Deep Dive: NAT, Port Forwarding & Tunneling

Understanding Network Access Control and Security

Table of Contents

---

1. Network Address Translation (NAT) - Complete Deep Dive

What is NAT and Why Does It Exist?

NAT was created to solve the **IPv4 address exhaustion problem** and provide **network security**. With only ~4.3 billion IPv4 addresses available globally, NAT allows multiple devices to share a single public IP address.

The IPv4 Exhaustion Problem

```
Total IPv4 addresses: 2^32 = 4,294,967,296
Reserved addresses:   ~588 million (private ranges, multicast, etc.)
Usable addresses:     ~3.7 billion
Global internet users: 5+ billion devices

Solution: NAT allows 1 public IP → thousands of private devices
```

Types of NAT

1. Static NAT (One-to-One Mapping)

```
Public IP: 203.0.113.10  ←→  Private IP: 192.168.1.100
Public IP: 203.0.113.11  ←→  Private IP: 192.168.1.101
```

Use Cases:

- Servers requiring consistent external access
- DMZ (Demilitarized Zone) servers
- Legacy applications expecting specific IPs

2. Dynamic NAT (Pool-based Mapping)

```
Public IP Pool: 203.0.113.10-20 (11 IPs)
Private Network: 192.168.1.0/24 (254 devices)

First-come-first-served allocation:
Device 192.168.1.50 → Gets 203.0.113.10
Device 192.168.1.75 → Gets 203.0.113.11
...when pool exhausted → connection denied
```

3. PAT (Port Address Translation) / NAT Overload

This is what **home routers use** - the most common NAT type:

```
Single Public IP: 203.0.113.5
Multiple Private Devices sharing it via different ports

Internal Request      NAT Translation Table      External View
192.168.1.10:3000  ←→  203.0.113.5:50001  ←→  Server:80
192.168.1.15:4000  ←→  203.0.113.5:50002  ←→  Server:443
192.168.1.20:5000  ←→  203.0.113.5:50003  ←→  Server:8080
```

How NAT Works - Packet Level Analysis

Outbound Connection (Internal → Internet)

```
Step 1: Device sends packet
Source: 192.168.1.100:3000
Dest: 8.8.8.8:53 (Google DNS)

Step 2: Router receives packet
- Checks NAT table for existing mapping
- Creates new mapping if none exists
- Replaces source IP and port

Step 3: Router forwards modified packet
Source: 203.0.113.5:50001  (router's public IP:random port)
Dest: 8.8.8.8:53

Step 4: NAT table entry created
Internal: 192.168.1.100:3000 ←→ External: 203.0.113.5:50001
State: ESTABLISHED
Timeout: 300 seconds
```

Inbound Response (Internet → Internal)

```
Step 1: Server responds
Source: 8.8.8.8:53
Dest: 203.0.113.5:50001

Step 2: Router receives response
- Looks up NAT table for port 50001
- Finds mapping to 192.168.1.100:3000
- Replaces destination IP and port

Step 3: Router forwards to internal device
Source: 8.8.8.8:53
Dest: 192.168.1.100:3000  (restored original)
```

Detailed NAT Table Structure

```
NAT Translation Table:
+---------------+---------------+-----------------+---------+---------+
| Internal IP   | Internal Port | External Port   | Protocol | Timeout |
+---------------+---------------+-----------------+---------+---------+
| 192.168.1.100 | 3000          | 50001           | TCP     | 7200s   |
| 192.168.1.100 | 3001          | 50002           | TCP     | 300s    |
| 192.168.1.150 | 80            | 50003           | TCP     | 7200s   |
| 192.168.1.200 | 53            | 50004           | UDP     | 60s     |
+---------------+---------------+-----------------+---------+---------+

State Tracking:
- NEW: Connection initiation
- ESTABLISHED: Active bidirectional communication
- RELATED: Related to existing connection (FTP data channel)
- INVALID: Packet doesn't match any known connection
```

NAT Behavior Analysis

Connection States and Timeouts

```python
```

```
# Different protocols have different timeout behaviors
TCP_TIMEOUTS = {
    'SYN_SENT': 120,      # Initial connection attempt
    'ESTABLISHED': 7200,  # Active connection (2 hours)
    'FIN_WAIT': 120,      # Connection closing
    'CLOSE_WAIT': 60,     # Waiting for final close
}

UDP_TIMEOUTS = {
    'NEW': 30,            # First UDP packet
    'ESTABLISHED': 180,   # Bidirectional UDP flow
}

ICMP_TIMEOUTS = {
    'ECHO_REQUEST': 30,   # Ping requests
}
```

Why External Connections Fail by Default

The Fundamental Problem:

```
External Request Arrives:
Source: 203.0.113.100:12345
Dest: 203.0.113.5:3000  (your router's public IP)

Router NAT Table Lookup:
- Searches for entry with external port 3000
- No entry found (no internal device initiated connection to port 3000)
- Result: Packet DROPPED (no forwarding rule)

Error: "Connection refused" or timeout
```

---

2. Why Routers Block External Access by Default

Security by Design - The Firewall Principle

Home routers implement **default deny** security policy:

1. Stateful Connection Tracking

```
Allowed Traffic:
✓ Outbound connections (internal → external)
✓ Return traffic for established connections
✗ Inbound connections (external → internal)
✗ Unsolicited external traffic
```

2. No Listening Services by Default

```bash
# What a typical home router blocks:
netstat -ln | grep LISTEN
# Nothing listening on public interface by default

# Internal services only:
192.168.1.1:80    # Router admin interface
192.168.1.1:53    # Internal DNS server
127.0.0.1:22      # SSH (if enabled, localhost only)
```

The Network Security Model

Defense in Depth Strategy

```
Internet
   ↓
[ISP Firewall] ← First line of defense
   ↓
[Router/NAT] ← Second line (your home router)
   ↓
[Host Firewall] ← Third line (your computer)
   ↓
[Application Security] ← Final line
```

Default Security Posture

```python
```

```python
# Router's default ruleset (conceptual)
class RouterFirewall:
    def __init__(self):
        self.default_policy = "DROP"
        self.rules = [
            # Allow established connections
            {"state": "ESTABLISHED,RELATED", "action": "ACCEPT"},
            # Allow outbound from internal network
            {"src": "192.168.1.0/24", "direction": "OUT", "action": "ACCEPT"},
            # Allow internal network communication
            {"src": "192.168.1.0/24", "dst": "192.168.1.0/24", "action": "ACCEPT"},
            # Drop everything else
            {"action": "DROP"}
        ]

    def process_packet(self, packet):
        if packet.direction == "INBOUND" and packet.state == "NEW":
            return "DROP"  # This is why external access fails!
```

Technical Reasons for Access Denial

1. No NAT Mapping Exists

```
Problem: External request to 203.0.113.5:3000
Router Logic:
1. Check NAT table for port 3000 mapping
2. No entry found (internal device never connected outbound on port 3000)
3. No destination to forward to
4. Drop packet
```

2. Stateful Firewall Rules

```
Connection State Analysis:
External → Internal request = "NEW" connection
Router policy: Only allow "ESTABLISHED" and "RELATED"
Result: Connection blocked
```

3. Port Scanning Protection

```bash
```

```
# What attackers do:
nmap -p 1-65535 your-public-ip

# What router protects against:
for port in 1..65535:
    if not port_forwarding_rule_exists(port):
        drop_packet()  # Invisible to scanner
```

Real-World Example: API Access Attempt

Scenario Setup

```
Your Setup:
- Router Public IP: 203.0.113.5
- Your PC Private IP: 192.168.1.100
- API Server running on: localhost:3000

External User Attempts:
curl http://203.0.113.5:3000/api/data
```

What Actually Happens (Packet Trace)

```
1. External client sends SYN packet:
   Source: 203.0.113.100:45678
   Dest: 203.0.113.5:3000
   Flags: SYN

2. Router receives packet on WAN interface:
   - Checks NAT table for port 3000
   - No existing mapping found
   - Checks port forwarding rules
   - No rule for port 3000
   - Firewall rule: DROP

3. Router action:
   - Silently drops packet (no response sent)
   - Logs: "Blocked inbound connection to port 3000"

4. Client experience:
   - Connection timeout (no response)
   - Error: "Connection timed out" or "No route to host"
```

Network Capture Analysis

```bash
# On router (if accessible):
tcpdump -i eth0 port 3000
# Shows: Incoming SYN packets, no outgoing responses

# On your PC:
netstat -an | grep 3000
# Shows: 127.0.0.1:3000 LISTEN (only localhost)
# Missing: 0.0.0.0:3000 LISTEN (would accept external)
```

---

3. Port Forwarding - Technical Deep Dive

What Port Forwarding Actually Does

Port forwarding creates **permanent NAT mappings** that allow external access:

Creating the NAT Exception

```
Normal NAT: Dynamic mapping created by outbound connection
Port Forward: Static mapping created by administrator

Configuration:
External Port: 8080
Internal IP: 192.168.1.100
Internal Port: 3000

Result:
NAT table gets permanent entry:
192.168.1.100:3000 ←→ 203.0.113.5:8080 (STATIC)
```

Types of Port Forwarding

1. Simple Port Forwarding (Port Translation)

```
Router Configuration:
External Port: 8080 → Internal: 192.168.1.100:3000

Traffic Flow:
Internet request to 203.0.113.5:8080
↓
Router translates to 192.168.1.100:3000
↓
Your API server receives request
```

2. Port Range Forwarding

```
External Ports: 8080-8090 → Internal: 192.168.1.100:3000-3010

Use Case: Multiple services or load balancing
Port 8080 → Service on port 3000
Port 8081 → Service on port 3001
...
Port 8090 → Service on port 3010
```

3. DMZ (Demilitarized Zone)

```
Configuration: DMZ Host = 192.168.1.100

Effect: ALL external traffic forwarded to one internal IP
- Essentially makes one device fully exposed
- Bypasses NAT for that device
- High security risk but maximum accessibility
```

Port Forwarding Implementation Details

Router Configuration Methods

1. Consumer Router Web Interface

```html
<!-- Typical router interface -->
<form action="/port_forwarding" method="post">
  <input name="service_name" value="API Server" />
  <input name="external_port" value="8080" />
  <input name="internal_ip" value="192.168.1.100" />
  <input name="internal_port" value="3000" />
  <select name="protocol">
    <option value="TCP">TCP</option>
    <option value="UDP">UDP</option>
    <option value="BOTH">TCP+UDP</option>
  </select>
  <input type="submit" value="Add Rule" />
</form>
```

2. Advanced Router (CLI Configuration)

```bash
```

```
# Cisco Router
configure terminal
ip nat inside source static tcp 192.168.1.100 3000 interface GigabitEthernet0/0 8080

# Mikrotik RouterOS
/ip firewall nat add chain=dstnat action=dst-nat protocol=tcp dst-port=8080 to-addresses=192.168.1.100 to-ports=30

# pfSense
nat on $ext_if inet proto tcp from any to ($ext_if) port 8080 -> 192.168.1.100 port 3000
```

3. Linux Server as Router (iptables)

```bash
# Enable IP forwarding
echo 1 > /proc/sys/net/ipv4/ip_forward

# Port forwarding rule
iptables -t nat -A PREROUTING -p tcp --dport 8080 -j DNAT --to-destination 192.168.1.100:3000

# Allow forwarded traffic
iptables -A FORWARD -p tcp -d 192.168.1.100 --dport 3000 -m state --state NEW,ESTABLISHED,RELATED -j ACCEPT

# Masquerade return traffic
iptables -t nat -A POSTROUTING -j MASQUERADE

# Save rules (Ubuntu/Debian)
iptables-save > /etc/iptables/rules.v4
```

Port Forwarding Traffic Analysis

Complete Packet Flow with Port Forwarding

Step 1: External Client Request

```
Client sends:
Source: 203.0.113.100:45678
Dest: 203.0.113.5:8080  (your router's public IP)
Data: GET /api/users HTTP/1.1
```

Step 2: Router Processing

```python
```

```python
# Router's packet processing logic
def process_inbound_packet(packet):
    # Check port forwarding rules first
    for rule in port_forward_rules:
        if packet.dest_port == rule.external_port:
            # Found matching rule
            packet.dest_ip = rule.internal_ip
            packet.dest_port = rule.internal_port

            # Create NAT table entry for return traffic
            nat_table[packet.src_ip:packet.src_port] = {
                'internal_ip': rule.internal_ip,
                'internal_port': rule.internal_port,
                'external_port': rule.external_port
            }

            return forward_to_internal(packet)

    # No port forwarding rule found
    return drop_packet(packet)
```

Step 3: Forwarded to Internal Server

```
Router forwards modified packet:
Source: 203.0.113.100:45678  (unchanged)
Dest: 192.168.1.100:3000     (translated)
Data: GET /api/users HTTP/1.1 (unchanged)
```

Step 4: Internal Server Response

```
Your API server responds:
Source: 192.168.1.100:3000
Dest: 203.0.113.100:45678
Data: HTTP/1.1 200 OK\n{"users": [...]}
```

Step 5: Router Return Processing

```
Router modifies response:
Source: 203.0.113.5:8080     (translated back)
Dest: 203.0.113.100:45678    (unchanged)
Data: HTTP/1.1 200 OK\n{"users": [...]} (unchanged)
```

Advanced Port Forwarding Scenarios

1. Multiple Services on One Server

```bash
# Web server
iptables -t nat -A PREROUTING -p tcp --dport 80 -j DNAT --to-destination 192.168.1.100:80

# API server
iptables -t nat -A PREROUTING -p tcp --dport 8080 -j DNAT --to-destination 192.168.1.100:3000

# Database (careful!)
iptables -t nat -A PREROUTING -p tcp --dport 5432 -j DNAT --to-destination 192.168.1.100:5432

# SSH access
iptables -t nat -A PREROUTING -p tcp --dport 2222 -j DNAT --to-destination 192.168.1.100:22
```

2. Load Balancing with Port Forwarding

```bash
# Round-robin load balancing
iptables -t nat -A PREROUTING -p tcp --dport 8080 -m statistic --mode random --probability 0.33 -j DNAT --to-destina
iptables -t nat -A PREROUTING -p tcp --dport 8080 -m statistic --mode random --probability 0.50 -j DNAT --to-destina
iptables -t nat -A PREROUTING -p tcp --dport 8080 -j DNAT --to-destination 192.168.1.102:3000
```

3. Protocol-Specific Forwarding

```bash
# HTTP traffic to web server
iptables -t nat -A PREROUTING -p tcp --dport 80 -j DNAT --to-destination 192.168.1.100:80

# HTTPS traffic to different server
iptables -t nat -A PREROUTING -p tcp --dport 443 -j DNAT --to-destination 192.168.1.101:443

# DNS queries to internal DNS server
iptables -t nat -A PREROUTING -p udp --dport 53 -j DNAT --to-destination 192.168.1.10:53
iptables -t nat -A PREROUTING -p tcp --dport 53 -j DNAT --to-destination 192.168.1.10:53
```

Security Implications of Port Forwarding

1. Attack Surface Expansion

```
Without Port Forwarding:
Internet → [Router Firewall] → Internal Network
Attack surface: Router's management interface only

With Port Forwarding:
Internet → [Router] → Internal Service
Attack surface: Internal service directly exposed
```

2. Common Security Mistakes

```bash
# BAD: Forwarding sensitive services
Port 22 (SSH) → Internal server
Port 3389 (RDP) → Internal server
Port 5432 (PostgreSQL) → Database server

# BETTER: Use non-standard ports + additional security
Port 2222 (SSH) → Internal server + key-based auth
Port 8443 (HTTPS) → API server + authentication
Port 5433 (PostgreSQL) → Database + SSL + restricted users
```

3. Monitoring Port Forwarded Services

```bash
# Monitor connections
netstat -an | grep :8080
ss -tlnp | grep :8080

# Log analysis
tail -f /var/log/nginx/access.log | grep "203.0.113"
journalctl -f -u myapi.service

# Intrusion detection
fail2ban-client status
fail2ban-client status myapi-jail
```

---

4. Tunneling - Complete Analysis

What is Tunneling?

Tunneling **encapsulates** one network protocol inside another, creating a secure "tunnel" through potentially insecure networks.

Basic Tunneling Concept

```
Original Communication:
[App Data] → [Direct Network] → [App Data]

Tunneled Communication:
[App Data] → [Tunnel Protocol] → [Carrier Network] → [Tunnel Protocol] → [App Data]
               ↓                                     ↓
         Encryption/Wrapping                 Decryption/Unwrapping
```

Why Tunneling Exists

1. Security in Untrusted Networks

```
Problem: Sending sensitive data over public internet
Solution: Encrypt data inside secure tunnel

Example:
Original: HTTP → Internet → HTTP (visible to anyone)
Tunneled: HTTP → HTTPS tunnel → Internet → HTTPS tunnel → HTTP (encrypted)
```

2. Network Topology Bypass

```
Problem: Firewall blocks direct access
Solution: Tunnel through allowed protocol

Example:
Blocked: Client → Firewall (blocks port 3000) → Server
Tunneled: Client → SSH tunnel (port 22 allowed) → Server
```

3. Protocol Translation

```
Problem: Legacy protocol not supported by modern network
Solution: Wrap old protocol in new one

Example:
IPv4 application → IPv6 tunnel → IPv6 network → IPv6 tunnel → IPv4 application
```

Types of Tunneling

1. Network Layer Tunneling

VPN (Virtual Private Network)

```
Site-to-Site VPN:
Office A [192.168.1.0/24] ←→ Internet ←→ Office B [192.168.2.0/24]
         ↓                          ↓
    [VPN Gateway]               [VPN Gateway]
         ↓                          ↓
         └────────── Encrypted Tunnel ──────────┘

Traffic Flow:
192.168.1.100 → VPN Gateway → Internet (encrypted) → VPN Gateway → 192.168.2.100
```

IPSec Tunnel

```python
# IPSec packet structure
class IPSecPacket:
    def __init__(self, original_packet):
        self.esp_header = ESPHeader()
        self.encrypted_payload = encrypt(original_packet)
        self.esp_trailer = ESPTrailer()
        self.authentication_data = calculate_hmac(self.encrypted_payload)

# Traffic flow
original_ip_packet = IPPacket(src="192.168.1.100", dst="192.168.2.100", data="Hello")
ipsec_packet = IPSecPacket(original_ip_packet)
# Result: Encrypted packet that appears to go from VPN gateway to VPN gateway
```

2. Transport Layer Tunneling

SSH Tunneling (Most Common for Development)

```bash
# Local port forwarding
ssh -L 8080:target-server:80 user@jump-server

Flow:
localhost:8080 → SSH tunnel → jump-server → target-server:80
```

Technical Deep Dive:

```python
```

```python
# SSH tunnel packet flow
class SSHTunnel:
    def __init__(self, local_port, remote_host, remote_port, ssh_server):
        self.local_port = local_port
        self.remote_host = remote_host
        self.remote_port = remote_port
        self.ssh_connection = establish_ssh_connection(ssh_server)

    def handle_local_connection(self, local_socket):
        # Client connects to local port 8080
        ssh_channel = self.ssh_connection.open_channel()

        # SSH server connects to target
        ssh_channel.request_port_forward(self.remote_host, self.remote_port)

        # Relay data bidirectionally
        while True:
            data_from_client = local_socket.recv(1024)
            ssh_channel.send(data_from_client)  # Encrypted over SSH

            data_from_server = ssh_channel.recv(1024)  # Decrypted from SSH
            local_socket.send(data_from_server)
```

SOCKS Proxy Tunneling

```bash
# Create SOCKS proxy
ssh -D 1080 user@proxy-server

# Application configuration
export http_proxy=socks5://localhost:1080
export https_proxy=socks5://localhost:1080
```

3. Application Layer Tunneling

HTTP Tunneling

```http
```

```
# CONNECT method for tunneling
CONNECT target-server:443 HTTP/1.1
Host: target-server:443

# Proxy establishes TCP connection to target
# Then relays raw TCP data bidirectionally
```

WebSocket Tunneling

```javascript
// Client-side WebSocket tunnel
const ws = new WebSocket('wss://tunnel-server.com/tunnel');

ws.onopen = function() {
  // Tunnel established
  sendHTTPRequest('/api/data');
};

ws.onmessage = function(event) {
  // Receive tunneled HTTP response
  handleHTTPResponse(event.data);
};
```

Modern Tunneling Solutions Deep Dive

1. ngrok - Technical Analysis

How ngrok Works

```
Your Machine        ngrok Cloud         Internet User
[API :3000] ←→ [ngrok client] ←→ [ngrok server] ←→ [User Browser]
                     ↓                    ↓
            Persistent WebSocket   Public HTTPS endpoint
            Connection (encrypted)  (abc123.ngrok.io)
```

ngrok Protocol Flow

```python
```

```python
# Simplified ngrok protocol
class NgrokTunnel:
    def __init__(self, local_port):
        self.local_port = local_port
        self.control_connection = establish_websocket("tunnel.ngrok.com")
        self.tunnel_url = self.register_tunnel()

    def register_tunnel(self):
        registration = {
            "type": "http",
            "local_port": self.local_port,
            "subdomain": "random"  # or custom
        }
        response = self.control_connection.send(registration)
        return response["public_url"]  # https://abc123.ngrok.io

    def handle_request(self, request_id, http_request):
        # ngrok server receives external HTTP request
        # Forwards to your machine via WebSocket
        response = forward_to_local(self.local_port, http_request)
        self.control_connection.send({
            "request_id": request_id,
            "response": response
        })
```

ngrok Traffic Analysis

```
1. External user visits https://abc123.ngrok.io/api/data

2. ngrok server receives request:
  - Identifies tunnel: abc123.ngrok.io → your machine
  - Generates request ID: req_123456
  - Forwards via WebSocket control connection

3. ngrok client receives via WebSocket:
  {
    "request_id": "req_123456",
    "method": "GET",
    "path": "/api/data",
    "headers": {...}
  }

4. ngrok client makes local HTTP request:
  GET http://localhost:3000/api/data

5. Your API responds:
  HTTP/1.1 200 OK
  {"users": [...]}

6. ngrok client sends response back:
  {
    "request_id": "req_123456",
    "status": 200,
    "headers": {...},
    "body": "{\"users\": [...]}"
  }

7. ngrok server forwards to external user:
  HTTP/1.1 200 OK
  {"users": [...]}
```

2. Cloudflare Tunnel (cloudflared)

Architecture Overview

```
Your Machine             Cloudflare Edge          Internet
[API :3000] ←→ [cloudflared] ←→ [CF Tunnel Service] ←→ [Global CDN] ←→ [Users]
                  ↓                  ↓                  ↓
          Outbound-only      Distributed tunnel     Custom domain
          connections        infrastructure         (api.yourdomain.com)
```

Technical Implementation

```yaml
# ~/.cloudflared/config.yml
tunnel: a1b2c3d4-e5f6-7g8h-9i0j-k1l2m3n4o5p6
credentials-file: ~/.cloudflared/tunnel-credentials.json

ingress:
  - hostname: api.yourdomain.com
    service: http://localhost:3000
    originRequest:
      httpHostHeader: api.yourdomain.com

  - hostname: admin.yourdomain.com
    service: http://localhost:3001
    originRequest:
      httpHostHeader: admin.yourdomain.com

  # Catch-all rule (required)
  - service: http_status:404
```

Cloudflare Tunnel Protocol

```python
```

```
# Simplified cloudflared operation
class CloudflareTunnel:
    def __init__(self, tunnel_id, credentials):
        self.tunnel_id = tunnel_id
        self.credentials = credentials
        self.edge_connections = []

    def start(self):
        # Connect to multiple Cloudflare edge servers
        for i in range(4):  # Typically 4 connections for redundancy
            edge_conn = self.connect_to_edge()
            self.edge_connections.append(edge_conn)

    def connect_to_edge(self):
        edge_server = select_closest_edge()
        connection = establish_quic_connection(edge_server, self.credentials)

        # Register tunnel and ingress rules
        connection.send({
            "tunnel_id": self.tunnel_id,
            "ingress": self.load_ingress_rules()
        })

        return connection

    def handle_request(self, edge_request):
        # Cloudflare edge forwards request via QUIC
        local_response = forward_to_local_service(edge_request)
        return local_response
```

3. SSH Reverse Tunneling for Self-Hosted Solutions

Setting Up Reverse Tunnel

```bash
# On your local machine with API
ssh -R 8080:localhost:3000 user@your-vps.com

# Creates tunnel:
# VPS:8080 → (SSH tunnel) → Your machine:3000
```

Persistent Reverse Tunnel Setup

```bash
```

```
# ~/.ssh/config
Host tunnel-server
  HostName your-vps.com
  User tunnel
  Port 22
  ServerAliveInterval 30
  ServerAliveCountMax 3
  ExitOnForwardFailure yes
  RemoteForward 8080 localhost:3000

# Systemd service for persistence
# /etc/systemd/system/api-tunnel.service
[Unit]
Description=SSH Tunnel for API
After=network-online.target
Wants=network-online.target

[Service]
Type=simple
User=your-user
ExecStart=/usr/bin/ssh -N tunnel-server
Restart=always
RestartSec=5

[Install]
WantedBy=multi-user.target
```

VPS Nginx Configuration

```nginx
# /etc/nginx/sites-available/api-tunnel
server {
  listen 80;
  server_name api.yourdomain.com;

  # Redirect HTTP to HTTPS
  return 301 https://$server_name$request_uri;
}

server {
  listen 443 ssl http2;
```