沙盒更高的效率。

缺点:体量较大,实现复杂度高。

4.3 路线三:bpf 优化用户态沙盒

这一路线为备选路线,当对内核 bpf 模块修改失败后会转为使用此路线。这一路线依据于 gVisor 等用户态沙盒会在劫持系统调用过程中损失大量的性能这一特征。BPF 可以作为优秀的监控拦截模块,使用现有的 BPF 框架即可高效简单的实现 gVisor 中需要的劫持系统调用这一步骤,可以同时优化 gVisor 的性能与安全性,选择此路线时我们将选择某个现有的用户态沙盒应用,着手使用 bpf 结构优化其中的部分功能。

优点:代码难度较低。

缺点:相较现有的成熟解决方案缺少突破性的创新,无法做到根本性的优化。

参考文献

- [1] “An introduction to common linux viruses.” Website, 2020. <https://segmentfault.com/a/1190000022761270>.
- [2] A. Di Federico, A. Cama, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, “How the {ELF} ruined christmas,” in *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pp. 643–658, 2015.
- [3] K. Rieck, T. Holz, C. Willems, P. Düssel, and P. Laskov, “Learning and classification of malware behavior,” in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 108–125, Springer, 2008.
- [4] “Revealed bank trojan chthonic: the latest variant of the online silver thief zeus.” Website, 2018. <https://cloud.tencent.com/developer/article/1036506>.
- [5] C. Boettiger, “An introduction to docker for reproducible research,” *ACM SIGOPS Operating Systems Review*, vol. 49, no. 1, pp. 71–79, 2015.
- [6] B. B. Rad, H. J. Bhatti, and M. Ahmadi, “An introduction to docker and analysis of its performance,” *International Journal of Computer Science and Network Security (IJCSNS)*, vol. 17, no. 3, p. 228, 2017.
- [7] T. Bui, “Analysis of docker security,” *arXiv preprint arXiv:1501.02967*, 2015.
- [8] A. Hoog and K. Strzempka, *iPhone and iOS forensics: Investigation, analysis and mobile security for Apple iPhone, iPad and iOS devices*. Elsevier, 2011.
- [9] D. Blazakis, “The apple sandbox,” *Arlington, VA, January*, 2011.
- [10] M. Blochberger, J. Rieck, C. Burkert, T. Mueller, and H. Federrath, “State of the sandbox: Investigating macos application security,” in *Proceedings of the 18th ACM Workshop on Privacy in the Electronic Society*, pp. 150–161, 2019.
- [11] “[0day] mojave’s sandbox is leaky.” Website, 2018. https://objective-see.com/blog/blog_0x39.html.
- [12] D. Calavera and L. Fontana, *Linux Observability with BPF: Advanced Programming for Performance Analysis and Networking*. O’Reilly Media, 2019.