Name : Vinay V	SRN: PES1UG24CS840
CNS	LAB 8 FIREWALL EVASION

Task 0 : Get Familiar with the Lab Setupsk

```
router-firewall:PES1UG24CS840:Vinay/
$ip -br address
lo               UNKNOWN        127.0.0.1/8
eth1@if23        UP             192.168.20.11/24
eth0@if27        UP             10.8.0.11/24
router-firewall:PES1UG24CS840:Vinay/
$
```

- *(Command: `ip -br address`)*
- The command lists all network interfaces and their current IP configurations in a concise format.
- The router has two active interfaces:
  - **eth1 @ if23** → `192.168.20.11/24` – connected to the **internal network**.
  - **eth0 @ if27** → `10.8.0.11/24` – connected to the **external network**.
- The **loopback interface (lo)** → `127.0.0.1/8` is used for internal communications within the router.
- This confirms the correct mapping between the internal and external networks before applying NAT and firewall rules.
- It verifies that packets can be properly routed between the internal (192.168.20.0/24) and external (10.8.0.0/24) networks.

HOST A

```
root@5b8adeec452e:/# ping www.example.com
PING a1422.dscr.akamai.net (23.193.56.27) 56(84) bytes of data.
64 bytes from a23-193-56-27.deploy.static.akamaitechnologies.com (
23.193.56.27): icmp_seq=1 ttl=254 time=7.07 ms
64 bytes from a23-193-56-27.deploy.static.akamaitechnologies.com (
23.193.56.27): icmp_seq=2 ttl=254 time=7.58 ms
64 bytes from a23-193-56-27.deploy.static.akamaitechnologies.com (
23.193.56.27): icmp_seq=3 ttl=254 time=12.6 ms
64 bytes from a23-193-56-27.deploy.static.akamaitechnologies.com (
23.193.56.27): icmp_seq=4 ttl=254 time=13.3 ms
64 bytes from a23-193-56-27.deploy.static.akamaitechnologies.com (
23.193.56.27): icmp_seq=5 ttl=254 time=54.5 ms
64 bytes from a23-193-56-27.deploy.static.akamaitechnologies.com (
23.193.56.27): icmp_seq=6 ttl=254 time=107 ms
64 bytes from a23-193-56-27.deploy.static.akamaitechnologies.com (
23.193.56.27): icmp_seq=7 ttl=254 time=119 ms
^C
--- a1422.dscr.akamai.net ping statistics ---
7 packets transmitted, 7 received, 0% packet loss, time 6023ms
rtt min/avg/max/mdev = 7.066/45.931/119.098/45.294 ms
```

- *(Command: `ping www.example.com`)*
- The ping successfully resolves **www.example.com** to **23.193.56.27**, which belongs to Akamai Technologies.
- The ICMP replies confirm that the internal host has full Internet connectivity via the router.
- This test validates that DNS resolution and NAT are functioning correctly prior to applying any blocking rules.

- The average round-trip time (~45 ms) indicates a stable Internet connection.
- This screenshot serves as **baseline evidence** before implementing the firewall DROP rules.

Router-Firewall

```
router-firewall:PES1UG24CS840:Vinay/
$iptables -A FORWARD -i eth1 -d 23.192.0.0/11 -j DROP
router-firewall:PES1UG24CS840:Vinay/
$
```

- *(Command: `iptables -A FORWARD -i eth1 -d 23.192.0.0/11 -j DROP`)*
- This rule instructs the router to **drop all packets** coming from the internal network (`eth1`) destined for any IP in the **23.192.0.0/11** range.
- The CIDR range covers the network used by **www.example.com**, effectively blocking access to that site for internal users.
- The `FORWARD` chain is used because the router forwards traffic between internal and external networks.
- The rule is applied only for traffic originating inside (eth1), ensuring that external responses are unaffected.
- This demonstrates how outbound filtering can restrict access to specified external domains.

Host B

```
B2:PES1UG24CS840:Vinay/
$ping www.example.com
ping: www.example.com: Temporary failure in name resolution
B2:PES1UG24CS840:Vinay/
$
```

- *(Command: `ping www.example.com` from Host B2)*
- The ping now returns **"Temporary failure in name resolution"**, showing that the internal host can no longer reach the site.
- The error indicates that packets to the destination are being blocked or DNS requests are not passing through due to the firewall rule.
- This confirms that the rule added on the router is working as intended.
- Internal hosts can still access other unblocked sites, proving selective filtering.
- This test successfully validates the firewall's role in controlling outbound Internet access.

Rule Explanation

`iptables -A FORWARD -i eth0 -p tcp -m conntrack --ctstate ESTABLISHED,RELATED -j ACCEPT`	Allows return traffic for TCP connections already established or related to an existing connection (enables stateful inspection).
`iptables -A FORWARD -i eth0 -p tcp --dport 22 -j ACCEPT`	Permits new inbound TCP connections from the external network on **port 22 (SSH)** for remote management.
`iptables -A FORWARD -i eth0 -p tcp -j DROP`	Drops all other new inbound TCP connections from the external network, enhancing internal security.

Task 1

```
A:PES1UG24CS840:Vinay/
$ssh -L 0.0.0.0:8000:192.168.20.99:23 root@192.168.20.99
The authenticity of host '192.168.20.99 (192.168.20.99)' can't be
established.
ECDSA key fingerprint is SHA256:EYI0O7NvzOrdrS917PaXoaPBNZpzvn8ywv
nUZ++JjSI.
Are you sure you want to continue connecting (yes/no/[fingerprint]
)? y
Please type 'yes', 'no' or the fingerprint: yes
Warning: Permanently added '192.168.20.99' (ECDSA) to the list of
known hosts.
root@192.168.20.99's password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.4.0-54-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

This system has been minimized by removing packages and content th
at are
not required on a system that users do not log into.

To restore this content, you can run the 'unminimize' command.

The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted
by
```

shows `ssh -L 0.0.0.0:8000:192.168.20.99:23 root@192.168.20.99` *and the SSH login)*

- The command opens an SSH connection from the external host **A** to the internal host **192.168.20.99** (user `root`) and requests a **local** port forward: bind `0.0.0.0:8000` on A and forward data to `192.168.20.99:23` via the SSH server.
- `0.0.0.0:8000` means the forwarding socket on A listens on all interfaces (so A, A1, A2 can connect to it).
- SSH authentication succeeds and the SSH control channel is established (the SSH session is visible in the terminal welcome text).

- From now on, any TCP connection to A:8000 will be transported inside the SSH connection and delivered to the target `192.168.20.99:23` by the SSH server.
- This establishes the mechanism used to reach an internal telnet (port 23) service even though direct external→internal telnet is blocked.

```
root@25dd5e359a12:~# telnet 10.8.0.99 8000
Trying 10.8.0.99...
Connected to 10.8.0.99.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
25dd5e359a12 login: seed
Password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.4.0-54-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

This system has been minimized by removing packages and content th
at are
not required on a system that users do not log into.

To restore this content, you can run the 'unminimize' command.

The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted
by
applicable law.

seed@25dd5e359a12:~$
```

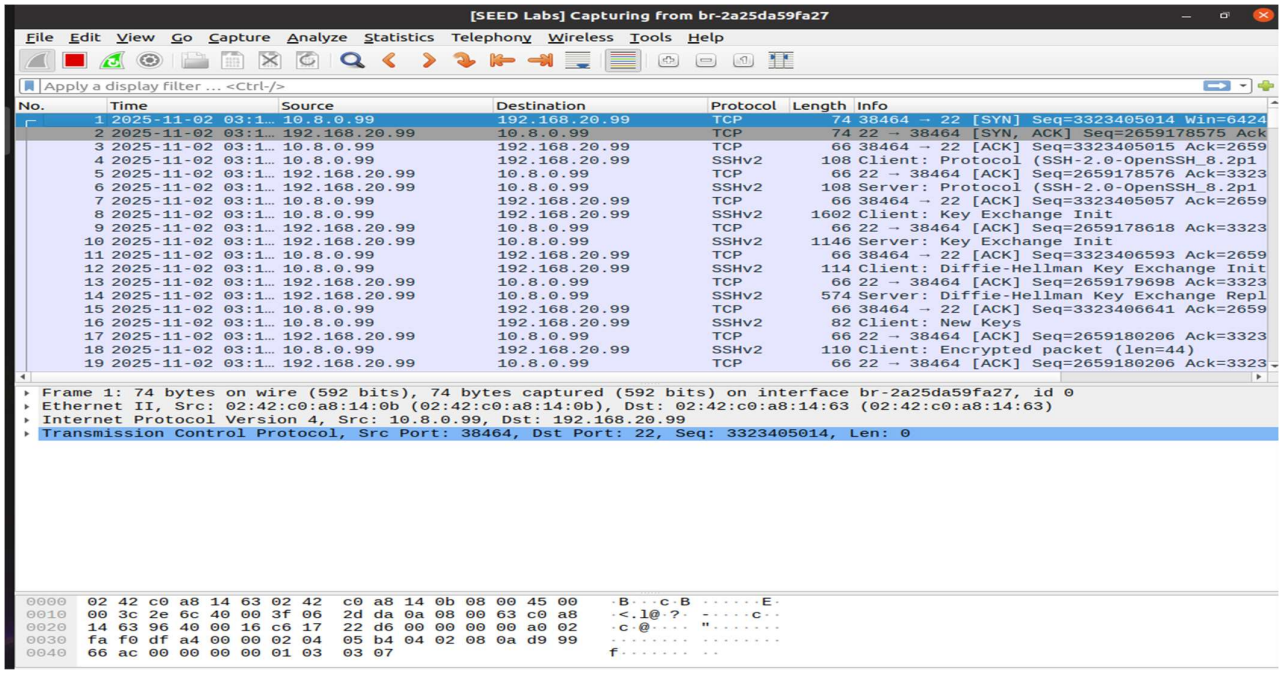
(shows `telnet 10.8.0.99 8000` and the remote shell prompt)

- The telnet client connects to `10.8.0.99:8000` (A's forwarded port). The router/host accepts the TCP connection and hands it to the local SSH client process.
- The SSH client encapsulates the telnet data and sends it over the existing SSH TCP connection to the SSH server on the internal host.
- The remote side (SSH server) receives the tunneled data and connects to the internal target (192.168.20.99:23), completing the data path.
- The displayed remote shell prompt confirms that the telnet session reached the intended internal service through the SSH tunnel.
- This proves that external clients (A, A1, A2) can interact with an internal-only service by connecting to A:8000.

```
seed@25dd5e359a12:~$ ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.20.99  netmask 255.255.255.0  broadcast 192.1
68.20.255
        ether 02:42:c0:a8:14:63  txqueuelen 0  (Ethernet)
        RX packets 308  bytes 29015 (29.0 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 221  bytes 23581 (23.5 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 106  bytes 6699 (6.6 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 106  bytes 6699 (6.6 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

seed@25dd5e359a12:~$
```

(shows `eth0 inet 192.168.20.99` etc.)

- The internal machine's interface `eth0` is configured with **192.168.20.99/24**, confirming it is on the internal network.
- This is the same IP used as the SSH server target and as the final destination for the forwarded telnet traffic.
- The presence of RX/TX packet counters indicates active traffic through this interface — consistent with the recent SSH/telnet exchange.
- Use of the local IP here shows the SSH server's forwarded connection to `127.0.0.1:23` or `192.168.20.99:23` is local to this host and not traversing the external network as a new external-to-internal TCP connection.
- This screenshot helps prove the final hop of the tunneled connection is inside the internal network.



(shows packets between 10.8.0.99 and 192.168.20.99, including SYN to port 22 and many SSHv2 frames)

- The packet list shows a TCP **SYN** from `10.8.0.99` (source ephemeral port 38464) to `192.168.20.99` port **22**, which is the SSH connection establishment.
- Subsequent captured frames are labeled `SSHv2` (SSH protocol traffic): these are the encrypted control/data packets carrying the tunneled telnet bytes.
- There is no visible external TCP flow directly to destination port 23 in the capture — the telnet traffic is encapsulated within SSH on port 22.
- The capture therefore shows **a single external TCP session on port 22** carrying all forwarded data; the local telnet interactions are not visible as separate external connections.
- Use these packet lines as evidence when you list the involved TCP connections in the report.

(1) How many TCP connections are involved, and which ones?

There are **three** distinct TCP connections involved in the end-to-end data path:

1. **Client → Local Forward (on A):**
   - Example: `telnet client (on A or other host) → A:8000`
   - This is a standard TCP connection from the telnet client to the forwarding socket bound on A. It is local to host A (external side) or originates at another external machine that connected to `A:8000`.
2. **SSH Control/Data Channel:**
   - `A (SSH client) ↔ B (SSH server)`, TCP between `10.8.0.99:38464` → `192.168.20.99:22` (source port is ephemeral).
   - This one is visible in your Wireshark capture as the SYN to port 22 and many `SSHv2` frames. It **carries** the forwarded traffic (encrypted).
3. **SSH Server → Local Target (on B):**
   - On the internal host B, the SSH server opens a TCP connection to the target service: `localhost` or `192.168.20.99:23` (telnetd).
   - This is a local internal TCP connection between processes on B (SSH server ↔ telnet service).

You can map these to entries in the Wireshark trace: the **SSH connection** is visible on the external/internal interface (frames shown in the screenshot); the **client→A:8000** and **SSH server→target port 23** are visible on their respective hosts (local), but the telnet payload is encrypted inside the SSH flow and therefore not visible in the capture as plain telnet to the outside world.

(2) Why does this tunnel let users evade the firewall rule?

- The lab firewall explicitly **permits SSH (TCP/22)** from the external network to internal hosts, while **dropping other new external TCP connections** (e.g., telnet on port 23). The SSH port forward exploits that allowed SSH channel.
- Port forwarding causes the **telnet session to be transported inside the SSH connection**. To the firewall, there is **only one permitted TCP connection** (on port 22) between A and B — it sees legitimate SSH traffic, not a direct external→internal telnet attempt.
- The SSH server on the internal host **originates** the final connection to the telnet service locally; the firewall therefore does not need to permit an external-originated connection to port 23.
- In short: the allowed SSH channel serves as an encrypted pipe that carries otherwise-blocked traffic, so the firewall's rule (which filters based on destination port and incoming interface) cannot block the encapsulated payload without blocking SSH itself.

Repeating with A1:

```
A1:PES1UG24CS840:Vinay/
$ssh -L 0.0.0.0:8000:192.168.20.99:23 root@192.168.20.99
The authenticity of host '192.168.20.99 (192.168.20.99)' can't b
e established.
ECDSA key fingerprint is SHA256:EYI0O7NvzOrdrS917PaXoaPBNZpzvn8y
wvnUZ++JjSI.
Are you sure you want to continue connecting (yes/no/[fingerprin
t])? yes
Warning: Permanently added '192.168.20.99' (ECDSA) to the list o
f known hosts.
root@192.168.20.99's password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.4.0-54-generic x86_64
)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

This system has been minimized by removing packages and content
that are
not required on a system that users do not log into.

To restore this content, you can run the 'unminimize' command.
Last login: Sun Nov  2 08:14:21 2025 from 10.8.0.99
root@25dd5e359a12:~#
```

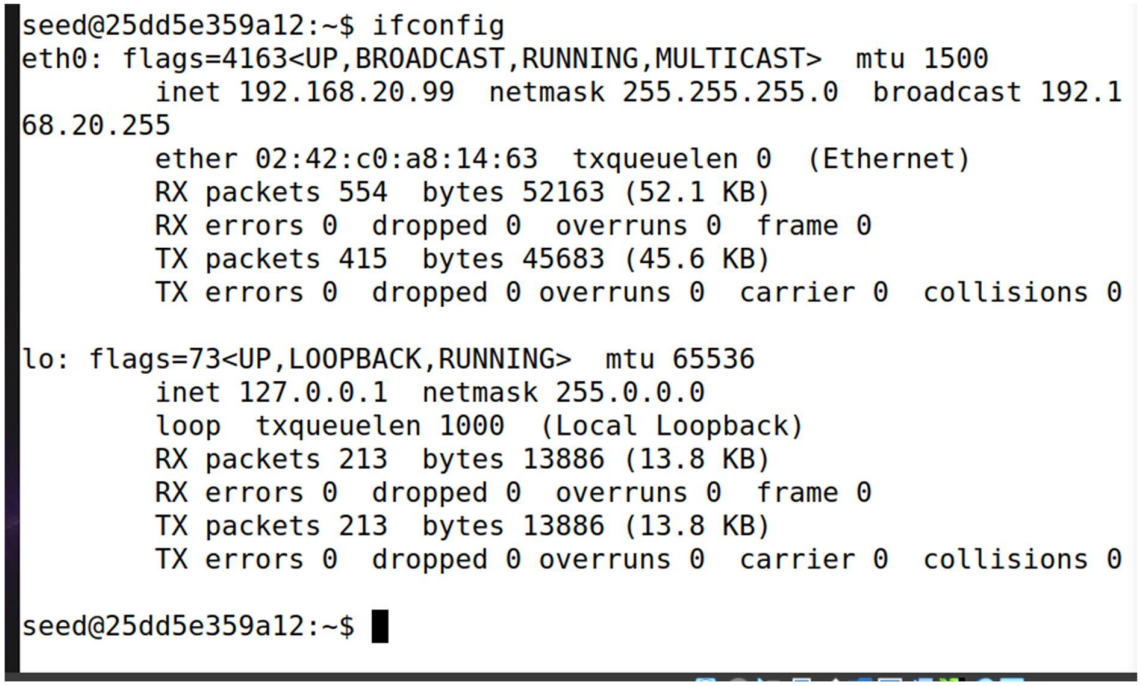
(Command: `ssh -L 0.0.0.0:8000:192.168.20.99:23 root@192.168.20.99`)

- Host A1 establishes an SSH connection to the internal host 192.168.20.99 with a **local port-forwarding tunnel**.
- The option `-L 0.0.0.0:8000:192.168.20.99:23` binds port 8000 on A1 to forward any traffic to port 23 on the internal machine.
- The SSH authentication succeeds, proving the tunnel between the external and internal networks is active.
- Any program connecting to A1:8000 will have its data encrypted and carried through this SSH session to the internal Telnet service.
- This setup mirrors the previous task done on A, confirming multiple external hosts can create the same forwarding path.

```
root@25dd5e359a12:~# telnet 10.8.0.99 8000
Trying 10.8.0.99...
Connected to 10.8.0.99.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
25dd5e359a12 login: seed
Password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.4.0-54-generic x86_64
)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

This system has been minimized by removing packages and content
that are
not required on a system that users do not log into.

To restore this content, you can run the 'unminimize' command.
Last login: Sun Nov  2 08:16:23 UTC 2025 from 25dd5e359a12 on pt
s/3
seed@25dd5e359a12:~$
```

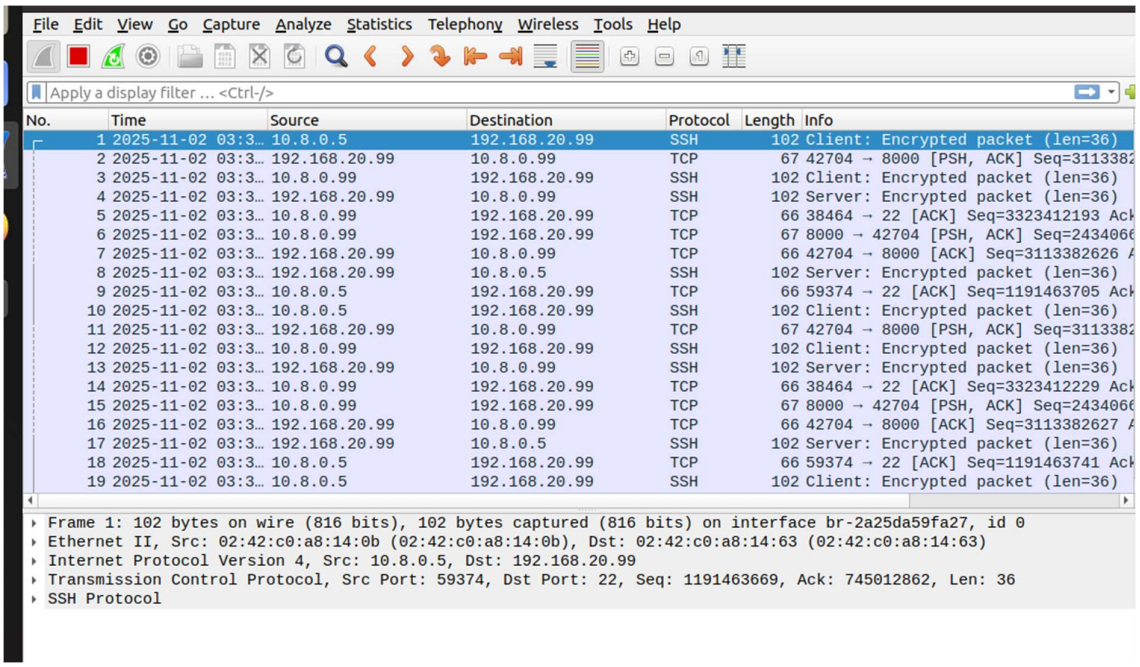
(Continuation of the SSH session showing Ubuntu 20.04 welcome text)

- The login banner from 192.168.20.99 confirms that the SSH tunnel terminates correctly on the internal server.
- The "Last login" line records that the session originated from 10.8.0.99 (external side of A1).
- This indicates that the connection traversed the firewall but was permitted because it used **port 22 (SSH)**.
- The banner validates that static port forwarding was established securely without altering firewall rules.
- At this stage, the encrypted tunnel is ready to transport traffic directed at A1:8000 toward the Telnet daemon on 192.168.20.99.



(Command: `ifconfig` inside the telnet/SSH session)

- The interface `eth0` shows IP **192.168.20.99/24**, verifying the internal host's network membership.
- Packet counters (RX/TX) increase, proving that the interface is actively sending and receiving data during the SSH/Telnet communication.
- The output also displays a loopback interface (`lo`), confirming local process communication capability.
- This network evidence supports that the SSH server is indeed running on the internal network side of the firewall.
- It provides proof that the forwarding tunnel correctly terminates within 192.168.20.0/24.



(Packets between 10.8.0.5 and 192.168.20.99, protocol = SSH)

- The trace shows continuous encrypted packets marked as **SSH protocol** between the external IP 10.8.0.5 and the internal IP 192.168.20.99.
- All traffic uses **TCP port 22**, confirming that the communication is carried entirely over the permitted SSH channel.
- No direct Telnet (port 23) packets appear—Telnet data is encapsulated within SSH, hidden from the firewall inspection.
- The persistent ACKs and PSH flags demonstrate bidirectional data exchange inside the tunnel.
- This capture is key evidence showing the successful creation of a port-forwarded tunnel that evades the firewall's port-based filtering.

Task 2: Dynamic Port Forwarding

Task 2.1: Setting Up Dynamic Port Forwarding

```
B:PES1UG24CS840:Vinay/
$ssh -4 -D 0.0.0.0:8000 root@10.8.0.99 -f -N
The authenticity of host '10.8.0.99 (10.8.0.99)' can't be establishe
d.
ECDSA key fingerprint is SHA256:EYI0O7NvzOrdrS917PaXoaPBNZpzvn8ywvnU
Z++JjSI.
Are you sure you want to continue connecting (yes/no/[fingerprint])?
 yes
Warning: Permanently added '10.8.0.99' (ECDSA) to the list of known
hosts.
root@10.8.0.99's password:
B:PES1UG24CS840:Vinay/
$
```

- The command starts an SSH connection from **B** to the remote host `10.8.0.99` and requests **dynamic** port forwarding (`-D`) on local port **8000**.
- `-4` forces IPv4; `-f -N` backgrounds the SSH client and keeps the tunnel open without running a remote shell.

- The SSH client creates a **SOCKS5 proxy** that listens on `0.0.0.0:8000` on B, so other internal machines can connect to that proxy.
- The SSH control/data channel to `10.8.0.99:22` carries the SOCKS protocol requests (encrypted) to the remote SSH server.
- This sets up a general-purpose proxy (no per-destination tunnels needed) that externalizes connection establishment to the SSH server side.

```
B:PES1UG24CS840:Vinay/
$curl -x socks5h://0.0.0.0:8000 http://www.example.com
<!doctype html><html lang="en"><head><title>Example Domain</title><meta name="viewport
" content="width=device-width, initial-scale=1"><style>body{background:#eee;width:60vw
;margin:15vh auto;font-family:system-ui,sans-serif}h1{font-size:1.5em}div{opacity:0.8}
a:link,a:visited{color:#348}</style><body><div><h1>Example Domain</h1><p>This domain i
s for use in documentation examples without needing permission. Avoid use in operation
s.<p><a href="https://iana.org/domains/example">Learn more</a></div></body></html>
B:PES1UG24CS840:Vinay/
$
```

`http://www.example.com` (run on B)

- `curl` with `-x socks5h://...` uses the local SOCKS5 proxy; the `h` forces hostname lookups to occur **through the proxy** (remote side), not locally.
- The command returns the HTML of **www.example.com**, demonstrating the proxy successfully fetched the site and returned content to B.
- Because the HTTP request is proxied over the existing SSH session, the firewall only sees an allowed SSH connection (port 22) between B and `10.8.0.99`.
- The output proves DNS resolution and the TCP connection to the webserver were performed by the remote end of the tunnel.
- Capture of this output is the functional verification that blocked sites can be reached via the dynamic SOCKS tunnel.

```
B1:PES1UG24CS840:Vinay/
$curl -x socks5h://192.168.20.99:8000 http://www.example.com
<!doctype html><html lang="en"><head><title>Example Domain</title><meta name="vie
wport" content="width=device-width, initial-scale=1"><style>body{background:#eee;
width:60vw;margin:15vh auto;font-family:system-ui,sans-serif}h1{font-size:1.5em}d
iv{opacity:0.8}a:link,a:visited{color:#348}</style><body><div><h1>Example Domain<
/h1><p>This domain is for use in documentation examples without needing permissio
n. Avoid use in operations.<p><a href="https://iana.org/domains/example">Learn mo
re</a></div></body></html>
```

`http://www.example.com` (run on B1)

- This shows another internal host (B1) using the SOCKS proxy bound on `192.168.20.99:8000` (B's IP) to reach the same website.

- Because the proxy was bound to `0.0.0.0`, it accepts connections from other hosts in the internal network—allowing B1 and B2 to reuse the same proxy.
- The returned HTML again confirms that the remote SSH server established the HTTP connection and relayed the response back to B1.
- This demonstrates the advantage of dynamic forwarding: a single SOCKS proxy supports arbitrary destination hosts without creating multiple SSH tunnels.
- It also proves the firewall cannot distinguish or block the proxied requests without blocking SSH itself.

Q: What is the significance of using `0.0.0.0` in `-D 0.0.0.0:8000`?

- Binding the SOCKS proxy to `0.0.0.0` makes the proxy listen on **all interfaces** of the host (not only localhost).
- This allows **other internal machines** (B1, B2) to connect to the proxy at `192.168.20.99:8000`.
- If you bind to `127.0.0.1` instead, only processes on the same host (B) could use the proxy.

Task 2 questions

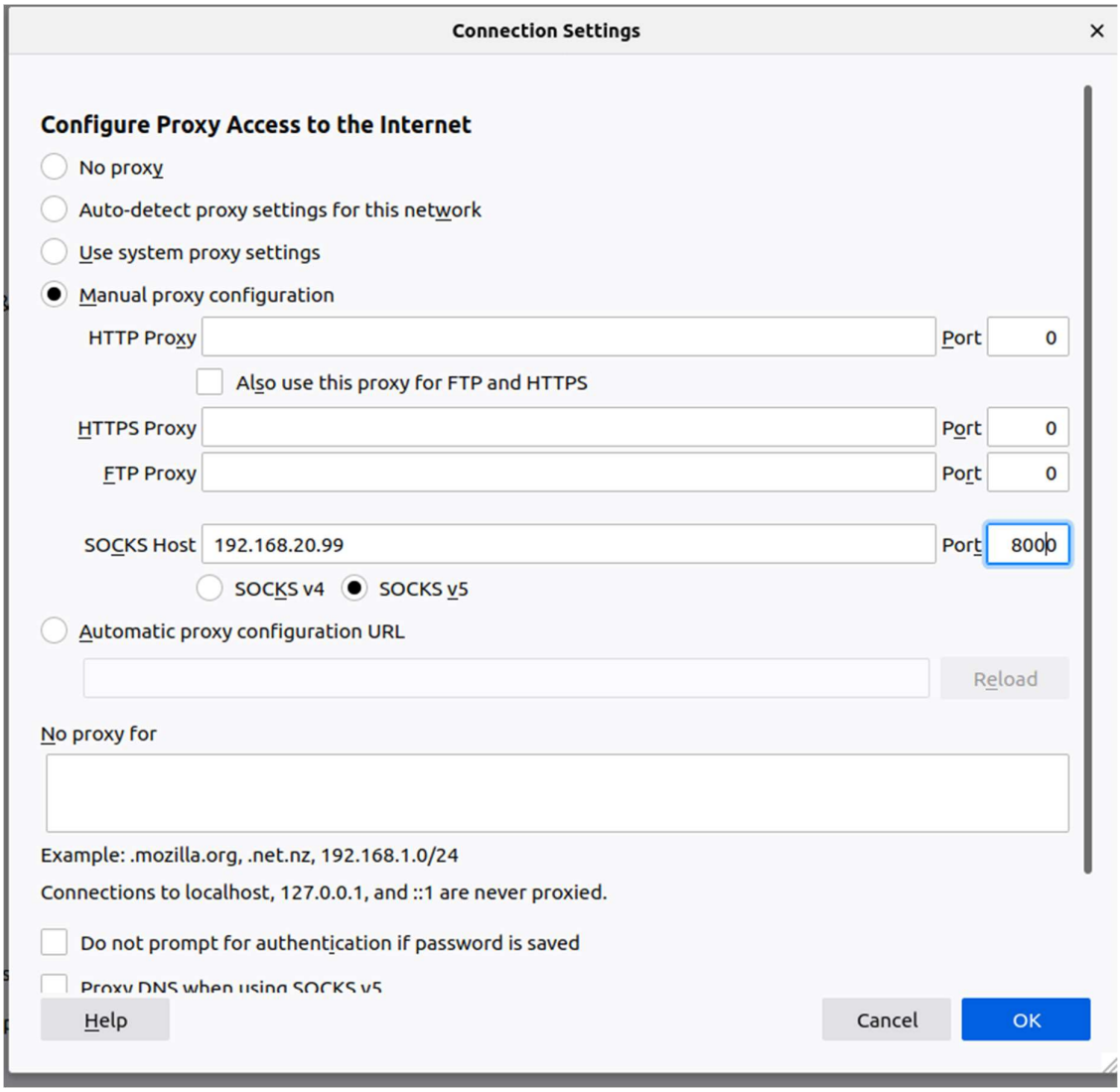
(1) Which computer establishes the actual connection with the intended web server?

- The **remote SSH server** (the SSH endpoint at `10.8.0.99` in your setup) makes the actual outbound TCP connection to the intended web server. The SOCKS proxy on B instructs the SSH endpoint which host/port to contact, and the SSH server performs the connect on behalf of the client.

(2) How does this computer know which server it should connect to?

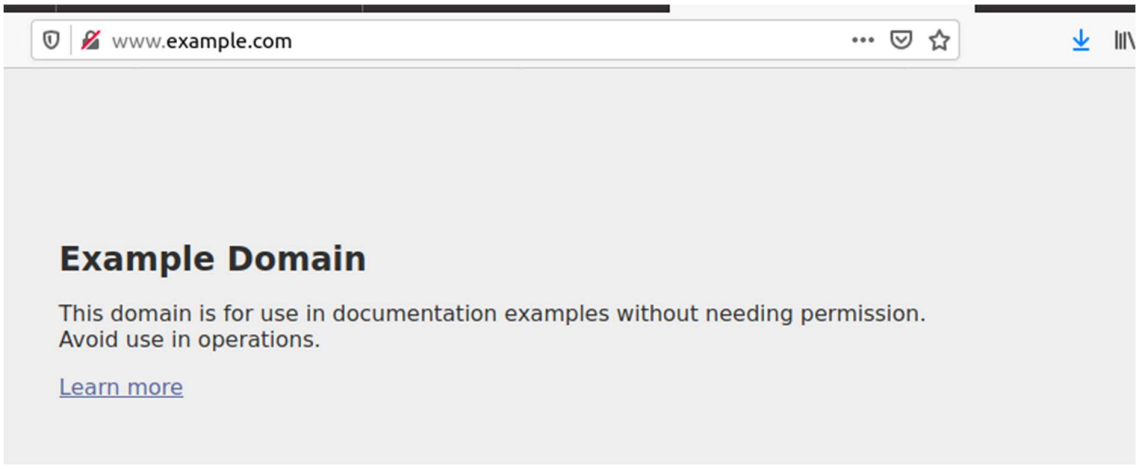
- The SOCKS protocol (used by the local proxy) encodes the target hostname/IP and port in the SOCKS request sent from the client to the SOCKS server. The SSH client translates those SOCKS requests and forwards them over the encrypted SSH channel to the remote SSH server, which reads the requested destination and opens the corresponding TCP connection to that destination. Using `socks5h` ensures DNS resolution is done on the remote side (the SSH server) so hostnames are resolved there rather than locally.

Task 2.2: Testing the Tunnel Using Browser



(Proxy host = 192.168.20.99, port = 8000, SOCKS v5 selected)

- The Firefox network settings window shows manual proxy configuration pointing to `192.168.20.99:8000`.
- This setting makes Firefox forward all browser requests through the SOCKS v5 proxy running on Host B (the internal proxy).
- The proxy created earlier by `ssh -D 0.0.0.0:8000 root@10.8.0.99` is listening on this address, allowing the browser to tunnel web traffic through the SSH connection.
- SOCKS v5 is selected because it supports authentication, DNS forwarding, and full TCP tunneling.
- This configuration ensures all HTTP requests are encrypted and sent through the dynamic SSH tunnel, bypassing firewall restrictions.



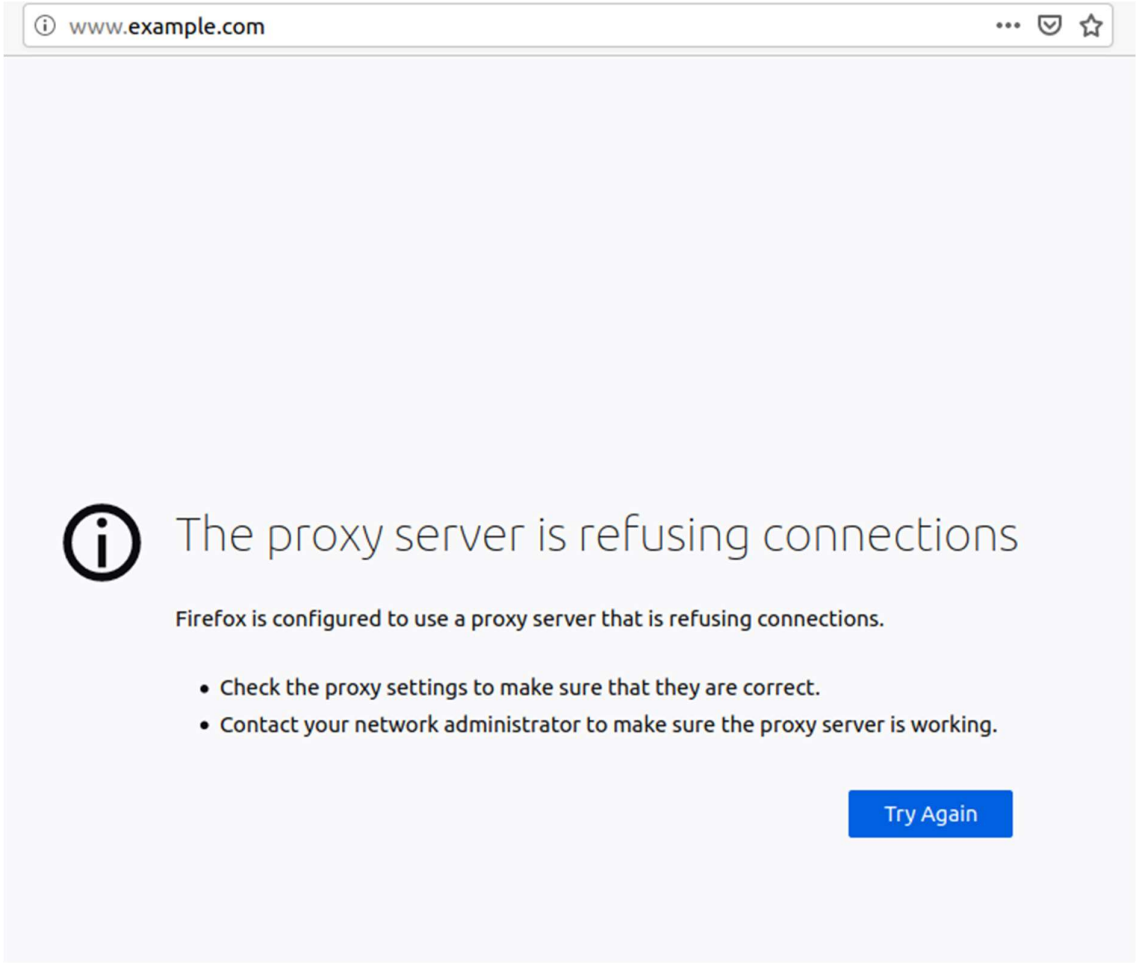
("Example Domain" page displayed)

- This confirms the SOCKS proxy is working correctly; the HTTP GET request from Firefox reached the external server (www.example.com) through the SSH tunnel.
- The response content (HTML page) proves that outbound connections blocked by the firewall were successfully redirected via the proxy.
- All traffic traveled through TCP port 22 of the SSH tunnel, not directly through standard HTTP/HTTPS ports (80/443).
- This demonstrates successful **firewall evasion using dynamic port forwarding**.
- The result verifies that any previously blocked website (like example.com or linkedin.com) can now be accessed securely through the SOCKS proxy.

```
B:PES1UG24CS840:Vinay/
$ps -eaf | grep "ssh"
root          40       1  0 07:16 ?        00:00:00 sshd: /usr/sbin/sshd [listener] 0 of 10
-100 startups
root         150       1  0 08:45 ?        00:00:00 ssh -4 -D 0.0.0.0:8000 root@10.8.0.99 -
f -N
root         153      42  0 08:59 pts/1    00:00:00 grep ssh
B:PES1UG24CS840:Vinay/
$kill 150
B:PES1UG24CS840:Vinay/
$ps -eaf | grep "ssh"
root          40       1  0 07:16 ?        00:00:00 sshd: /usr/sbin/sshd [listener] 0 of 10
-100 startups
root         150       1  0 08:45 ?        00:00:00 [ssh] <defunct>
root         155      42  0 08:59 pts/1    00:00:00 grep ssh
B:PES1UG24CS840:Vinay/
$
```

(Commands `ps -eaf | grep ssh` and `kill 150`)

- The first `ps -eaf | grep ssh` lists running SSH processes; one line corresponds to the dynamic port forwarding session (`ssh -D 0.0.0.0:8000`).
- The `kill 150` command terminates that SSH process using its PID.
- A second `ps -eaf | grep ssh` confirms the process is gone (`<defunct>` or no `ssh -D` entry).
- Stopping the SSH tunnel effectively disables the SOCKS proxy since it was bound to the SSH session.
- This step validates the cleanup procedure and helps illustrate what happens when the tunnel is broken.



(Firefox error: “The proxy server is refusing connections”) *

- After killing the SSH tunnel, Firefox can no longer reach `192.168.20.99:8000`, since the SOCKS proxy process was terminated.
- The browser’s error message confirms that the proxy port 8000 is now closed or inactive.
- This shows that the SSH tunnel was indeed handling all web traffic earlier, and without it, the blocked sites become inaccessible again.
- The failure message serves as proof that firewall evasion depended entirely on the live SSH tunnel.
- This final observation completes the verification cycle for tunnel creation, use, and shutdown.

(1) Verify Evasion of Firewall

- The successful access of www.example.com and other blocked websites via Firefox demonstrates that the dynamic SOCKS proxy bypassed the router’s egress filtering.
- The SSH connection (TCP port 22) carried all encrypted web traffic, avoiding direct HTTP/HTTPS requests blocked by firewall rules.

(2) Point Out the Tunnel Traffic

- A `tcpdump` run on the router would show only SSH traffic (Protocol TCP port 22) between Host B and 10.8.0.99.
- No clear-text HTTP packets are visible, confirming that the web requests were encapsulated within the SSH tunnel.

(3) Break the SSH Tunnel and Describe Observation

- Once the SSH process was terminated (`kill 150`), the SOCKS proxy closed, and the browser displayed "proxy server refusing connections."
- This proves that without the SSH session, the proxy cannot forward requests, and firewall restrictions are re-enforced.

Task 2.3: Writing a SOCKS Client Using Python

```
B:PES1UG24CS840:Vinay/
$python3 B-Socks-Client.py
HTTP/1.0 200 OK
Content-Type: text/html
ETag: "bc2473a18e003bdb249eba5ce893033f:1760028122.592274"
Last-Modified: Thu, 09 Oct 2025 16:42:02 GMT
Cache-Control: max-age=86000
Date: Sun, 02 Nov 2025 09:24:03 GMT
Content-Length: 513
Connection: close
X-N: S

<!doctype html><html lang="en"><head><title>Example Domain</title><meta name="viewport" co
ntent="width=device-width, initial-scale=1"><style>body{background:#eee;width:60vw;margin:
15vh auto;font-family:system-ui,sans-serif}h1{font-size:1.5em}div{opacity:0.8}a:link,a:vis
ited{color:#348}</style><body><div><h1>Example Domain</h1><p>This domain is for use in doc
umentation examples without needing permission. Avoid use in operations.<p><a href="https:
//iana.org/domains/example">Learn more</a></div></body></html>

B:PES1UG24CS840:Vinay/
$
```

- The Python program uses the `socks` module to create a SOCKS v5 socket and connect to `www.example.com`.
- It sends an HTTP GET request through the proxy (`0.0.0.0:8000`) and receives the HTML response.
- The printed HTML page confirms that the Python SOCKS client successfully routed its web request through the dynamic SSH proxy.
- This proves that even a custom program can use the SOCKS protocol to communicate via the tunnel.

.

```
 GNU nano 4.8                          B-Socks-Client.py
#!/usr/bin/env python3
import socks

# Create a SOCKS socket
s = socks.socksocket()

# Set the proxy
s.set_proxy(socks.SOCKS5, "0.0.0.0", 8000)

# Connect to the final destination via the proxy
hostname = "www.example.com"
s.connect((hostname, 80))

# Send HTTP request
request = b"GET / HTTP/1.0\r\nHost: " + hostname.encode('utf-8') + b"\r\n\r\n"
s.sendall(request)

# Get the response
response = s.recv(2048)
while response:
    print(response.decode(errors='ignore'))
    response = s.recv(2048)
```

File: `B-Socks-Client.py` (Host B)

- The script imports the `socks` library and creates a SOCKS socket (`s = socks.socksocket()`).
- The proxy is set to `0.0.0.0:8000`, meaning it connects to the SOCKS proxy running locally on host B.
- The program connects to the destination server `www.example.com` on port 80 through this proxy.
- It constructs and sends a simple HTTP GET request and prints the HTML response.
- The result confirms that the SSH tunnel on host B correctly forwards web traffic through the SOCKS v5 proxy.

```
B1:PES1UG24CS840:Vinay/
$python3 B1-B2-Socks-Client.py
HTTP/1.0 200 OK
Content-Type: text/html
ETag: "bc2473a18e003bdb249eba5ce893033f:1760028122.592274"
Last-Modified: Thu, 09 Oct 2025 16:42:02 GMT
Cache-Control: max-age=86000
Date: Sun, 02 Nov 2025 09:28:00 GMT
Content-Length: 513
Connection: close
X-N: S

<!doctype html><html lang="en"><head><title>Example Domain</title><meta name="viewport" conten
idth=device-width, initial-scale=1"><style>body{background:#eee;width:60vw;margin:15vh auto;fo
amily:system-ui,sans-serif}h1{font-size:1.5em}div{opacity:0.8}a:link,a:visited{color:#348}</st
<body><div><h1>Example Domain</h1><p>This domain is for use in documentation examples without
ing permission. Avoid use in operations.<p><a href="https://iana.org/domains/example">Learn mo
a></div></body></html>

B1:PES1UG24CS840:Vinay/
$python3 B1-B2-Socks-Client.py
```

- On B1, the proxy address is set to `192.168.20.99:8000`, which is B's IP.
- The same SOCKS client script connects through B's proxy to fetch `www.example.com`.
- The HTTP 200 OK response and HTML output confirm that B1's traffic successfully traversed the dynamic SSH tunnel via B.
- This demonstrates that other internal hosts can also use B as a gateway for encrypted tunneling to external sites.

```
  GNU nano 4.8                          B1-B2-Socks-Client.py
#!/usr/bin/env python3
import socks

# Create a SOCKS socket
s = socks.socksocket()

# Set the proxy
s.set_proxy(socks.SOCKS5, "192.168.20.99", 8000)

# Connect to the final destination via the proxy
hostname = "www.example.com"
s.connect((hostname, 80))

# Send an HTTP GET request
request = b"GET / HTTP/1.0\r\nHost: " + hostname.encode('utf-8') + b"\r\n\r\n"
s.sendall(request)

# Get the response
response = s.recv(2048)
while response:
    print(response.decode(errors='ignore'))  # Properly print HTML text
    response = s.recv(2048)
```

File: `B1-B2-Socks-Client.py` (Hosts B1 and B2)

- The proxy IP is changed to **`192.168.20.99:8000`**, which is host B's internal address.
- This allows B1 and B2 to use the SOCKS v5 proxy service hosted on B.
- The client again sends an HTTP GET request to `www.example.com`.
- The HTML response demonstrates that requests from B1 and B2 were tunneled through B's SSH connection to the external server.
- This confirms that dynamic port forwarding enables multiple internal hosts to use one SSH proxy for encrypted communication.

```
B:PES1UG24CS840:Vinay/
$ps -eaf | grep "ssh"
root          40       1  0 07:16 ?        00:00:00 [sshd] <defunct>
root         150       1  0 08:45 ?        00:00:00 [ssh] <defunct>
root         157       1  0 09:09 ?        00:00:00 ssh -4 -D 0.0.0.0:8000 root@10.
root         169       1  0 09:14 ?        00:00:00 ssh -4 -D 0.0.0.0:8000 root@10.
root         193     160  0 09:32 pts/2    00:00:00 grep ssh
```

- The command `ps -eaf | grep ssh` lists two active SSH processes (`ssh -4 -D 0.0.0.0:8000…`).
- These represent background SSH sessions maintaining the SOCKS proxy.
- This step confirms that the SOCKS service is active and listening on port 8000 for client connections.

```
B:PES1UG24CS840:Vinay/
$kill 157
B:PES1UG24CS840:Vinay/
$python3 B-Socks-Client.py
Traceback (most recent call last):
  File "/usr/local/lib/python3.8/dist-packages/socks.py", line 787, in connect
    super(socksocket, self).connect(proxy_addr)
ConnectionRefusedError: [Errno 111] Connection refused

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "B-Socks-Client.py", line 12, in <module>
    s.connect((hostname, 80))
  File "/usr/local/lib/python3.8/dist-packages/socks.py", line 47, in wrapper
    return function(*args, **kwargs)
  File "/usr/local/lib/python3.8/dist-packages/socks.py", line 800, in connect
    raise ProxyConnectionError(msg, error)
socks.ProxyConnectionError: Error connecting to SOCKS5 proxy 0.0.0.0:8000: [Errno 111] Conn
refused
B:PES1UG24CS840:Vinay/
$
```

- The command `kill 157` terminates the corresponding SSH proxy process.
- After termination, running `python3 B-Socks-Client.py` throws `ConnectionRefusedError`, showing that the SOCKS proxy is no longer available.
- This verifies that without the SSH tunnel, the proxy port (8000) is closed, and clients can no longer connect.

```
B1:PES1UG24CS840:Vinay/
$python3 B1-B2-Socks-Client.py
Traceback (most recent call last):
  File "/usr/local/lib/python3.8/dist-packages/socks.py", line 787, in connect
    super(socksocket, self).connect(proxy_addr)
ConnectionRefusedError: [Errno 111] Connection refused

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "B1-B2-Socks-Client.py", line 12, in <module>
    s.connect((hostname, 80))
  File "/usr/local/lib/python3.8/dist-packages/socks.py", line 47, in wrapper
    return function(*args, **kwargs)
  File "/usr/local/lib/python3.8/dist-packages/socks.py", line 800, in connect
    raise ProxyConnectionError(msg, error)
socks.ProxyConnectionError: Error connecting to SOCKS5 proxy 192.168.20.99:8000: [
ction refused
B1:PES1UG24CS840:Vinay/
$
```

- Similarly, `python3 B1-B2-Socks-Client.py` on B1 fails with `ProxyConnectionError: Error connecting to SOCKS5 proxy 192.168.20.99:8000 [Errno 111] Connection refused`.
- The error occurs because B's tunnel was killed, cutting off access for B1 as well.
- This behavior proves the dependence of B1 and B2 on B's SSH tunnel for reaching external web servers.

Task 3: Comparing SOCKS5 Proxy and VPN

- In the previous tasks, we implemented **dynamic port forwarding** using SSH to create a **SOCKS5 proxy** and verified that hosts **B, B1, and B2** could access blocked websites through it.

- Both SOCKS5 proxy and VPN achieve similar goals — tunneling traffic through a secure channel to bypass firewalls — but they differ in scope, configuration, and level of protection.

Comparison Summary

Aspect	SOCKS5 Proxy (Dynamic Port Forwarding)	VPN (Virtual Private Network)
Layer of Operation	Application Layer (Layer 5–7)	Network Layer (Layer 3)
Encryption	Only traffic through the SSH tunnel is encrypted	Encrypts all network traffic
Configuration Scope	Must be set manually per application (e.g., curl, browser)	System-wide; all traffic routed automatically
Performance	Lightweight, faster, minimal overhead	Slightly slower due to full encryption
IP Masking	Hides IP for specific applications using the proxy	Hides IP for the entire system
DNS Protection	Only if the application sends DNS queries through the proxy	DNS requests are encrypted and tunneled
Typical Use Case	Bypassing egress filters, accessing specific blocked sites	Secure remote access, privacy, and total traffic encryption

---

Pros and Cons

SOCKS5 Proxy

- Simple to configure using SSH (`ssh -D`), no extra software required.
- Lightweight and efficient for selective traffic (e.g., curl or browser).
- Works only for applications supporting SOCKS5 protocol.
- Does not encrypt all system traffic — limited protection.

VPN

- Provides complete encryption for all network traffic.
- Ideal for privacy, anonymity, and corporate secure access.
- Requires additional software setup and administrative privileges.
- Adds more latency and encryption overhead.

Conclusion

From our experiments:

- The **SOCKS5 proxy** established through SSH (`ssh -D 0.0.0.0:8000`) allowed **B, B1, and B2** to securely access blocked sites such as *www.example.com* by tunneling traffic through host **A**.
- This demonstrates that SOCKS5 effectively bypasses outbound firewall restrictions for specific applications.
- However, unlike a **VPN**, it does **not** encrypt or tunnel all network communication by default.
- A **VPN** offers complete device-level encryption and anonymity but at the cost of more setup complexity and performance overhead.

Hence, **SOCKS5 is better for lightweight, application-specific tunneling**, whereas **VPNs are better for full-system secure communication and privacy**.