

Project Report - Group 4

Blocking System Calls for processes in specific namespaces.

In Linux, namespaces are used to isolate processes by grouping resources like filesystems, network interfaces, and process IDs. This isolation is important for running containers and maintaining system boundaries. Sometimes for security reasons we might need to block certain system calls for processes in specific namespaces.

For example, we can restrict dangerous syscalls like `ptrace` or `mount` for certain namespaces to prevent processes from interfering with the host system or other namespaces. Similarly, syscalls that allow unrestricted networking or interprocess communication can also be restricted to avoid privilege escalation or denial-of-service attacks.

Since the Linux kernel is complex, we started with `xv6`, a simpler operating system developed with just the essential features of Linux. Tracing syscalls in `xv6` helped us understand the basic mechanics and draw comparisons to the Linux kernel. From there, we explored three different approaches to implement syscall restrictions based on namespaces:

- Kprobes
- eBPFs
- LSMs

Each of these approaches have been explained in this report, along with their code, testing and observations.

Here's the [link](#) to our GitHub repository containing various codes discussed.

Xv6 Public

xv6 is a minimalist Unix-like operating system designed for educational purposes. It is a reimplementation of the Sixth Edition Unix (V6), that was developed to run on modern hardware or emulators, such as QEMU, which is what we used to run it. Xv6 is much simpler than Unix but provides a fully functional environment with essential Unix features, including system calls, a shell, and user-space programs. xv6 however does not feature namespaces and We looked into possibly implementing namespaces in xv6 ourselves for the purposes of this project but it was infeasible for several reasons. Thus, to understand how namespaces might interact with system call mechanisms in Linux, we first analysed how system calls are implemented and handled in xv6.

To investigate how system calls are handled in the kernel, we set a breakpoint at the syscall entry point in the xv6 kernel. We continued execution to trace the flow until a breakpoint was hit. We then used the backtrace command to examine the stack and track the system call's journey. This led us to the kernel files, which helped us to understand the flow.

The system flow is detailed as follows:

1. System Call Trigger and Trap Handling

- When an application calls a system function, it triggers a trap, which invokes the alltraps handler located in the assembly file trapasm.S.
- The trapasm.S assembly code is responsible for saving the current state of CPU registers and setting up the segment registers. Thus the trap frame is built on the stack by the hardware and the assembly code of trapasm.S and is passed to the trap function in trap.c when it is called.
- After saving the state, the code transitions to kernel mode, pushing the stack pointer (trap frame) onto the stack, and then calls the trap function in trap.c.

```
(gdb) target remote localhost:26000
Remote debugging using localhost:26000
0x0000ffff 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)
```

```
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     }
```

2. System Call Execution Flow

- The trap function in trap.c evaluates the trap number (trapno) in the trap frame pointer that is passed to it.
- If this trap number corresponds to a system call (T_SYSCALL), the process's trap frame (tf) is updated, and the syscall() function in syscall.c is called.
- The syscall() function identifies the system call number from the trap frame and executes the corresponding system call function. If the system call number is invalid, a suitable error message for the same is logged.

```
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);
```

KProbe

1. KProbe Control-Flow

The following is the procedure involved in the control flow of KProbe

- 1. Registering KProbe:** A KProbe has to be first registered with the kernel, at this step the instruction at the address/symbol at which KProbe is to be attached is saved and replaced with an interrupt instruction to invoke KProbe when the instruction pointer reaches that address.
- 2. Calling of pre-handler:** When the instruction pointer reaches the address, which is replaced with an interrupt, the interrupt signals the kernel to invoke the KProbe, and then the KProbe pre-handler is invoked.
- 3. Executing the saved instruction:** After pre-handler execution, the actual instruction that was saved is executed.
- 4. Calling of post-handler (optional):** If the post-handler is defined the post-handler is executed after executing the saved instruction.
- 5. Back to normal flow:** After executing the handlers of the KProbe the instruction pointer returns to normal execution flow.

2. KProbe as a module

Kprobes are written as loadable kernel modules which can be compiled the way in which a general module is compiled and can be inserted via the *insmod* command, which is also the way in which a general kernel module gets inserted into the kernel.

3. Commands and Headers used

<code>#include <linux/kernel.h></code>	Contains core kernel macros, functions,types ,ex:- printk
<code>#include <linux/kprobes.h></code>	Kprobe API for intercepting and handling execution of kernel functions.
<code>#include <linux/module.h></code>	Provides macros for defining kernel modules, metadata.

	ex:- MODULE_LICENSE, module_init, module_exit
#include <linux/nsproxy.h>	Defines structures, functions for managing namespace proxies
#include <linux/panic.h>	Provides functions and macros for triggering kernel panics
#include <linux/pid_namespace.h>	Provides structures and functions for managing PID namespaces
#include <linux/sched.h>	Contains process scheduling structures and task-related functions, like <i>task_struct</i> .
#include <linux/sched/signal.h>	Extends <i>sched.h</i> for signal handling for tasks, like sending, handling process signals.
#include <linux/uaccess.h>	Provides functions for safely accessing user-space memory from kernel space, ex:- copy_to_user, copy_from_user

strace : Traces and displays the system calls and signals used by a process. Here we used it for tracing syscalls to find the functions for inserting kprobe and for commands like rm which use other syscalls in their implementation.

insmod : Inserts a kernel module into the running Linux kernel

cat /proc/kallsyms : /proc/kallsyms is a file listing all exported kernel symbols along with their memory addresses. We used it to help find certain functions for tracing syscalls like __x64_sys_ prefixed functions which are to be intercepted.

ls -l /proc/\$\$/ns : Lists the namespace information of the current shell process. From this, we find the current pid namespace for blocking syscalls

4. How we got probing points/symbols :

We looked at the kernel source code from [this](#) website. For a few syscalls we traced the control flow in kernel and user space using tools like perf, strace, etc. We also searched through the list of kernel symbols using cat /proc/kallsyms command. The aim was to initially find the file where

context switch to kernel mode happens and a common file that all syscalls go through in their control flow.

After finding the files we went through the functions which are being called in the control flow and tried to understand their implementation. From there we found the `__x64_sys_` prefixed symbols which we are using in the kprobe module.

5. Our Logic

One thing with KProbe is that it was not meant for blocking a system call, but there are a few methods by which we can enforce this blocking of syscall for specific namespace(s).

Method 1 (destructive in nature)[version 1.0]:

Whenever there is a system call from a namespace in which that syscall is prohibited, in the KProbe, just before entering the syscall function KProbe's pre-handler calls the panic function to panic the kernel. Thus effectively blocking the syscall for that particular namespace. After kernel panic, the system has to be restarted.

Method 2 (graceful termination)[version 2.0]:

Syscalls like `mkdir`, `open`, and `unlink` expect a pointer as one of its arguments that is passed in one of the general-purpose registers i.e. `RAX`, `RBX`, `RCX`, `RDX`, `RSI`, and `RDI`. **KProbe gives the power to change the register's value.** In this method, we will null out every value in the register, so if there is a pointer in the register it will get nullified. So when the syscall happens which is to be blocked in a namespace, KProbe will be called and its pre-handler would check the namespace of the process going to make syscall to see if the namespace making the syscall is allowed to do that syscall or not, if it is allowed then the syscall would happen else the pre-handler would nullify all the general purpose registers. If there is a pointer in the register it will get nullified, so when the syscall tries to dereference the pointer, it will report a kernel bug because of the NULL pointer dereferencing and the process will get killed, thus stopping the syscall from happening. This method applies to all the syscalls that take a pointer as one of its arguments like `read`, `write`, `open`, `unlink`, `execve`, `clone`, `mount`, `umount`, etc.

6. Code implementation

- **About `syscall_trap_v2.c`:** This is the kprobe module that we compile with the help of a Makefile and a script ("`run.sh`") that runs this Makefile. A macro called `PID_QSIZE` is defined to be the size of the array (called `TARGET_PID_NAMESPACE`) which is by default 1. `TARGET_PID_NAMESPACE` contains an array of namespaces to block for the

SYSCALL system call that we are blocking. By default, it contains {12345}.

This code iterates over each namespace in TARGET_PID_NAMESPACE and checks if the namespace of the task struct of the current process is found in this array or not. If yes, then this system call is blocked (based on the type of exit: either panic destructively or graceful exit).

“current” is a macro for “get_current()” that returns a pointer that points to the “task_struct” of the current process. “task_active_pid_ns(current)” returns the PID namespace of the current process, and “task_active_pid_ns(current)->ns.inum” is the required field to check the PID namespace from. To not block root syscalls, we put a check using “current_uid().val == 0”, 0 being the root user; current_uid gives info about which user profile is present on the current process.

- **About Makefile:** The dependency obj-m specifies the object files to be built into kernel modules (eg. mkdir_trap_v2.o, unlinkat_trap_v2.o). If inp_list.txt is by chance empty, then as a safeguard, a default MODULE_NAME dependency is mentioned (given to be syscall_trap_v2), which acts as a placeholder in the obj-m dependency. So instead of having “obj-m := (empty)” the Makefile will have “obj-m := \$(MODULE_NAME).o”. While running the run.sh file, based on the syscall, its object file name is constructed and placed in the obj-m dependency. The default dependency can be changed accordingly, from syscall_trap_v2, as per need (in case only one module has to be compiled).

The KDIR parameter contains the path to the directory containing necessary makefiles needed for building kernel modules. “make -C \$(KDIR) M=\$(PWD) modules” changes the directory to the kernel build directory, and instructs the kernel build system to compile the modules listed in obj-m. M = \$(PWD) is the path to the directory containing the kernel modules. (We’re running in the same working directory so we use PWD).

- **About run.sh:** Replaces SYSCALL placeholder in “syscall_trap_v2.c” file. Also, based on the input given in inp_list.txt (in the form of <syscall_name>,<ns_1>,<ns_2>,<ns_3>, ..., <ns_(n)>), the parameters PID_QSIZE is changed to n, and the array TARGET_NAMESPACE_PID is updated replacing the placeholder array {12345}. The updated module code is saved in “SYSCALL_trap_v2.c”. The obj-m dependency of the Makefile is updated with the names of all the syscalls whose kprobe modules are needed to be compiled.

After compilation, there’s a prompt to compile all these modules (so “make” is run), and a further prompt to insert them (so “sudo insmod” is run for each module).

7. Testing and Observations

- **Unlinkat:** The unlinkat system call removes a file or directory entry relative to a directory file descriptor.

Using `ll /proc/$$/ns` command, we first find the pid namespace. Then in `inp_list.txt`, make the corresponding entry for blocking `unlinkat` syscall and run the script (`run.sh`).

The kprobe module unlinkat_trap_v2.ko has successfully been inserted.

Run the command `rm <filename>`. The `rm` command internally uses `unlinkat` syscall (we find this from `strace`).

This command gets **killed** (Because of the syscall block).

Using `dmesg` command, we can look at kernel messages. We see the `printk` statements used in the code and messages indicating that the module has successfully blocked the `unlinkat` syscall.

```

[ 456.246242] The current code segment: 10
[ 456.246243] Root user detected, not blocking unlinkat syscall
[ 456.246247] Kprobe handler called
[ 456.246248] Syscall happened for unlinkat
[ 456.246249] The current process PID: 345
[ 456.246250] The current process name: systemd-udevd
[ 456.246252] The current instruction pointer: ffffffffbb54fe601
[ 456.246253] The current code segment: 10
[ 456.246254] Root user detected, not blocking unlinkat syscall
[ 506.158458] Kprobe handler called
[ 506.158461] Syscall happened for unlinkat
[ 506.158461] The current process PID: 4420
[ 506.158462] The current process name: rm
[ 506.158463] The current instruction pointer: ffffffffbb54fe601
[ 506.158463] The current code segment: 10
[ 506.158464] Blocked unlinkat syscall for task in PID namespace: 4026531836
[ 506.158468] BUG: kernel NULL pointer dereference, address: 0000000000000060
[ 506.158471] #PF: supervisor read access in kernel mode
[ 506.158472] #PF: error_code(0x0000) - not-present page
[ 506.158473] PGD 0 P4D 0
[ 506.158475] Oops: 0000 [#1] PREEMPT SMP NOPTI
[ 506.158477] CPU: 1 PID: 4420 Comm: rm Tainted: G             OE        6.8.0-49-generic #49-Ubuntu
[ 506.158479] Hardware name: innotek GmbH VirtualBox/VirtualBox, BIOS VirtualBox 12/01/2006
[ 506.158480] RIP: 0010:__x64_sys_unlinkat+0xa/0x80
[ 506.158484] Code: 85 c0 75 d4 eb c3 66 2e 0f 1f 84 00 00 00 00 00 90 90 90 90 90 90 90 90 90 90 90 e8
fb 29 30 0b 55 48 89 e5 53 <48> 8b 47 60 48 8b 4f 68 8b 5f 70 a9 ff fd ff ff 75 52 31 d2 31 f6
[ 506.158485] RSP: 0018:fffffa8a28606fd70 EFLAGS: 00010246
[ 506.158487] RAX: 0000000000000000 RBX: 0000000000000000 RCX: 0000000000000000
[ 506.158488] RDX: 0000000000000000 RSI: 0000000000000000 RDI: 0000000000000000
[ 506.158488] RBP: ffffffa8a28606fd78 R08: 0000000000000000 R09: 0000000000000000
[ 506.158489] R10: 0000000000000000 R11: 0000000000000000 R12: fffffa8a28606ff58
[ 506.158490] R13: 00000000000000107 R14: 0000000000000000 R15: 0000000000000000
[ 506.158491] FS: 00007f5eb5b174d0(0000) CS: ffff90b23c-0000(0000) k1:cs-0000000000000000

```



```
vaishnavi@vaishnavi-VirtualBox: ~/Downloads/code2/project
vaishnavi@vaishnavi-VirtualBox:~/Downloads/code2/project$ touch file.c
vaishnavi@vaishnavi-VirtualBox:~/Downloads/code2/project$ rm file.c
Killed
vaishnavi@vaishnavi-VirtualBox:~/Downloads/code2/project$ ls
file.c      modules.order  run.sh          unlinkat_trap_v2.ko  unlinkat_trap_v2.mod.o
inp_list.txt Module.symvers syscall_trap_v2.c  unlinkat_trap_v2.mod  unlinkat_trap_v2.o
Makefile    README         unlinkat_trap_v2.c unlinkat_trap_v2.mod.c
vaishnavi@vaishnavi-VirtualBox:~/Downloads/code2/project$ sudo rmmod unlinkat_trap_v2
vaishnavi@vaishnavi-VirtualBox:~/Downloads/code2/project$ rm file.c
vaishnavi@vaishnavi-VirtualBox:~/Downloads/code2/project$ ls
inp_list.txt  Module.symvers  syscall_trap_v2.c  unlinkat_trap_v2.mod  unlinkat_trap_v2.o
Makefile      README          unlinkat_trap_v2.c unlinkat_trap_v2.mod.c
modules.order run.sh          unlinkat_trap_v2.ko unlinkat_trap_v2.mod.o
vaishnavi@vaishnavi-VirtualBox:~/Downloads/code2/project$
```

- Similarly, we have tested out the code for other syscalls like mkdir, mount.
- Alternatively, we tried it for syscalls like execve, which when blocked (without the root user constraint mentioned above) crashes the entire shell since it is an important syscall used by many background processes. So, we created a new shell with a new namespace pid and blocked execve for that namespace pid so that the current shell is still functional and the module can be removed when needed.

eBPF

1.Key Concepts in eBPF

eBPF (Extended Berkeley Packet Filter) is a Linux kernel technology that allows users to execute custom programs within the kernel safely and efficiently. Originally it was designed for packet filtering. Now eBPF has evolved to support a wide range of observability, networking, and security tasks.

- **How eBPF Works:**

eBPF programs are written in C, compiled into bytecode, and verified by the kernel before being loaded. They execute in a restricted environment, ensuring security and stability.

- **Use Cases:**

eBPF programs are used for tasks such as:

- Tracing and monitoring system calls or kernel events.
- Implementing network filters or load balancers.
- Enforcing security policies dynamically without kernel recompilation.

- **Kernel Hooks:**

eBPF programs attach to specific hooks, such as tracepoints, kprobes, or system call entry points, to monitor or manipulate kernel behavior dynamically.

2. eBPF Workflow

The general workflow for implementing eBPF programs is as follows:

1. **Compile eBPF Program:** Write the eBPF program(in C) and compile it into bytecode.
2. **Load eBPF Program:** Load the compiled bytecode into the kernel using a tool like BCC (BPF Compiler Collection).
3. **Attach eBPF Program:** Attach the eBPF program to specific kernel events/hooks.

In this implementation, BCC has been used to integrate eBPF programs into Python scripts, simplifying the interaction between user-space applications and kernel-space monitoring mechanisms.

3. Tools and Libraries Used

The main components used in this implementation are:

- **Python-BCC Library (BPF Compiler Collection):** BCC is a framework that facilitates the integration of eBPF programs with Python scripts. Executing the Python script generates eBPF bytecode and loads it into the kernel.
- **eBPF Maps:** Shared data structures used for communication between user-space and kernel-space. These maps allow storing and retrieving information like namespace identifiers to influence system behavior.
- **strace:** A debugging tool used to trace system calls. In this case, **strace** was used to identify the relevant syscall (**mkdirat**) for the operation we wanted to intercept.

4. Overview of the Implementation

The goal of this implementation was to intercept the **mkdirat** system call and change its behavior for specific namespaces. This was achieved by blocking the directory creation operation in specific namespaces, returning a **-EACCES** error, or Permission Denied instead.

Key Concepts in the Implementation:

- **Kprobes:** A type of eBPF hook that attaches dynamically to kernel functions, such as system calls, and triggers eBPF programs when those functions are entered or exited.
- **Namespace Isolation:** Linux namespaces allow isolating system resources like network, process IDs, and mount points. The goal was to block the **mkdirat** syscall in specific mount namespaces, preventing processes in those namespaces from creating directories.

5. Kernel-Space Program Overview

The kernel-space eBPF program intercepts the **mkdirat** syscall and changes its behavior. This is done in the following way:

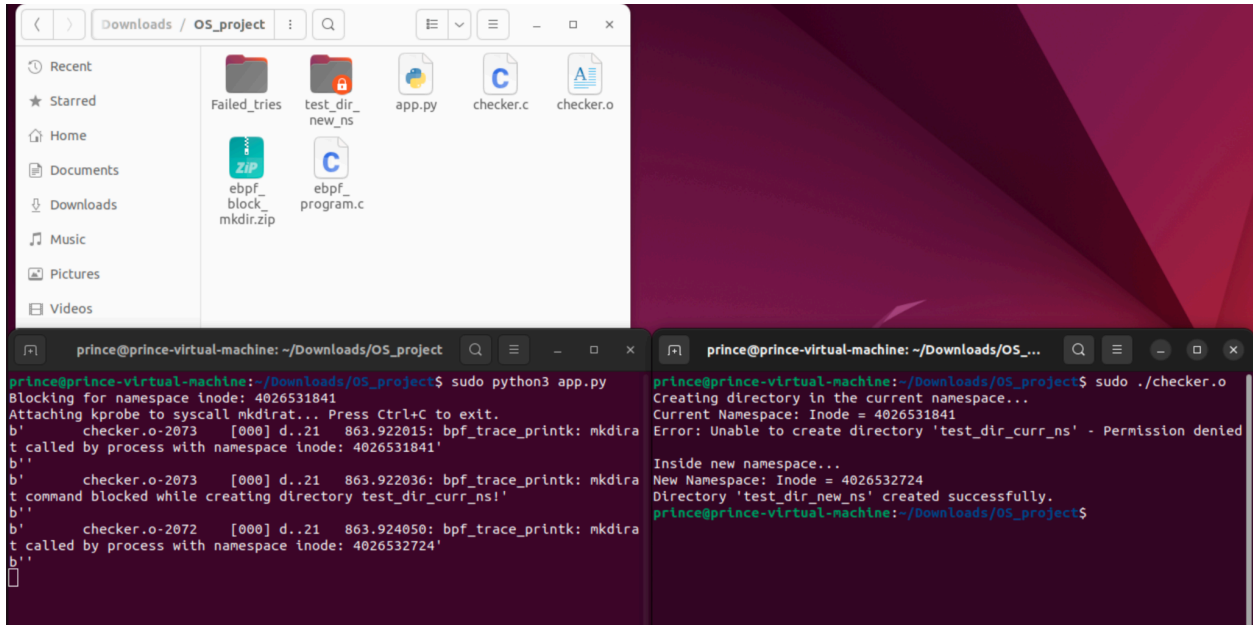
-
1. **Obtain the Namespace Information:** The program accesses the `mnt_ns` (mount namespace) of the calling process by traversing the task structure and its associated namespace proxy (`nsproxy`). The `inode_num` of the current namespace uniquely identifies the namespace.
 2. **Check Namespace in eBPF Map:** The eBPF program checks if the current namespace's `inode_num` exists in a predefined eBPF map (`target_ns`). If the namespace is found in the map, the syscall is blocked.
 3. **Override the Syscall Return Value:** The function `bpf_override_return()` is used to alter the return value of the syscall. If the namespace is restricted, the syscall is blocked, and a `-EACCES` error is returned, thereby preventing the directory creation.
 4. **Logging:** The program uses `bpf_trace_printk()` to log relevant information such as the namespace ID and the filename being processed.

6. User-Space Program Overview

The user-space program, written in Python, interacts with the kernel-space eBPF program to configure the list of namespaces to block. The Python script performs the following steps:

1. **Load the eBPF Program:** The Python script loads the eBPF program into the kernel using BCC.
2. **Attach Kprobe:** The script attaches a kprobe to the `mkdirat` syscall, using the appropriate function prefix for the architecture.
3. **Populate eBPF Map:** The namespaces to be blocked are added to the `target_ns` eBPF map. We have added the current process's mount namespace for demonstration purposes.
4. **Print Trace Output:** The Python script continuously prints the trace output, which logs any blocked `mkdirat` syscalls.

7. About checker.c (Testing) : It tries to create a directory in the current namespace, and then using `unshare(CLONE_NEWNS)`, it tries to create another directory in the new `mnt_namespace` and displays appropriate error messages when it can't.



```

from bcc import BPF
import ctypes as ct
import os

# Import required libraries

# Function to add elements in the shared hash map
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("ebp_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()

```

```

// Compile using strace when 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 = 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, -EACCESS); // Block, give -EACCESS return code i- Permission Denied
    }

    return 0; // allow the syscall
}

```

LSMs

1. Key Concepts in LSMs

- Linux Security Modules (LSMs) are a framework in the Linux kernel that allows various security policies to be applied to control what a process can do on the system.
- LSMs provide a way to add extra security layers on top of traditional Unix permissions, enabling features like mandatory access control, which restricts their access beyond just file permissions (e.g., read and write).
- LSMs allow users to write custom security modules and enable systems to implement advanced security models like:
 1. SELinux (Security-Enhanced Linux): Controls access based on policies that assign users and processes a security context.
 2. AppArmor: Restricts programs to certain capabilities based on profiles that specify allowed operations.
- These modules hook into the kernel's security functions, giving a “yes” or “no” response to whether an action should be allowed

2. Key Concepts in LSMs

- Hook Functions: The Linux kernel has multiple "hook points" for LSMs. A hook is essentially a checkpoint that verifies if an action should proceed, like opening a file or executing a command. When a process tries to perform an action, the kernel calls the hook, which checks if the action aligns with the security policy. Each LSM module can define its own hook function for each point.

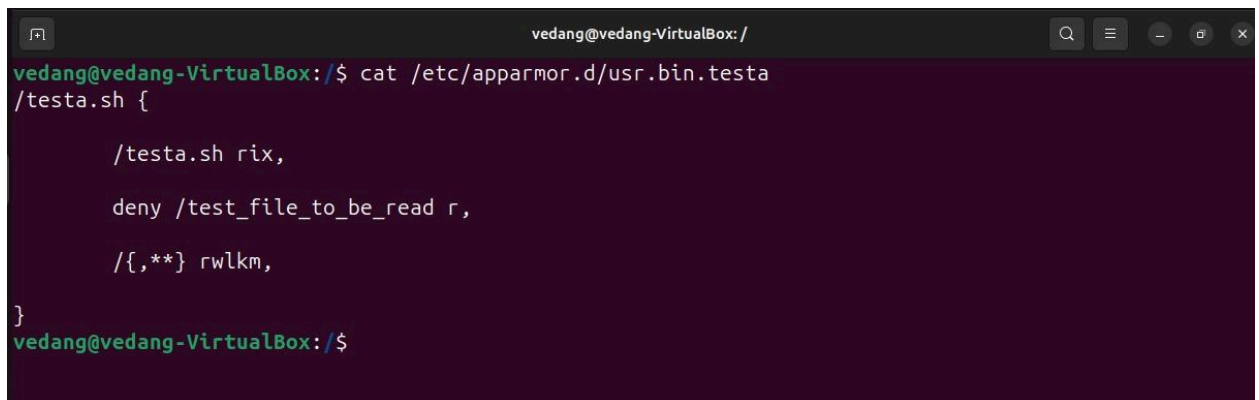
3. AppArmor

- AppArmor (Application Armor) is a Linux Security Module (LSM) that provides a mechanism for access control. It 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 uses profiles, which are sets of rules specifying what files, capabilities, or resources a program can access.
 - Modes: Profiles can be in:
 - Enforce mode: Violations of the profile rules are blocked and logged.
 - Complain mode: Violations are logged but not blocked, useful for debugging.

How AppArmor is implemented

- LSM Framework: AppArmor hooks into the LSM framework in the Linux kernel. The LSM framework allows various security modules to enforce security policies by intercepting system calls and kernel operations.
- Profiles: AppArmor profiles are written in text files located in `/etc/apparmor.d/`. Each profile corresponds to a specific application or process.
- Path-based Access Control: AppArmor enforces security policies based on file paths, specifying which paths a program can read, write, or execute.
- Kernel Module: The AppArmor kernel module enforces the defined profiles by:
 - Hooking into LSM interface functions.
 - Validating system calls against the profile rules.
- User-Space Tools: AppArmor comes with tools for managing profiles, such as:
 - `aa-complain` to switch a profile to complain mode.
 - `aa-enforce` to enforce a profile.
 - `aa-genprof` for generating profiles interactively.
- Example of a profile



```
vedang@vedang-VirtualBox: /
vedang@vedang-VirtualBox:/$ cat /etc/apparmor.d/usr.bin.testa
/testa.sh {
    /testa.sh rix,
    deny /test_file_to_be_read r,
    /[,**} rwlkm,
}
vedang@vedang-VirtualBox:/$
```

The above profile simply prohibits the script `/testa.sh` from reading the `/test_file_to_be_read`. As in the picture, the `deny` keyword locks the read permission (`r`). However, this rule is only for the process run by the `/testa.sh` script. This can be loaded

as follows: `sudo apparmor-parser -r /etc/apparmor.d/usr.bin.testa sudo aa-enforce /etc/apparmor.d/usr.bin.testa` (setting it to enforce mode and not complain)

```
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:/$
```

- a. As we see, the script is denied access to the file, whereas any other process will be permitted unless specified by another apparmor profile.
- b. That's how Apparmor works : It is used to give access permissions for a given process, and not files.
- c. This is useful for a majority of cases such as restricting file accesses, sandboxing environments etc. 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.
- d. System calls are not inherently tied to the file paths : For eg : `read()` carries only the file descriptor and not the file path.
- e. Thus, Apparmor is not a very powerful LSM which can block system calls, but can block the other necessary functionalities leading to the syscall for a given process.

4. Custom LSM as a part of Kernel Code

- Custom Linux Security Modules (LSMs) are integrated into the kernel's security framework to implement specific access control and security policies. These modules allow customization of security behavior by intercepting kernel operations and applying user-defined policies.
- To add a custom LSM, you need to modify the kernel source code, including placing the LSM's implementation in the `security/` directory. This implementation typically contains the security hooks that define the behavior of the module, such as intercepting file accesses or process execution.
- Registration of the LSM within the kernel is done by calling the `security_add_hooks()` function. This function links the hooks defined in your module to the kernel's security

framework, allowing your LSM to take effect during kernel operations. The registration code is executed during the kernel's boot phase.

- The Makefile in the security/ directory must also be updated to include the object file corresponding to your LSM, ensuring it is compiled and linked when the kernel is built.
- To integrate the LSM into the kernel configuration process, an entry must be added to the Kconfig file in the security/ directory. This entry defines the configuration options for the LSM, such as whether it should be built into the kernel or compiled as a module. It also includes descriptive text to guide users in enabling the LSM through configuration tools like make menuconfig.

```
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:/$
```