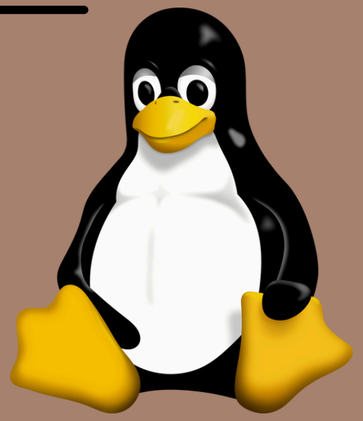


Group - 4

Project Overview

Link to GitHub Repo

Project Topic

struct nsproxy is a kernel data structure that stores namespace pointers for each process

Blocking system calls for processes in specific namespaces

Syscalls require switching to the kernel space. We will go through how exactly syscalls are executed and make modifications in the code factoring in the namespace IDs

Team Members

Aarav Giri

Aditya Palwe

Drone Dangwal

Mihir Vaishampayan

Neelavo Sen

Prince Garg

Riddhi Agarwal

Shreyas Bargale

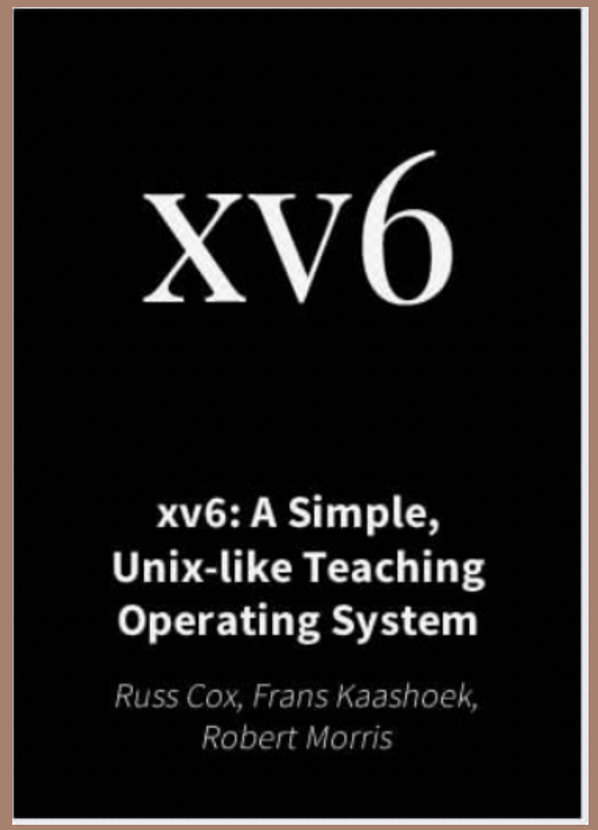
Vaishnavi Rajesh

Vedang Bahuguna

Vikash K Ojha

xv6 Public

After having a breakpoint at syscall, we backtraced to trace the switch manually from user mode to kernel mode, noting all the functions called and their respective files.




```
(gdb) target remote localhost:26000
Remote debugging using localhost:26000
0x0000fff0 in ?? ()
(gdb) break syscall
Breakpoint 1 at 0x80104a60: file syscall.c, line 135.
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:135
135         struct proc *curproc = myproc();
(gdb) backtrace
#0  syscall () at syscall.c:135
#1  0x80105aad in trap (tf=0x8dffffb4) at trap.c:43
#2  0x8010584f in alltraps () at trapasm.S:20
(gdb)
```

xv6 Public

Flow Overview:

- Application calls a system function, which triggers a trap, invoking the alltraps handler in trapasm.S.
- Trap Handling in Assembly (trapasm.S): Assembly code saves current register state, sets up segment registers. The trap frame structure is built on the stack by the hardware and trapasm.S and passed to trap().
- Switches to kernel mode, after pushing stack pointer (trap frame) . Then calls trap in trap.c. For example, the trap frame in xv6 for RISC architecture is a structure containing the kernel_satp, kernel_sp,kernel_trap,epc,kernel_hartid and 31 register values.
- Updates process's trap frame if needed, then calls syscall() in syscall.c.
- The syscall() function checks that the syscall was valid and calls the corresponding syscall function.

```
37 trap(struct trapframe *tf)
38 {
39   if(tf->trapno == T_SYSCALL){
40     if(myproc()->killed)
41       exit();
42     myproc()->tf = tf;
43     syscall();
44     if(myproc()->killed)
45       exit();
46     return;
47   }
```

*syscall.c

```
1 #include "mmu.h"
2
3   # vectors.S sends all traps here.
4 .globl alltraps
5 alltraps:
6   # Build trap frame.
7   pushl %ds
8   pushl %es
9   pushl %fs
10  pushl %gs
11  pushal
12
13  # Set up data segments.
14  movw $(SEG_KDATA<<3), %ax
15  movw %ax, %ds
16  movw %ax, %es
```

```
131 void
132 syscall(void)
133 {
134   int num;
135   struct proc *curproc = myproc();
136
137   num = curproc->tf->eax;
138   if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
139     curproc->tf->eax = syscalls[num]();
140   } else {
141     cprintf("%d %s: unknown sys call %d\n",
142             curproc->pid, curproc->name, num);
```

Tracing syscalls in Linux kernel + finding namespace

```
root@vikash-Dell:~# cat /proc/kallsyms | grep mkdir
ffffffff9c758360 T __pfx___traceiter_cgroup_mkdir
ffffffff9c758370 T __traceiter_cgroup_mkdir
ffffffff9c7583c0 T __pfx___probestub_cgroup_mkdir
ffffffff9c7583d0 T __probestub_cgroup_mkdir
ffffffff9c761920 T __pfx_cgroup_mkdir
ffffffff9c761930 T cgroup_mkdir
ffffffff9c7965b0 t __pfx_instance_mkdir
ffffffff9c7965c0 t instance_mkdir
ffffffff9c804750 t __pfx_shmem_mkdir
ffffffff9c804760 t shmem_mkdir
ffffffff9c8972f0 T __pfx_vfs_mkdir
ffffffff9c897300 T vfs_mkdir
ffffffff9c89c2d0 T __pfx_do_mkdirat
ffffffff9c89c2e0 T do_mkdirat
ffffffff9c89c410 T __pfx___x64_sys_mkdirat
ffffffff9c89c420 T __x64_sys_mkdirat
ffffffff9c89c490 T __pfx___ia32_sys_mkdirat
ffffffff9c89c4a0 T __ia32_sys_mkdirat
ffffffff9c89c4e0 T __pfx___x64_sys_mkdir
ffffffff9c89c4f0 T __x64_sys_mkdir
```

/proc/kallsyms is a file listing all exported kernel symbols along with their memory addresses

```
vaishnavirajesh@DESKTOP-7C2QIFI:~$ sudo perf probe -a '__x64_sys_mkdir'
Added new event:
  probe:__x64_sys_mkdir (on __x64_sys_mkdir)

You can now use it in all perf tools, such as:

        perf record -e probe:__x64_sys_mkdir -aR sleep 1

vaishnavirajesh@DESKTOP-7C2QIFI:~$ sudo perf record -e probe:__x64_sys_mkdir -g -- mkdir test_dir
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.032 MB perf.data (1 samples) ]
vaishnavirajesh@DESKTOP-7C2QIFI:~$ sudo perf report
vaishnavirajesh@DESKTOP-7C2QIFI:~$ sudo perf script
mkdir   10703 [001]   240.231207: probe:__x64_sys_mkdir: (ffffffff9c48c6f4)
        ffffffff9c48c6f5 __x64_sys_mkdir+0x5 ([kernel.kallsyms])
        ffffffff9cfff461 do_syscall_64+0x61 ([kernel.kallsyms])
        ffffffff9d20012a entry_SYSCALL_64_after_hwframe+0x6e ([kernel.kallsyms])
            7fb90dfad99b __GI___mkdir+0xb (/usr/lib64/libc.so.6)
            55fd5ba7ade5 [unknown] (/usr/bin/mkdir)
            55fd5ba70257 [unknown] (/usr/bin/mkdir)
            7fb90dece14a __libc_start_call_main+0x7a (/usr/lib64/libc.so.6)
            7fb90dece20b __libc_start_main@@GLIBC_2.34+0x8b (/usr/lib64/libc.so.6)
            55fd5ba70875 [unknown] (/usr/bin/mkdir)

(END)
```

do_syscall_64

include / linux / ns_common.h

```c
/* SPDX-License-Identifier: GPL-2.0 */
#ifndef _LINUX_NS_COMMON_H
#define _LINUX_NS_COMMON_H

#include <linux/refcount.h>

struct proc_ns_operations;

struct ns_common {
        struct dentry *stashed;
        const struct proc_ns_operations *ops;
        unsigned int inum;
        refcount_t count;
};

#endif
```

```c
static inline void prepare_task(struct task_struct *next)
{
#ifdef CONFIG_SMP
        /*
         * Claim the task as running, we do this before switching to it
         * such that any running task will have this set.
         *
         * See the smp_load_acquire(&p->on_cpu) case in ttwu() and
         * its ordering comment.
         */
        WRITE_ONCE(next->on_cpu, 1);
#endif
}
```

sched.h , nsproxy.h , ns_common.h

Files backtracked

“current->nsproxy->{cgroup_ns, ipc_ns, mnt_ns, net_ns, uts_ns}->ns.inum”

```c
 */
static __always_inline struct rq *
context_switch(struct rq *rq, struct task_struct *prev,
               struct task_struct *next, struct rq_flags *rf)
{
        prepare_task_switch(rq, prev, next);

        /*
         * For paravirt, this is coupled with an exit in switch_to to
         * combine the page table reload and the switch backend into
         * one hypercall.
         */
        arch_start_context_switch(prev);

        /*
         * kernel -> kernel   lazy + transfer active
         *   user -> kernel   lazy + mmgrab_lazy_tlb() active
         *
         * kernel ->   user   switch + mmdrop_lazy_tlb() active
         *   user ->   user   switch
         *
         * switch_mm_cid() needs to be updated if the barriers provided
         * by context_switch() are modified.
         */
        if (!next->mm) {                                // to kernel
                enter_lazy_tlb(prev->active_mm, next);

                next->active_mm = prev->active_mm;
```

include / linux / nsproxy.h

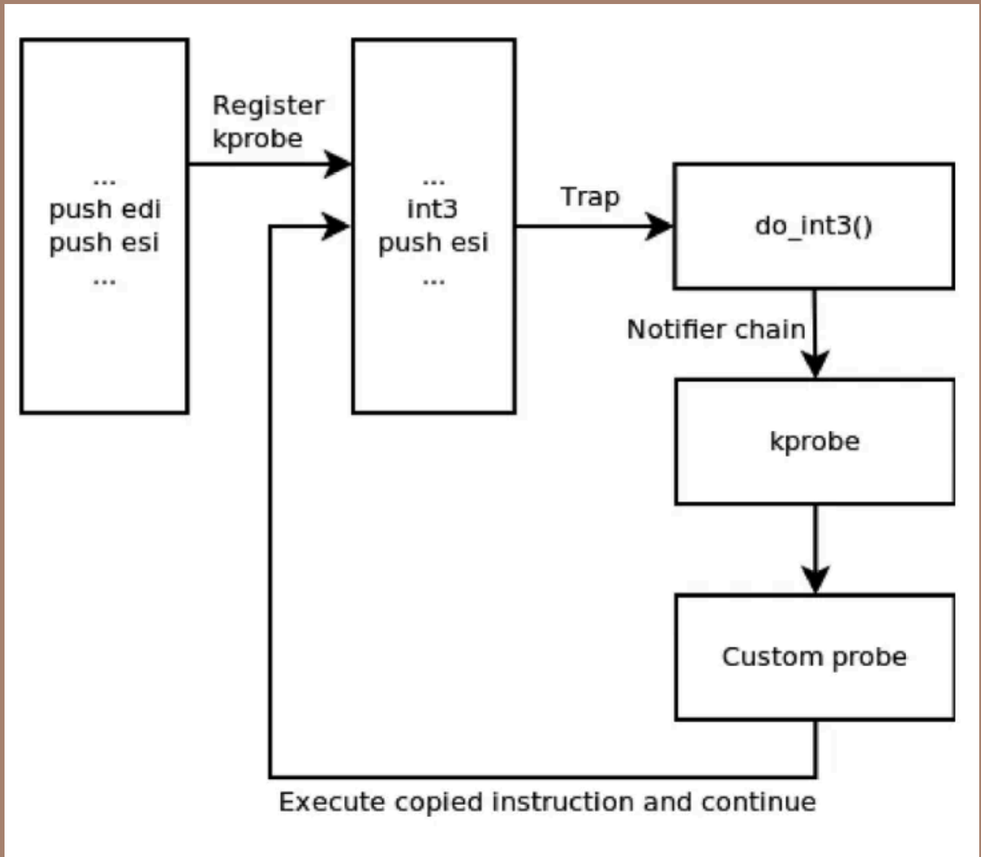
```c
struct nsproxy {
        refcount_t count;
        struct uts_namespace *uts_ns;
        struct ipc_namespace *ipc_ns;
        struct mnt_namespace *mnt_ns;
        struct pid_namespace *pid_ns_for_children;
        struct net           *net_ns;
        struct time_namespace *time_ns;
        struct time_namespace *time_ns_for_children;
        struct cgroup_namespace *cgroup_ns;
};
extern struct nsproxy init_nsproxy;
```

KProbes

- Dynamically monitoring and tracing almost any kernel instruction
- Precise debugging without needing kernel recompilation, test kernel-level changes or policies without altering the source code
- Insert probes at runtime, without disrupting normal kernel operations
- Define pre- and post-handlers to examine registers, variables, and memory states at specific execution points
- Influence kernel execution paths, such as redirecting or blocking specific system calls

For the purpose of our project, we can insert a check for the required namespaces while intercepting system calls, allowing us to block or modify their execution dynamically based on the namespace context

KProbe Control-Flow



- When a kprobe is registered by the kernel, the address to probe is replaced by an interrupt instruction, and the original instruction is saved.
- On interrupt the kprobe is called and the pre-handler is executed then the saved instruction is executed and after that, the post-handler is invoked (if the post-handler was defined).
- Thereafter the control goes back to the original instruction set and the execution continues.

```c
static int handler_pre(struct kprobe *p, struct pt_regs *regs) {
    printk(KERN_INFO "Kprobe handler called\n");
    printk(KERN_INFO "Syscall happened for SYSCALL\n");
    printk(KERN_INFO "The current process PID: %d\n", current->pid);
    printk(KERN_INFO "The current process name: %s\n", current->comm);
    printk(KERN_INFO "The current instruction pointer: %lx\n", regs->ip);
    printk(KERN_INFO "The currnt code segment: %lx\n", regs->cs);
    int TARGET_PID_NAMESPACE[] = {12345};
    // TARGET_PID_NAMESPACE is an array of inum of the PID namespaces for which the "SYSCALL" syscall is to be blocked
    for (int i=0; i<PID_QSIZE; ++i) {
        // Check if the current process is in the target PID namespace
        // current is a macro that points to the task_struct of the current process --> #define current (get_current())
        // task_active_pid_ns(current)->ns.inum returns the inum of the PID namespace of the current process
        /*************************************************************************************************************
         | In the current implementation, pid namespace is used to block the syscall.                                |
         | To use other namespaces, the following can be used:                                                       |
         | current->nsproxy->{cgroup_ns, ipc_ns, mnt_ns, net_ns, uts_ns}->ns.inum can be compared with the target namespace inum |
         *************************************************************************************************************/
        // If the current process is in the target PID namespace, then block the "SYSCALL" syscall
        if(task_active_pid_ns(current)->ns.inum == TARGET_PID_NAMESPACE[i]) {
            //if root then don't block
            if(current_uid().val == 0) {
                printk(KERN_INFO "Root user detected, not blocking SYSCALL syscall\n");
                return 0;
            }
            printk(KERN_INFO "Blocked SYSCALL syscall for task in PID namespace: %u\n", task_active_pid_ns(current)->ns.inum);
            /*************************************************************************************************************
             * Below we edit the registers values to block the syscall.                                                  *
             * All the registers are set to 0.                                                                           *
             * So if any register is carrying a pointer to some memory location, then in dereferencing the pointer, a kernel bug will be reported. *
             * The kernel bug will be reported because the pointer is pointing to the memory location 0, which is not a valid memory location *
             * and the process will get eventually killed.                                                               *
             *************************************************************************************************************/
            regs->ax = 0;
            regs->bx = 0;
            regs->cx = 0;
            regs->dx = 0;
            regs->si = 0;
            regs->di = 0;
        }
    }
    return 0;
}
```

Codebase

This is the pre-handler of our version 2.0 code in which we are nulling out register values, so if there is a pointer value in the register, it will get nullified. So when the syscall, which takes a pointer as one of its arguments, happens in a namespace in which it was supposed to be blocked, its arguments which are present in the register will get nullified. So when the syscall will try to dereference a null pointer the kernel will report a kernel bug and the process doing the syscall will get killed.

Codebase

```c
static int __init kprobe_init(void) {
    int ret;
    ret = register_kprobe(&kp); // registering kprobe module
    if (ret < 0) { // checking if kprobe is registered successfully
        printk(KERN_ERR "Failed to register kprobe: %d\n", ret);
        return ret;
    }
    kp.pre_handler = handler_pre; // setting pre_handler for kprobe
    // pre_handler -> handler function to be called before the function is executed
    printk(KERN_INFO "Kprobe registered for __x64_sys_SYSCALL\n");
    return 0;
}

static void __exit kprobe_exit(void) {
    unregister_kprobe(&kp); // unregistering kprobe module
    printk(KERN_INFO "Kprobe unregistered for __x64_sys_SYSCALL\n");
}

module_init(kprobe_init);
// modult_init -> macro to define the function to be called when the module is loaded
module_exit(kprobe_exit);
// module_exit -> macro to define the function to be called when the module is unloaded
```

Registering Kprobe and assigning it s pre-handler

```c
/* PID_QSIZE is the size of the array of inum
of the PID namespaces for which the "SYSCALL" syscall is to be blocked
the script will edit the value of PID_QSIZE to the number of entries
in the TARGET_PID_NAMESPACE array */
#define PID_QSIZE 1

/* kp is a kprobe structure that will resolve the address of the
 __x64_sys_SYSCALL function and put a probe
just before the function is executed */
static struct kprobe kp = {
    // symbol_name -> name of the symbol to be resolved
    // Note: SYSCALL is just a placeholder
    // the script will replace it with the actual syscall name
    .symbol_name = "__x64_sys_SYSCALL",
};
```

Kprobe structure where the symbol name represents the symbol address at which the probe is to be inserted

Blocking syscall

Implementation1: kernel panic

```
            // task_active_pid_ns(current) returns task_struct of the current process which contains info about the PID namespace o
            if(task_active_pid_ns(current)->ns.inum == TARGET_PID_NAMESPACE[i]) {
                printk(KERN_INFO "Blocked SYSCALL syscall for task in PID namespace: %u\n", task_active_pid_ns(current)->ns.inum);
                // panic -> function to print the message and stop the execution of the kernel
                panic("Blocked SYSCALL syscall for task in PID namespace: %u\n", task_active_pid_ns(current)->ns.inum);
            }
        }
        return 0;
    }
```

Raising kernel panic

```
        // task_active_pid_ns(current) returns task_struct of the current process which contains info about the PID namespace of the c
        if(task_active_pid_ns(current)->ns.inum == TARGET_PID_NAMESPACE[i]) {

            //if root then conitnue
            if(current_uid().val == 0) {
                printk(KERN_INFO "Root user detected, not blocking SYSCALL syscall\n");
                return 0;
            }

            printk(KERN_INFO "Blocked SYSCALL syscall for task in PID namespace: %u\n", task_active_pid_ns(current)->ns.inum);
            regs->ax = 0;
            regs->bx = 0;
            regs->cx = 0;
            regs->dx = 0;
            regs->si = 0;
            regs->di = 0;

        }
    }
    return 0;
}
```

Allowing root user to execute the syscalls

Edit the registers values to block the syscall.

Implementation2: graceful termination, process killed

unlinkat

finding pid namespace id

```
vaishnavi@vaishnavi-VirtualBox:~/Downloads/code2/project$ ll /proc/$$/ns | grep 'pid '
lrwxrwxrwx 1 vaishnavi vaishnavi 0 Nov 28 19:21 pid -> pid:[4026531836]
vaishnavi@vaishnavi-VirtualBox:~/Downloads/code2/project$ cat inp_list.txt
unlinkat, 4026531836
```

rm syscall uses unlinkat syscall internally

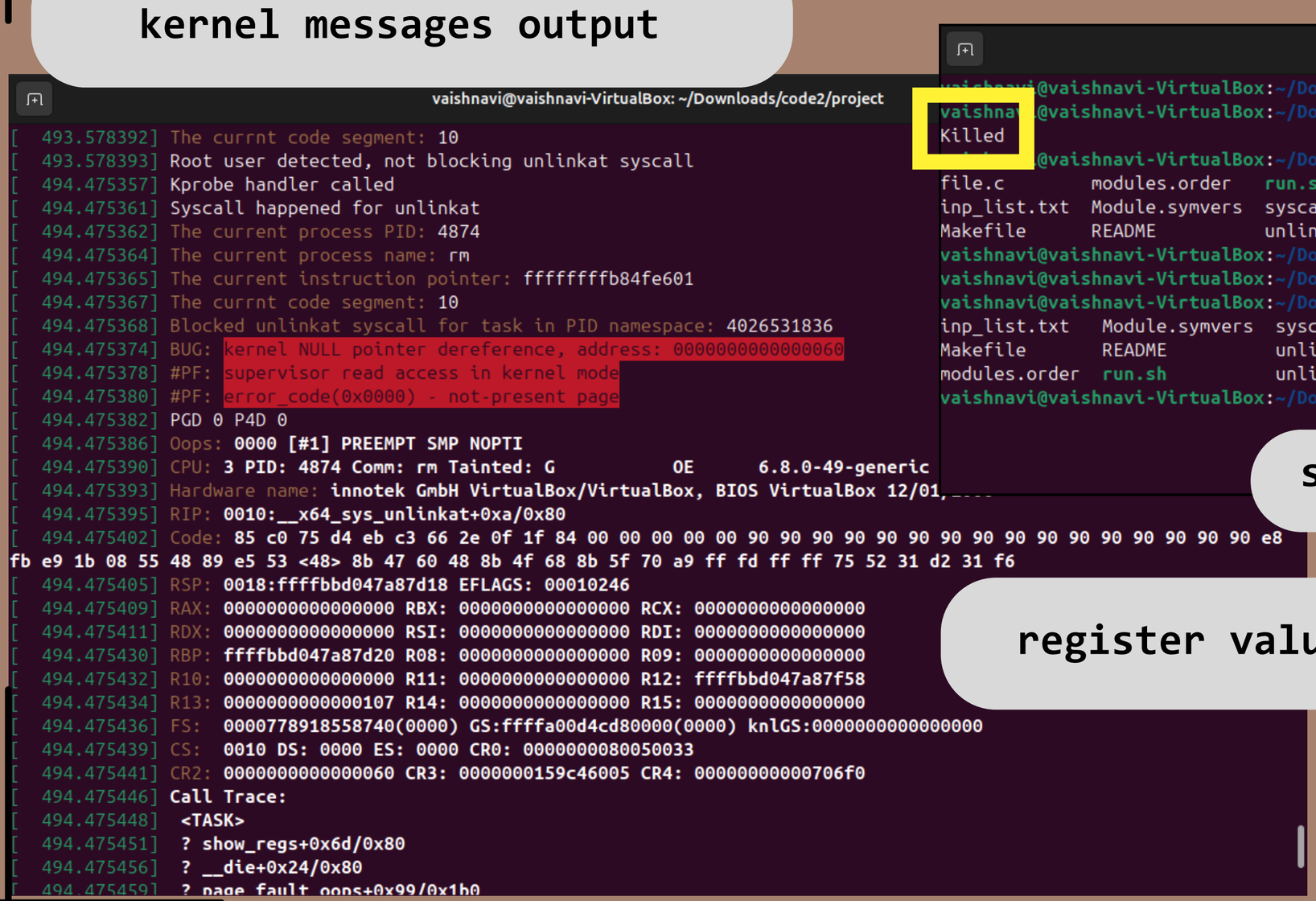
```
vaishnavi@vaishnavi-VirtualBox:~/Downloads/code2/project$ ./run.sh
Generated unlinkat_trap_v2.c
inp_list.txt is not empty. Overwriting obj-m line in Makefile with dependencies...
Makefile updated with dependencies: unlinkat_trap_v2.o
Files written. Makefile updated.
Would you like to compile the files? (enter 'y' if yes):
y
make -C /lib/modules/6.8.0-49-generic/build M=/home/vaishnavi/Downloads/code2/project modules
make[1]: Entering directory '/usr/src/linux-headers-6.8.0-49-generic'
warning: the compiler differs from the one used to build the kernel
  The kernel was built by: x86_64-linux-gnu-gcc-13 (Ubuntu 13.2.0-23ubuntu4) 13.2.0
  You are using:           gcc-13 (Ubuntu 13.2.0-23ubuntu4) 13.2.0
  CC [M]  /home/vaishnavi/Downloads/code2/project/unlinkat_trap_v2.o
  MODPOST /home/vaishnavi/Downloads/code2/project/Module.symvers
  CC [M]  /home/vaishnavi/Downloads/code2/project/unlinkat_trap_v2.mod.o
  LD [M]  /home/vaishnavi/Downloads/code2/project/unlinkat_trap_v2.ko
  BTF [M] /home/vaishnavi/Downloads/code2/project/unlinkat_trap_v2.ko
Skipping BTF generation for /home/vaishnavi/Downloads/code2/proje
make[1]: Leaving directory '/usr/src/linux-headers-6.8.0-49-gene
Compilation occurred successfully.
Would you like to insert the modules now? (enter 'y' if yes):
y
Inserting module unlinkat_trap_v2.ko...
Successfully inserted module unlinkat_trap_v2.ko
Successfully processed all the modules.
vaishnavi@vaishnavi-VirtualBox:~/Downloads/code2/project$
```

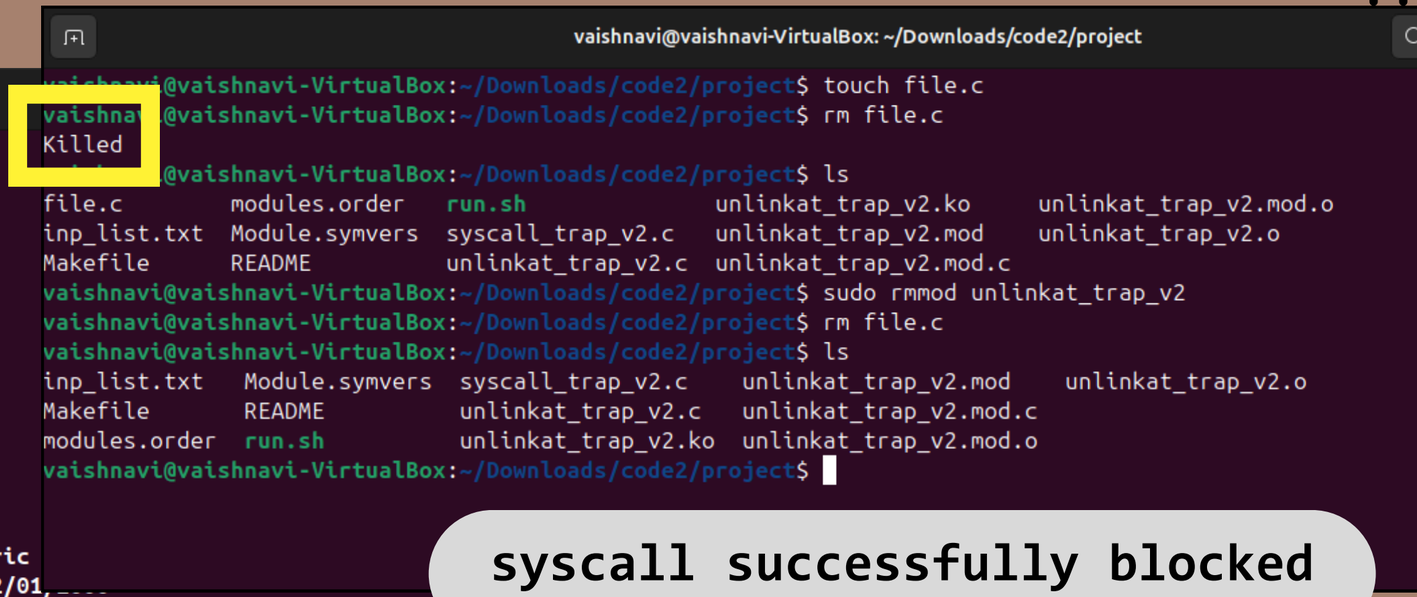
```
vaishnavi@vaishnavi-VirtualBox:~/Downloads/code2/project$ strace rm file.c 2>&1 | tail -n 8
(AT_FDCWD, "file.c", W_OK, AT_EACCESS) = 0
unlinkat(T_FDCWD, "file.c", 0)          = 0
lseek(0, , SEEK_CUR)                    = -1 ESPIPE (Illegal seek)
close(0)                                = 0
close(1)                                = 0
close(2)                                = 0
exit_group(0)                           = ?
+++ exited with 0 +++
vaishnavi@vaishnavi-VirtualBox:~/Downloads/code2/project$
```

namespace found,
kernel module inserted

unlinkat

kernel messages output



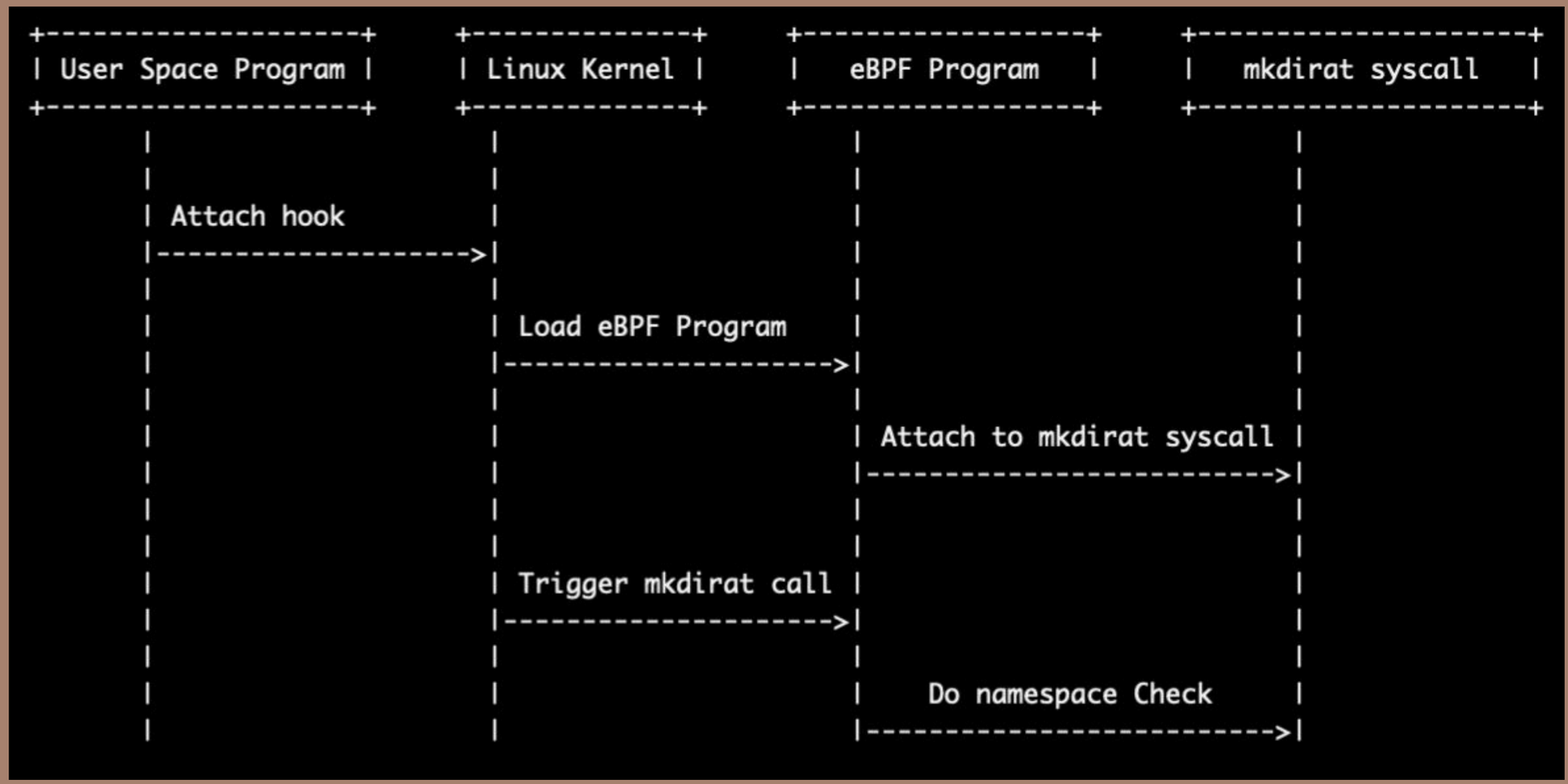


syscall successfully blocked

register values

EBPF

Used eBPF to intercept a system call without modifying the kernel code. Below is a control flow diagram of the same.



eBPF-Implementation

We have mainly written two programs:

1. A user-space program (written in Python using BCC) that communicates with the kernel.
2. The kernel-space eBPF program (written in C) that executes within the eBPF virtual machine, modifying the kernel's behavior.

Running ***strace mkdir test_dir*** revealed that the relevant syscall was ***mkdirat***. After which we attach kprobe to the function call to intercept to do checks on namespace.

```python
from bcc import BPF
import ctypes as ct
import os
# Import required libraries

# Function to add elements in the share
def add_target_ns(map, ns):
    key = map.Key()
    key.inode_num = ct.c_ulong(ns)
    value = ct.c_int(1)
    map[key] = value

def main():
    with open("ebpf_program.c") as f:
        bpf_prog = f.read()

    b = BPF(text=bpf_prog)

    #get the prefix of the system specific function name and add openat to it
    fnname_openat = b.get_syscall_prefix().decode() + 'mkdirat'
    b.attach_kprobe(event=fnname_openat, fn_name="syscall__mkdirat")
    target_ns = b.get_table("target_ns")

    # restricting for current namespace for demo
    devinfo = os.stat("/proc/self/ns/mnt")
    restricted_ns = [devinfo.st_ino]
    for ns in restricted_ns:
        add_target_ns(target_ns, ns)

    # Just Checking if the hash map is fine or not.
    for x in target_ns:
        print("Blocking for namespace inode:",x.inode_num)

    try:
        print("Attaching kprobe to syscall mkdirat... Press Ctrl+C to exit.")
        b.trace_print() # read the output of bpf_trace_printk unless interrupted.
    except KeyboardInterrupt:
        pass


if __name__ == "__main__":
    main()
```

```c
// mnt_namespace struct was said to be not
struct mnt_namespace {
    struct ns_common ns;
};


struct key_t { // Uniquely identifies a nar
    u64 inode_num;
};

BPF_HASH(target_ns, struct key_t, int); //
```

```c
// kprobe on mkdirat, hook onto the running kernel code
// parameters are :- context, directory file descriptor, filename pointer, associated flags
// found using strace mkdir test_dir
int syscall__mkdirat(struct pt_regs *ctx, int dfd, const char __user *filename, int flags) {
    struct key_t curr_ns = {};

    // get the inode of namespace file of the caller task. inode is:-
    // current_task->nsproxy->mnt_ns->ns.inum, but do it safely:-

    struct task_struct *current_task;
    struct nsproxy *nsproxy;
    struct mnt_namespace *mnt_ns;
    unsigned int inum;
    u64 ns_id;

    current_task = (struct task_struct *)bpf_get_current_task();

    // copies size bytes from kernel address space to the BPF stack
    // For safety kernel memory reads are done through this function
    if (bpf_probe_read_kernel(&nsproxy, sizeof(nsproxy), &current_task->nsproxy))
    return 0;

    if (bpf_probe_read_kernel(&mnt_ns, sizeof(mnt_ns), &nsproxy->mnt_ns))
    return 0;

    if (bpf_probe_read_kernel(&inum, sizeof(inum), &mnt_ns->ns.inum))
        return 0;

    ns_id =  (u64) inum;
    curr_ns.inode_num = ns_id;

    // Check if the current namespace is in the hash map shared
    int *fnd = target_ns.lookup(&curr_ns);
    bpf_trace_printk("mkdirat called by process with namespace inode: %llu\n",curr_ns.inode_num); // log

    if (fnd) {
        // Block mkdirat syscall if namespace is found
        char fname[NAME_MAX];
        // safely copy a NULL terminated string from user address space to the BPF stack
        bpf_probe_read_user_str(fname, sizeof(fname), (void *)filename);
        bpf_trace_printk("mkdirat command blocked while creating directory %s!\n", fname);  // log
        bpf_override_return(ctx, -EACCES); // Block, give -EACCES return code :- Permission Denied
    }

    return 0; // allow the syscall
}
```

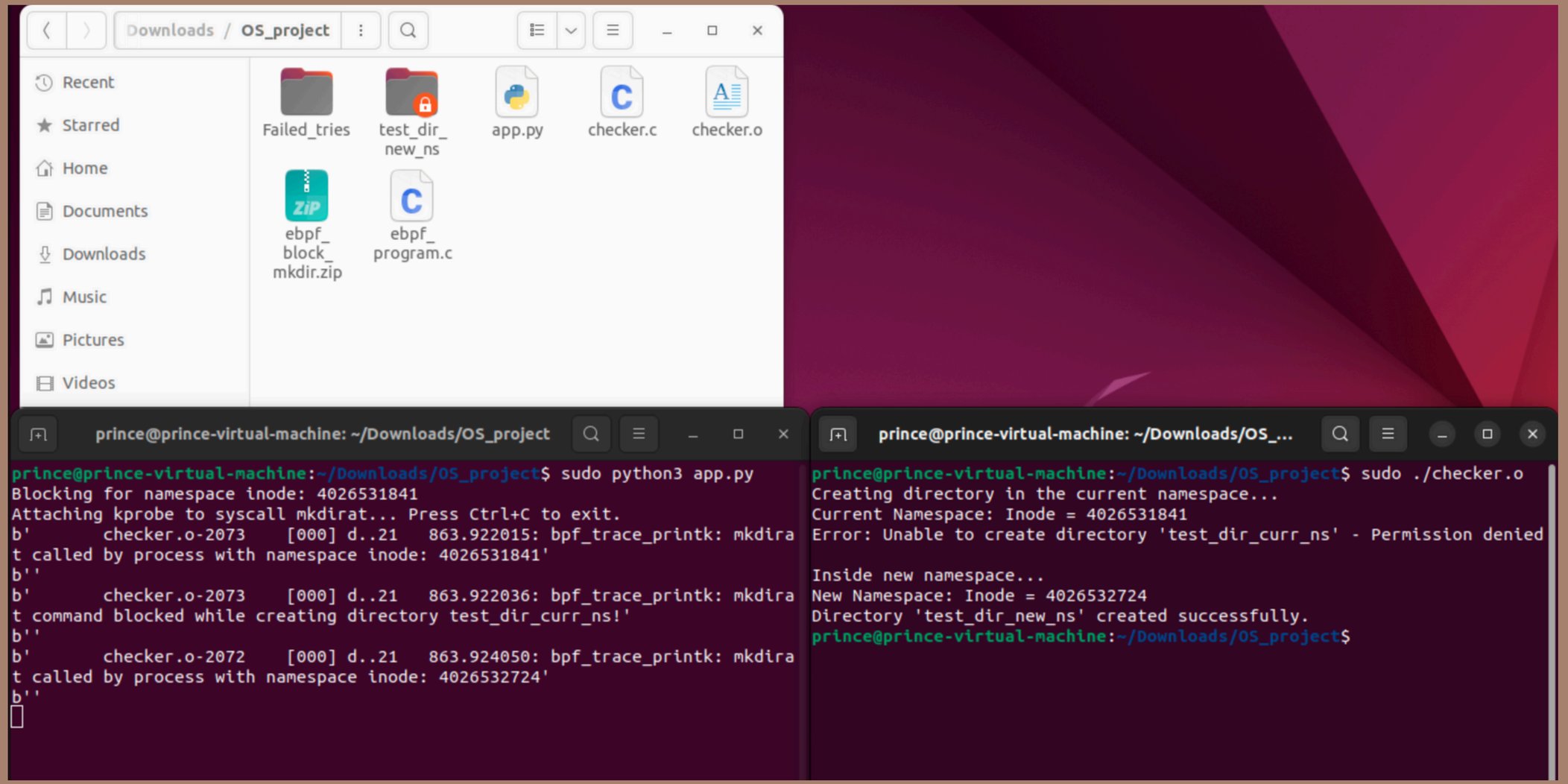
eBPF-Blocking

- We intercept the mkdirat system call and modify its behavior to return an -**EACCES** error (permission denied) when called from certain **namespaces**.
- We have used ***bpf_override_return(struct pt_regs regs, u64 rc)***. It is used for **error injection** and overrides the return value of the probed function
- To determine which namespaces should be blocked, we need a mechanism to **pass information** from user-space to kernel-space.

eBPF-Logic

- Used **eBPF maps** to serve as a conduit for data exchange between user-space and kernel-space.
- User-space can **populate** the map with the namespaces that need to be restricted (e.g., by adding the **inode_num** of the namespaces).
- Kernel-Space (eBPF Program):
  - The eBPF program checks the namespace of the calling process (based on inode_num).
  - If the current namespace is found in the eBPF map, the mkdirat syscall is blocked and returns an -EACCES error (Permission Denied).

OUTPUT


Downloads / OS_project
Recent
Starred
Home
Documents
Downloads
Music
Pictures
Videos
Failed_tries
test_dir_new_ns
app.py
checker.c
checker.o
ebpf_block_mkdir.zip
ebpf_program.c
prince@prince-virtual-machine: ~/Downloads/OS_project
prince@prince-virtual-machine:~/Downloads/OS_project$ sudo python3 app.py
Blocking for namespace inode: 4026531841
Attaching kprobe to syscall mkdirat... Press Ctrl+C to exit.
b' checker.o-2073 [000] d..21 863.922015: bpf_trace_printk: mkdirat called by process with namespace inode: 4026531841'
b''
b' checker.o-2073 [000] d..21 863.922036: bpf_trace_printk: mkdirat command blocked while creating directory test_dir_curr_ns!'
b''
b' checker.o-2072 [000] d..21 863.924050: bpf_trace_printk: mkdirat called by process with namespace inode: 4026532724'
b''
prince@prince-virtual-machine: ~/Downloads/OS_...
prince@prince-virtual-machine:~/Downloads/OS_project$ sudo ./checker.o
Creating directory in the current namespace...
Current Namespace: Inode = 4026531841
Error: Unable to create directory 'test_dir_curr_ns' - Permission denied
Inside new namespace...
New Namespace: Inode = 4026532724
Directory 'test_dir_new_ns' created successfully.
prince@prince-virtual-machine:~/Downloads/OS_project$

LSMs

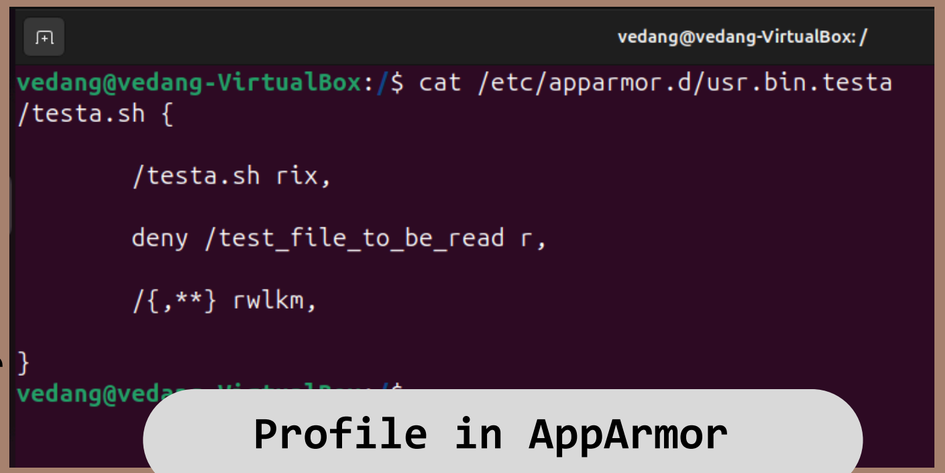
- Linux Security Modules (LSMs) are a framework in the Linux kernel that allows various security policies to be applied to control what a user or program can do on the system.
- Users can add extra security layers over traditional Unix permissions
- LSMs allow developers to write custom security modules and enable systems to implement advanced security models.

Some security models are

- **SELinux**
- **AppArmor**

AppArmor

- AppArmor (Application Armor) uses a profile-based model to restrict the capabilities of applications and processes, confining them to specific resources and operations based on defined rules.
- Profiles: AppArmor operates with profiles, which are sets of rules specifying what files, capabilities, or resources a program can access.



Profile in AppArmor

```
vedang@vedang-VirtualBox:/$ cat testa.sh
#!/usr/bin/bash

cat /test_file_to_be_read
vedang@vedang-VirtualBox:/$ sudo ./testa.sh
./testa.sh: line 3: /usr/bin/cat: Permission denied
vedang@vedang-VirtualBox:/$ cat test_file_to_be_read
Sucessful in reading the file
vedang@vedang-VirtualBox:/$
```

Denial of Access

However, Apparmor cannot block system calls such as chmod and kill as it operates at the path level, and not at the low-level syscalls. System calls are not inherently tied to the file paths
For eg : read()

Custom LSM as a part of kernel code

vedang@vedang-VirtualBox: /

```
vedang@vedang-VirtualBox:/$ sudo cat /linux/security/my_lsm.c
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/lsm_hooks.h>
#include <linux/cred.h>
#include <linux/sched.h>
#include <linux/security.h>

static int my_lsm_task_kill(struct task_struct *task, struct kernel_siginfo *info, int sig,  const struct cred *cred)
{
    pr_info("Task Kill Hook Called for the following %s\n", task->comm);
    return 0;
}

static struct security_hook_list my_lsm_hooks[] = {
    LSM_HOOK_INIT(task_kill, my_lsm_task_kill),
};

static struct lsm_id my_lsm_id = {
    .name = "my_lsm",
};

static int __init my_lsm_init(void)
{
    pr_info("My LSM Initialized\n");
    security_add_hooks(my_lsm_hooks, ARRAY_SIZE(my_lsm_hooks), &my_lsm_id);
    return 0;
}

static void __exit my_lsm_exit(void)
{
    pr_info("My LSM Exited\n");
}

module_init(my_lsm_init);
module_exit(my_lsm_exit);
vedang@vedang-VirtualBox:/$
```

Modify kernel code by placing LSM implementation in security/directory

Registration of the LSM within the kernel is done by calling the security_add_hooks()

Ensure its compilation and linking by updating makefile

An entry must be added to the Kconfig file in the security/ directory. It defines the configuration options for LSM

Work Division

xv6

Neelavo

Kprobe/Syscall Tracing

Vikash
Aarav
Vaishnavi

eBPF

Prince
Shreyas
Riddhi

LSMs

Mihir
Vedang
Drone
Aditya