НИУ "Высшая школа экономики"
Московский институт электроники и математики"

# Описание программы

и её применения

Юлдашев А. Х.

Москва 2024

# Содержание

# Введение

Для написания и тестирования программы были установлены:

- Заголовочные и исходные файлы ядра Linux версии 6.1;
- libbpf пакет версии 1.4 (с оптимизацией llvm);

Настройки окружения:

- Операционная система Arch Linux;
- Ядро версии 6.9.3;
- gcc версии 14.1;

В дальнейшем понадобиться дампить объектные файлы для чтения их инструкций, для этого применяется:

```
llvm-objdump-14 -d *.bpf.o
или
llvm-objdump -d *.bpf.o
```

Модуль (kernel security module) deny_unshare применяется для контроля системного вызова unshare, при создании нового пространства.

Для использования модуля необходимо:

1. Собрать все необходимые пакеты для компиляции BPF программ.
2. Собрать исполняемый файл.
3. запустить исполняемый файл:

```
./<file_name>
```

Для прекращения работы программы нужно послать сигнал SIGINT/INT (например, просто в терминале процесса нажать Ctrl+C).

Для тестирования работоспособности программы применяются попытки создания нового пространства.

# 1 Теория. Основы.

Данный документ не затрагивает развитие мысли вредоносности неконтролируемого создания пространств любым пользователем, поэтому ограничимся тем, что это приводит к большой проблеме безопасности для НОВЫХ пользователей подобных систем. Для примера применения приведу возможность "масштабирования"подобного пространства для того, что бы новые пользователи вносили свои данные в уже захваченную среду/систему. Так же, благодаря этому можно "выйти"из ограниченного окружения, что подробно описано в этой статье. Начну с пояснения используемых инструментов:

1. Технология BPF (см. главу Теория. BPF)
2. Технология LSM (см. главу Теория. LSM)

Для дальнейшего чтения стоит ознакомиться с соответствующими вышему главами. Для работы этих двух инструментов была использована внутренняя возможность LSM и BPF, которые изначально позволяют удобно внедрять BPF инструкции к LSM хукам.

## 1.1 Теория. BPF.

**Введение**

Технология BPF (Berkeley Packet Filter) изначально была разработана для фильтрации пакетов в сетевых приложениях, таких как tcpdump. Однако со временем BPF значительно эволюционировал, особенно в контексте ядра Linux, превратившись в мощный механизм, используемый для различных целей, включая мониторинг производительности, сетевую безопасность и диагностику.

**Исходная концепция**

- BPF был представлен в 1992 году как эффективный способ фильтрации пакетов в операционной системе BSD.
- Исходный BPF включал в себя байт-код, который выполняется в виртуальной машине внутри ядра, что позволяло быстро и эффективно обрабатывать сетевые пакеты.
- Изолированность среды выполнения BPF инструкций (байт-код, описанный выше).

**eBPF: Расширенный BPF**

eBPF (extended BPF) — это значительно расширенная версия оригинального BPF:

- Введение в ядре Linux: eBPF был интегрирован в ядро Linux начиная с версии 3.15.
- Расширение функциональности: eBPF позволяет выполнять произвольный код в ядре, что открывает возможности для различных типов мониторинга и обработки данных.
- Поддержка новых типов карт eBPF (BPF maps), которые являются ключевым элементом для хранения и обмена данными между программами eBPF и пользовательским пространством.
- Расширенная поддержка проб eBPF. Пробы (probes) eBPF используются для динамической вставки точек отслеживания в код ядра.
- BPF Type Format (BTF) — это новый формат, позволяющий программам eBPF получать доступ к информации о типах данных в ядре Linux.

* Остальные инновации нас интересовать будут меньше.

**Архитектура и компоненты eBPF**

1. Программы eBPF:
   - Написаны на высокоуровневом языке, таком как C, и затем компилируются в байт-код eBPF.
   - Загружаются в ядро через системные вызовы и могут быть прикреплены к различным точкам в ядре (например, к сетевым событиям, системным вызовам, трассировочным точкам).
2. BPF-карты (BPF maps):
   - Представляют собой структуры данных в ядре, используемые для обмена информацией между программами eBPF и пользовательским пространством.
   - Поддерживают различные типы данных, такие как хэш-таблицы, массивы, счетчики.
3. Системные вызовы для работы с eBPF:
   - bpf(): основной системный вызов для загрузки, управления и взаимодействия с программами и картами eBPF.

Технология BPF, а особенно её расширенная версия eBPF, представляет собой мощный инструмент для мониторинга, диагностики и управления системами на уровне ядра. Благодаря своей гибкости и производительности, eBPF находит широкое применение в современных вычислительных системах, предоставляя разработчикам и администраторам новые возможности для управления и оптимизации работы систем.

## 1.2 Теория. LSM.

**Введение**

LSM (Linux Security Modules) — это инфраструктура в ядре Linux, предназначенная для поддержки различных моделей безопасности. Она позволяет разработчикам создавать и интегрировать собственные модули безопасности, предоставляя механизмы для контроля доступа и безопасности на уровне ядра. Вот подробный обзор технологии LSM.

**Цели LSM**

- Предоставление универсального API для реализации различных политик безопасности.
- Обеспечение механизмов для контроля доступа к системным ресурсам, таким как файлы, процессы, сети и память.
- Позволяет внедрять модули безопасности без необходимости модификации ядра.

**Архитектура и компоненты LSM**

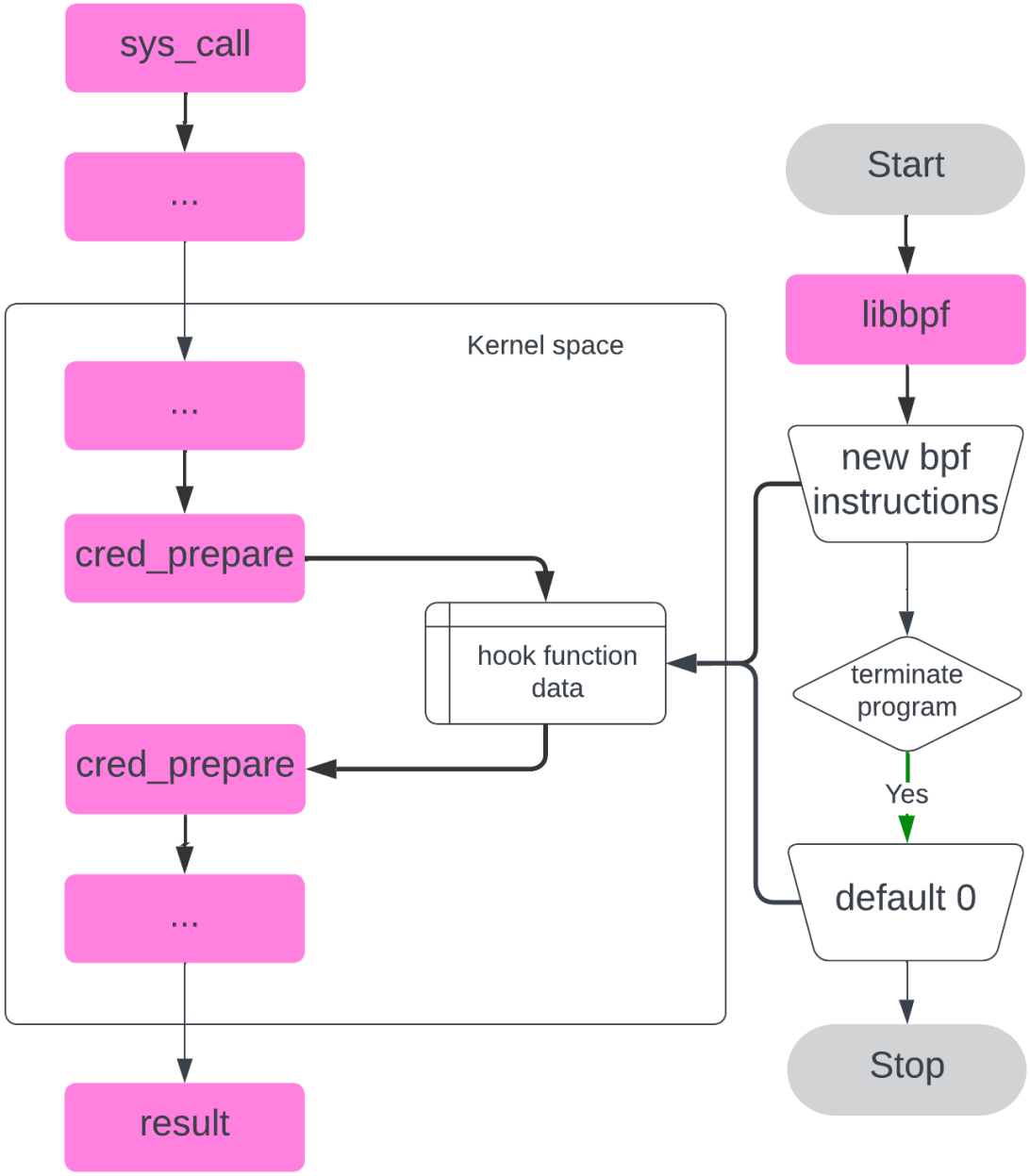
eBPF (extended BPF) — это значительно расширенная версия оригинального BPF:

1 LSM Hooks (крючки LSM):
   - Специальные точки в коде ядра, где могут быть вызваны функции безопасности.
   - Позволяют модулям безопасности внедрять свои проверки и решения в стандартные операции ядра, такие как создание файлов, доступ к памяти, межпроцессное взаимодействие и сетевые операции.

2 LSM Модули:
   - Наборы правил и политик безопасности, которые используют LSM hooks для контроля доступа.
   - Примеры популярных LSM модулей:
     - SELinux (Security-Enhanced Linux): предоставляет гибкую и мощную модель контроля доступа на основе меток.
     - AppArmor: пспользует профили для ограничения возможностей процессов на уровне пути к файлу.
     - И другие.

LSM — это мощный и гибкий механизм, предоставляющий разработчикам и администраторам возможность внедрения и управления различными моделями безопасности в Linux.

## 2 Общая схема интересующих процессов ядра




На данной схеме троеточием (...) обозначается незначительный процесс. Схема придерживается стандарта ГОСТ 19.701-90.

## 3 Определение хука

LSM (Linux Security Modules) предоставляет множество хуков, которые позволяют модулям безопасности внедрять свои политики и проверки в различные части ядра Linux. Эти хуки охватывают широкий спектр операций, таких как управление процессами, доступ к файловой системе, сетевые операции и другие.
Существует множество точек входа для внесения своего хука (все selinux хуки). Но для нашей задачи выбран хук cred_prepare:
Хук cred_prepare вызывается при подготовке структуры учетных данных (credentials) для нового процесса. Это важный этап, потому что учетные данные включают в себя информацию о правах доступа и идентификацию процесса (например, идентификаторы пользователя и группы).
Благодаря этому хуку, мы можем отловить процесс исполнения интересующего системного вызова. Он может быть не только sys_fchmod, но и, практически, любым другим (не для всех системных вызовов нужна структура учетных данных).

## 4 Работа программы

Теперь мы перейдем к описанию работы программы. Процесс может быть логически разделен на следующие ключевые компоненты:

1. **Написание хука:** Это начальный этап, включающий разработку функции хука, которая будет интегрироваться в ядро для выполнения определенных задач безопасности.

2. **Генерация низкоуровневых инструкций:** На этом этапе происходит трансляция кода хука в низкоуровневые инструкции, совместимые с внутренней архитектурой ядра.

3. **Модификация сдвигов памяти для обращений к данным ядра:** Этот шаг включает корректировку смещений памяти при обращениях к данным ядра. В данном контексте следует ознакомиться с технологией Co-Re (Compile Once - Run Everywhere), реализованной в библиотеке libbpf, которая позволяет динамически адаптировать программы eBPF к разным версиям ядра.

4. **Замена/регистрация нового хука:** После генерации и корректировки кода хука, производится его замена или регистрация в системе, что позволяет новому хуку получать данные ядра изнутри.

5. **Безопасное отключение хука:** Обеспечение безопасного отключения хука, что включает его корректное удаление из системы и освобождение всех связанных ресурсов. Этот шаг важен для поддержания стабильности и безопасности системы.

Пункт 1 остается за нами. Пункты с 2 по 4 выполняются при помощи встроенных возможностей библиотеки libbpf, которая позволяет уменьшить объем работы в десятки раз. Пункт 5 является не столько задачей сделать что-то, сколько задачей не делать ничего: у bpf программ есть особенность внесения и изъятия - для их однократной отработки достаточно внести их в ядро с определенным триггером, но для постоянной нужны триггер (в нашем случае - это вызов хука) и дескриптор (это не совсем так, но для простоты объяснения оставлю) файла, в котором хранятся инструкции. В нашем случае, файлом являются данные работающей программы, что при её завершении автоматически удалятся, что приведет к изъятию bpf инструкций из ядра.

### 4.1 Работа программы. Main.

```
#include <bpf/libbpf.h>
#include <unistd.h>
#include "security.skel.h"

int main(int argc, char *argv[])
{
    struct security_bpf *skel;
    int err;
    skel = security_bpf__open_and_load();
    if (!skel) {
        fprintf(stderr, "Ошибка генерации BPF каркаса.\n");
        goto cleanup;
    }
    err = security_bpf__attach(skel);
    if (err) {
        fprintf(stderr, "Ошибка внедрения BPF инструкций.\n");
        goto cleanup;
    }
    printf("BPF программа успешно загружена.\n");
    for (;;) {
        fprintf(stderr, ".");
        sleep(1);
    }
    cleanup:
    security_bpf__destroy(skel);
    return err;
}
```

Структура security.c файла достаточно проста. Она использует возможности libbpf (в частности, утилиту bpftool) для генерации всего необходимого, в том числе и байт-инструкций. После чего этим же инструментом вносит программу в ядро и входит в бесконечный цикл.

### 4.2 Работа программы. BPF.

В случае с кодом хука всё интереснее:

```
#include <linux/bpf.h>
#include <linux/capability.h>
#include <linux/errno.h>
#include <linux/sched.h>
#include <linux/types.h>
#include<linux/kernel.h>
#include <bpf/bpf_tracing.h>
#include <bpf/bpf_helpers.h>
#include <bpf/bpf_core_read.h>

#define X86_64_UNSHARE_SYSCALL 272
#define UNSHARE_SYSCALL X86_64_UNSHARE_SYSCALL

typedef unsigned int gfp_t;

struct pt_regs {
    long unsigned int di;
    long unsigned int orig_ax;
} __attribute__((preserve_access_index));

typedef struct kernflagsel_cap_struct {
    __u32 cap[_LINUX_CAPABILITY_U32S_3];
} __attribute__((preserve_access_index)) kernel_cap_t;

struct cred {
    kernel_cap_t cap_permitted;
} __attribute__((preserve_access_index));

struct task_struct {
    unsigned int flags;
    const struct cred *cred;
} __attribute__((preserve_access_index));

char LICENSE[] SEC("license") = "GPL";
SEC("lsm/cred_prepare")
int BPF_PROG(handle_cred_prepare, struct cred *new, const struct cred *old, gfp
```

```
{
    if (ret) return ret;

    struct pt_regs *regs;
    struct task_struct *task;
    int syscall;
    unsigned long flags;

    task = bpf_get_current_task_btf();
    regs = (struct pt_regs *) bpf_task_pt_regs(task);
    syscall = regs -> orig_ax;

    if (syscall != UNSHARE_SYSCALL) return 0;

    flags = PT_REGS_PARM1_CORE(regs);
    if (!(flags & CLONE_NEWUSER)) {
        return 0;
    }

    return -EPERM;
}
```

Вдаваться в подробности каждой структуры не стану - это очень глубокая лужа, которая со стороны не кажется опасной. интереснее здесь __attribute__((preserve_access_index)) - макрос для компилятора и для дальнейшей релокации данных, который как раз магически определяет неоднозначное определение полей. Это относится к Co-Re технологии. В самом начале хука мы возвращаем ret - возвращенное значение предыдущего такого же хука. Это необходимо из-за того, что при внесении нашего хука, он добавляется в очередь на вызов хуков. То есть он может быть не единственным, из-за чего, для сохранения результата предыдущих хуков, все должны передавать результат дальше по списку (и да, на этом месте также можно реализовать вредоносную программу, которая попросту будет нарушать это, что приведет к неработоспособности других хуков безопасности). regs - регистры. Они хранят все внешние параметры для системного вызова. В целом, остальное всё понятно.