**Assignment**

**Problem statement 1: Drop packets using eBPF**

Here’s a working demo split into two parts:

1. eBPF program in C (compiled with clang/LLVM to BPF)
2. Userspace loader in C (or Python via libbpf/bcc) to set the port dynamically

**1. eBPF Program (C)**

This program hooks at the TC ingress (traffic control) or XDP layer and drops TCP packets if they match the configured port.

// drop\_port\_kern.c

#include <linux/bpf.h>

#include <bpf/bpf\_helpers.h>

#include <linux/ip.h>

#include <linux/tcp.h>

struct {

\_\_uint(type, BPF\_MAP\_TYPE\_ARRAY);

\_\_uint(max\_entries, 1);

\_\_type(key, \_\_u32);

\_\_type(value, \_\_u16);

} blocked\_port SEC(".maps");

SEC("tc")

int drop\_tcp\_port(struct \_\_sk\_buff \*skb) {

void \*data = (void \*)(long)skb->data;

void \*data\_end = (void \*)(long)skb->data\_end;

struct iphdr \*ip = data;

if ((void \*)(ip + 1) > data\_end)

return BPF\_OK;

if (ip->protocol != IPPROTO\_TCP)

return BPF\_OK;

struct tcphdr \*tcp = (void \*)(ip + ip->ihl \* 4);

if ((void \*)(tcp + 1) > data\_end)

return BPF\_OK;

\_\_u32 key = 0;

\_\_u16 \*port = bpf\_map\_lookup\_elem(&blocked\_port, &key);

if (!port)

return BPF\_OK;

if (tcp->dest == bpf\_htons(\*port)) {

// Drop the packet

return BPF\_DROP;

}

return BPF\_OK;

}

char LICENSE[] SEC("license") = "GPL";

* By default, it checks TCP packets and compares destination port against the configured one.
* blocked\_port is stored in a BPF map (array with single entry).
* Port check uses bpf\_htons because TCP headers store ports in network byte order.

**2. Userspace Loader (C with libbpf)**

This loads the eBPF program and updates the blocked port (default: 4040, configurable via CLI arg).

// drop\_port\_user.c

#include <stdio.h>

#include <stdlib.h>

#include <bpf/libbpf.h>

#include <bpf/bpf.h>

int main(int argc, char \*\*argv) {

struct bpf\_object \*obj;

int prog\_fd, map\_fd;

char filename[] = "drop\_port\_kern.o";

int err;

\_\_u32 key = 0;

\_\_u16 port = 4040; // default

if (argc > 1) {

port = atoi(argv[1]);

}

// Open and load BPF object

obj = bpf\_object\_\_open\_file(filename, NULL);

if (libbpf\_get\_error(obj)) {

fprintf(stderr, "Error opening BPF object file\n");

return 1;

}

err = bpf\_object\_\_load(obj);

if (err) {

fprintf(stderr, "Error loading BPF object\n");

return 1;

}

// Get map fd

map\_fd = bpf\_object\_\_find\_map\_fd\_by\_name(obj, "blocked\_port");

if (map\_fd < 0) {

fprintf(stderr, "Failed to find map\n");

return 1;

}

// Update map with chosen port

if (bpf\_map\_update\_elem(map\_fd, &key, &port, BPF\_ANY) != 0) {

fprintf(stderr, "Failed to update port map\n");

return 1;

}

printf("Blocking TCP port %d\n", port);

// Attach program to TC ingress of interface (example: eth0)

struct bpf\_program \*prog = bpf\_object\_\_find\_program\_by\_name(obj, "drop\_tcp\_port");

prog\_fd = bpf\_program\_\_fd(prog);

// Example attach (replace "eth0" with your iface):

system("tc qdisc add dev eth0 clsact 2>/dev/null");

char cmd[256];

snprintf(cmd, sizeof(cmd),

"tc filter add dev eth0 ingress bpf direct-action obj %s sec tc",

filename);

system(cmd);

printf("Program loaded. Press Ctrl+C to exit.\n");

while (1) sleep(1);

}

**3. Build & Run**

# Compile kernel program

clang -O2 -target bpf -c drop\_port\_kern.c -o drop\_port\_kern.o

# Compile userspace program

gcc -o drop\_port\_user drop\_port\_user.c -lbpf

# Run with default (4040)

sudo ./drop\_port\_user

# Or specify port

sudo ./drop\_port\_user 8080

This will drop all TCP traffic destined for the configured port.  
We can change the blocked port dynamically by re-running the user program with another port.

**Problem statement 2: Drop packets only for a given process**

now we have two conditions:

1. Process name check (eBPF needs to know which process is sending the packet).
2. Port check (only allow TCP traffic to one port, e.g. 4040).

* eBPF at the network layer (XDP/TC) doesn’t have direct knowledge of the sending process. It only sees packets, not tasks.
* To filter traffic by process name (comm), we need to hook closer to the socket layer (like cgroup/bind/connect/sendmsg hooks), not just at XDP/TC.
* cgroup socket eBPF programs are ideal here: they let you filter connections per process, per port.

We’ll use a cgroup socket eBPF program (cgroup/connect4):

* It runs when a process tries to establish an outgoing TCP connection.
* We’ll check the process’s name (bpf\_get\_current\_comm).
* If the process matches (myprocess), we allow only port 4040.
* Any other port → connection is blocked.
* For all other processes, connections are unaffected.

**1. eBPF Program (C)**

// allow\_port\_by\_process.c

#include <linux/bpf.h>

#include <bpf/bpf\_helpers.h>

#include <linux/in.h>

char LICENSE[] SEC("license") = "GPL";

// Allowed process name

const char target\_comm[] = "myprocess";

// Allowed TCP port (default 4040)

const \_\_u16 allowed\_port = 4040;

SEC("cgroup/connect4")

int allow\_only\_port(struct bpf\_sock\_addr \*ctx) {

char comm[TASK\_COMM\_LEN];

// Get process name

if (bpf\_get\_current\_comm(&comm, sizeof(comm)) == 0) {

// Check if it's our target process

if (\_\_builtin\_memcmp(comm, target\_comm, sizeof(target\_comm)) == 0) {

// If process matches but port != 4040 -> deny

if (ctx->user\_port != bpf\_htons(allowed\_port)) {

return 0; // reject connection

}

}

}

return 1; // allow

}

**2. Userspace Loader (C with libbpf)**

// load\_prog.c

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <bpf/libbpf.h>

int main(int argc, char \*\*argv) {

struct bpf\_object \*obj;

int prog\_fd;

int err;

char filename[] = "allow\_port\_by\_process.o";

obj = bpf\_object\_\_open\_file(filename, NULL);

if (libbpf\_get\_error(obj)) {

fprintf(stderr, "Error opening BPF object\n");

return 1;

}

err = bpf\_object\_\_load(obj);

if (err) {

fprintf(stderr, "Error loading BPF object\n");

return 1;

}

struct bpf\_program \*prog = bpf\_object\_\_find\_program\_by\_name(obj, "allow\_only\_port");

if (!prog) {

fprintf(stderr, "Program not found\n");

return 1;

}

prog\_fd = bpf\_program\_\_fd(prog);

// Attach to cgroup (example: /sys/fs/cgroup/mygroup)

char \*cgroup\_path = "/sys/fs/cgroup";

int cg\_fd = open(cgroup\_path, O\_DIRECTORY | O\_RDONLY);

if (cg\_fd < 0) {

perror("open cgroup");

return 1;

}

err = bpf\_prog\_attach(prog\_fd, cg\_fd, BPF\_CGROUP\_INET4\_CONNECT, 0);

if (err) {

perror("bpf\_prog\_attach");

return 1;

}

printf("eBPF program attached to cgroup %s\n", cgroup\_path);

while (1) sleep(1);

}

**3. Build & Run**

# Compile eBPF program

clang -O2 -target bpf -c allow\_port\_by\_process.c -o allow\_port\_by\_process.o

# Compile userspace loader

gcc -o load\_prog load\_prog.c -lbpf

# Create a cgroup

sudo mkdir -p /sys/fs/cgroup/mygroup

# Run loader (attaches to cgroup)

sudo ./load\_prog

Then, move your process into the cgroup:

echo <PID-of-myprocess> | sudo tee /sys/fs/cgroup/mygroup/cgroup.procs

What happens?

* If myprocess tries to connect to TCP port 4040, it succeeds.
* If myprocess tries any other port → connection denied.
* Other processes are not affected.

**Problem Statement 3: Explain the code snippet**

**The Code**

package main

import "fmt"

func main() {

cnp := make(chan func(), 10) // buffered channel of functions

for i := 0; i < 4; i++ {

go func() { // launch 4 goroutines

for f := range cnp { // keep reading funcs from channel

f() // execute each function

}

}()

}

cnp <- func() {

fmt.Println("HERE1")

}

fmt.Println("Hello")

}

How the highlighted constructs work

1. make(chan func(), 10)
   * This creates a buffered channel that can hold functions with no arguments and no return values.
   * Capacity = 10 → you can send up to 10 function values into the channel without blocking.
   * Each function is like a "task" to be executed later.

Use-case: Building worker pools or task queues where work units are functions pushed onto the channel.

1. for i := 0; i < 4; i++ { go func() { … }() }
   * Starts 4 worker goroutines.
   * Each worker runs:

for f := range cnp {

f()

}

* + This means each goroutine keeps pulling tasks (functions) from the channel and executing them.
  + When the channel is closed, the loop exits.

Use-case: Implementing a worker pool pattern (multiple goroutines consuming tasks from a shared channel).

1. Sending into channel:

cnp <- func() {

fmt.Println("HERE1")

}

* + Sends an anonymous function into the channel.
  + One of the worker goroutines will eventually receive it and execute it → fmt.Println("HERE1").

What is the significance of the for loop with 4 iterations?

* It spawns 4 worker goroutines, so tasks (functions) from the channel can be executed in parallel.
* Why 4? Often chosen to match CPU cores, so you can utilize concurrency for processing multiple tasks.

What is the significance of make(chan func(), 10)?

* The buffer (10) means you can enqueue up to 10 tasks without blocking.
* It decouples task producers (main goroutine) from task consumers (worker goroutines).
* If it were unbuffered (make(chan func())), the main goroutine would block until a worker picked up the function.

Why is “HERE1” not getting printed?

Because of program exit timing:

* After sending the function into the channel, main immediately prints "Hello" and then returns.
* When main returns, the program ends → all goroutines are killed instantly.
* The worker goroutines don’t get enough time to read from the channel and execute the function.

That’s why "HERE1" is never printed.

Fix

To actually see "HERE1", you need synchronization, e.g. using a sync.WaitGroup or time.Sleep:

package main

import (

"fmt"

"sync"

)

func main() {

cnp := make(chan func(), 10)

var wg sync.WaitGroup

for i := 0; i < 4; i++ {

go func() {

for f := range cnp {

f()

wg.Done()

}

}()

}

wg.Add(1)

cnp <- func() {

fmt.Println("HERE1")

}

wg.Wait()

fmt.Println("Hello")

}

Now "HERE1" will always be printed before "Hello".

Summary

* make(chan func(), 10) → buffered channel storing functions (tasks).
* for i := 0; i < 4; i++ { go … } → spawns a 4-worker pool.
* Workers continuously read tasks from the channel and execute them.
* "HERE1" doesn’t print because the program exits before workers run it.