

Homework 4 - Information Security (ICS344)

Alfaifi, Ammar – 201855360

December 9, 2023

1 Firewalls and Connection Tracking

My machine is iMac M1, and I don't have easy access to Virtualbox capabilities, so I'll use three real machines for this setup.

1. iMac with macOS as Node A.
2. Laptop with Arch Linux as Router.
3. Raspberry Pi as Node B.

Now to setup Arch Linux as router I follow theses steps:

1. Assign the IP address 192.168.60.11 to the network interface card (NIC) named `enp7s0u1u2i5` and optionally enable a DHCP server to assign an IP address when a new device connects. This is by adding the following configs to file `/etc/systemd/network/20-dhcpd-wired.network`:

Listing 1: Assign IP addresses to interfaces

```
[Match]
Name=enp7s0u1u2i5

[Network]
Address=192.168.60.11/24
DHCPServer=true
IPMasquerade=ipv4

[DHCPServer]
PoolOffset=100
PoolSize=20
EmitDNS=yes
DNS=9.9.9.9
```

- **DHCPServer** Enables the DHCP server on the specified network interface. This means the machine will be providing dynamic IP addresses to other devices on the network.
 - **IPMasquerade** Configures IP masquerading for IPv4. This is typically used in network address translation (NAT) scenarios, allowing devices with private IP addresses to access the internet using the public IP address of the machine running the masquerading.
2. Doing the same for other NIC, but only change `Address` config to `10.1.1.0/24`, and `Name=wlp2s0`. And write it to another file in folder `/etc/systemd/network/`
 3. to allow Arch OS to forward packets as needed we enable that by, running the following:

Listing 2: Enable port forwarding

```
sudo sysctl net.ipv4.ip_forward=1
sudo sysctl net.ipv4.conf.all.forwarding=1
sudo sysctl net.ipv6.conf.all.forwarding=1
```

4. Lastly we reload configs by running `sudo networkctl reload`.

1.1 Results

- See Figure 1 for the Router interfaces settings after setting up the configs above.
- See Figure 2, when Router is being ping from Node A.
- See Figure 3, when Node B is being ping from Node A, and Figure 4 for wireshark logs.
- See Figure 5 where after configuring Router not to forward packets from Node B to Node A.

```
~ ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host noprefixroute
        valid_lft forever preferred_lft forever
2: enp7s0u1u2i5: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
    link/ether 00:1f:b5:2c:08:68 brd ff:ff:ff:ff:ff:ff
    inet6 fe80::6d93:5391:cd82:f5f6/64 scope link noprefixroute
        valid_lft forever preferred_lft forever
3: wlp2s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
    link/ether f4:0f:24:1a:13:f1 brd ff:ff:ff:ff:ff:ff
    inet 192.168.0.109/24 brd 192.168.0.255 scope global dynamic noprefixroute wlp2s0
        valid_lft 7052sec preferred_lft 7052sec
    inet6 fe80::5f81:ec48:2b60:3fdc/64 scope link noprefixroute
        valid_lft forever preferred_lft forever
```

Figure 1: IP and MAC addresses for the of the Router

```
~ ping 192.168.60.11
PING 192.168.60.11 (192.168.60.11): 56 data bytes
64 bytes from 192.168.60.11: icmp_seq=0 ttl=64 time=2.340 ms
64 bytes from 192.168.60.11: icmp_seq=1 ttl=64 time=2.478 ms
^C
--- 192.168.60.11 ping statistics ---
2 packets transmitted, 2 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 2.340/2.409/2.478/0.069 ms
```

Figure 2: Router is ping-ed from Node A

2 TUN Interface

See Figure 6 for how to create TUN in Router as well as the all the links active on Router including my TUN. In Figure 7 I assigned the IP address 192.168.60.60/24 to my TUN. Finally Figure 8 shows `alfaifi-tun0` with IP address 192.168.60.60 is reachable from Node A.

3 SQL Injection Attack

In Figure 9, I created the table in a PostgreSQL database. And in Figure 10 I populated it with some fake data.

Heres the code to create a simple python server to execute SQL query that has threat of SQL injection attack:

```
→ ~ ping 192.168.0.103
PING 192.168.0.103 (192.168.0.103): 56 data bytes
64 bytes from 192.168.0.103: icmp_seq=0 ttl=63 time=6.899 ms
64 bytes from 192.168.0.103: icmp_seq=1 ttl=63 time=3.696 ms
64 bytes from 192.168.0.103: icmp_seq=2 ttl=63 time=3.526 ms
^C
--- 192.168.0.103 ping statistics ---
3 packets transmitted, 3 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 3.526/4.707/6.899/1.552 ms
```

Figure 3: Ping Node B from Node A.

No.	Time	Source	Destination	Protocol	Length	Info
21	3.925895	192.168.60.113	192.168.0.103	ICMP	98	Echo (ping) request
22	3.929368	192.168.0.103	192.168.60.113	ICMP	98	Echo (ping) reply
42	4.931165	192.168.60.113	192.168.0.103	ICMP	98	Echo (ping) request
43	4.934871	192.168.0.103	192.168.60.113	ICMP	98	Echo (ping) reply
49	5.936364	192.168.60.113	192.168.0.103	ICMP	98	Echo (ping) request
50	5.939493	192.168.0.103	192.168.60.113	ICMP	98	Echo (ping) reply
51	6.938423	192.168.60.113	192.168.0.103	ICMP	98	Echo (ping) request
52	6.941726	192.168.0.103	192.168.60.113	ICMP	98	Echo (ping) reply
55	7.943533	192.168.60.113	192.168.0.103	ICMP	98	Echo (ping) request
56	7.946844	192.168.0.103	192.168.60.113	ICMP	98	Echo (ping) reply

Figure 4: Wireshark catching ICMP packets when ping Node B from Node A.

```
→ ~ ping 192.168.60.113
PING 192.168.60.113 (192.168.60.113): 56 data bytes
Request timeout for icmp_seq 0
Request timeout for icmp_seq 1
Request timeout for icmp_seq 2
^C
--- 192.168.60.113 ping statistics ---
4 packets transmitted, 0 packets received, 100.0% packet loss
→ ~
```

Figure 5: Node A is not reachable from Node B.

```
→ ~ sudo ip tuntap add name alfaifi-tun0 mode tun
→ ~ ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: enp7s0u1u2i5: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500
    link/ether 00:1f:b5:2c:08:68 brd ff:ff:ff:ff:ff:ff
3: wlp2s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
    link/ether f4:0f:24:1a:13:f1 brd ff:ff:ff:ff:ff:ff
4: alfaifi-tun0: <POINTOPOINT,MULTICAST,NOARP> mtu 1500 qdisc
    link/none
```

Figure 6: Shows command to create a TUN and the shows the all links on Router.

Listing 3: Login server connected to DB

```
from flask import Flask, render_template, request
import psycopg2

app = Flask(__name__)

# PostgreSQL database configuration
db_config = {
    "dbname": "postgres",
    "user": "ammar-imac",
    "password": "",
    "host": "localhost",
    "port": "5432",
}

def execute_query(query, fetchall=True):
    try:
        connection = psycopg2.connect(**db_config)
        cursor = connection.cursor()
        cursor.execute(query)
```

```
→ ~ sudo ip addr add 192.168.60.60/24 dev alfaifi-tun0
→ ~ ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mod
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: enp7s0u1u2i5: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_co
    link/ether 00:1f:b5:2c:08:68 brd ff:ff:ff:ff:ff:ff
3: wlp2s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel st
    link/ether f4:0f:24:1a:13:f1 brd ff:ff:ff:ff:ff:ff
4: alfaifi-tun0: <NO-CARRIER,POINTOPOINT,MULTICAST,NOARP,UP> mtu 1500 q
    link/none
→ ~ sudo ip addr add 192.168.60.60/24 dev alfaifi-tun0
Error: ipv4: Address already assigned.
```

Figure 7: Assign an IP address to my TUN.

```
→ ~ ping 192.168.60.60
PING 192.168.60.60 (192.168.60.60): 56 data bytes
64 bytes from 192.168.60.60: icmp_seq=0 ttl=64 time=4.460 ms
64 bytes from 192.168.60.60: icmp_seq=1 ttl=64 time=2.448 ms
64 bytes from 192.168.60.60: icmp_seq=2 ttl=64 time=2.452 ms
64 bytes from 192.168.60.60: icmp_seq=3 ttl=64 time=2.410 ms
64 bytes from 192.168.60.60: icmp_seq=4 ttl=64 time=2.551 ms
^C
--- 192.168.60.60 ping statistics ---
5 packets transmitted, 5 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 2.410/2.864/4.460/0.799 ms
→ ~ who
ammar-imac          console          Dec  5 21:44
```

Figure 8: Ping my TUN named alfaifi-tun from Node A.

```

        result = cursor.fetchall() if fetchall else cursor.fetchone()
        connection.commit()
        cursor.close()
        connection.close()

    return result
except Exception as e:
    return str(e)

@app.route("/")
def index():
    return render_template("login.html")

@app.route("/login", methods=["POST"])
def login():
    username = request.form.get("username")
    password = request.form.get("password")

    if not username or not password:
        return render_template(
            "login.html", error="Username and password are required."
        )

    query = (
        f"SELECT * FROM Employee WHERE Email='{username}' AND Password='{\n\
        ↪ password}';"
    )
    result = execute_query(query, fetchall=False)

    print(result)
    if result:
        return f"Welcome, {username}!"
    else:
        return render_template(
            "login.html", error="Invalid credentials. Please try again."
        )

if __name__ == "__main__":
    app.run(debug=True)

```

3.1 Performing Attack

Figure 11 shows an example of successful login. Now to trick DB to return the entire table we'll use the fact OR ''='' is always TRUE. See Figure 12, how we enter this trick in username input and password. Then from our code the server will substitute these values as a string into the SQL query. It'll be like this before execute:

```
SELECT * FROM Employee WHERE Email='' or ''='' AND Password='' or ''='';
```

Meaning every row in the table is returned and the attack is carried on, as seen in Figure 13 the returned employees' names and password.

Now I can use one of employees breached data to login and the result as in Figure 11.

```
→ ~ sudo ip tuntap add name alfaifi-tun0 mode tun
→ ~ ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: enp7s0u1u2i5: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500
    link/ether 00:1f:b5:2c:08:68 brd ff:ff:ff:ff:ff:ff
3: wlp2s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
    link/ether f4:0f:24:1a:13:f1 brd ff:ff:ff:ff:ff:ff
4: alfaifi-tun0: <POINTOPOINT,MULTICAST,NOARP> mtu 1500 qdisc
    link/none
```

Figure 9: Create table Empolyee.

```
postgres=# INSERT INTO Employee (EmployeeID, Name, Password, Salary, Birthday, SSN, Nickname, Email, Address, Phone)
VALUES
(1, 'John Doe', 'password123', 60000.00, '1990-05-15', '123-45-6789', 'JD', 'john.doe@email.com', '123 Main St, Cityville', '555-1234'),
(2, 'Jane Smith', 'securepass456', 75000.00, '1985-08-22', '987-65-4321', 'JS', 'jane.smith@email.com', '456 Oak St, Townsville', '555-5678'),
(3, 'Bob Johnson', 'secretword789', 50000.00, '1995-02-10', '543-21-8765', 'BJ', 'bob.johnson@email.com', '789 Pine St, Villageto
wn', '555-9012');
INSERT 0 3
postgres=#
```

Figure 10: Populate table with some data.

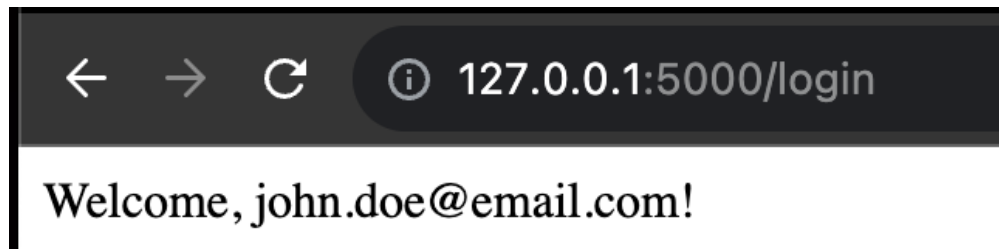
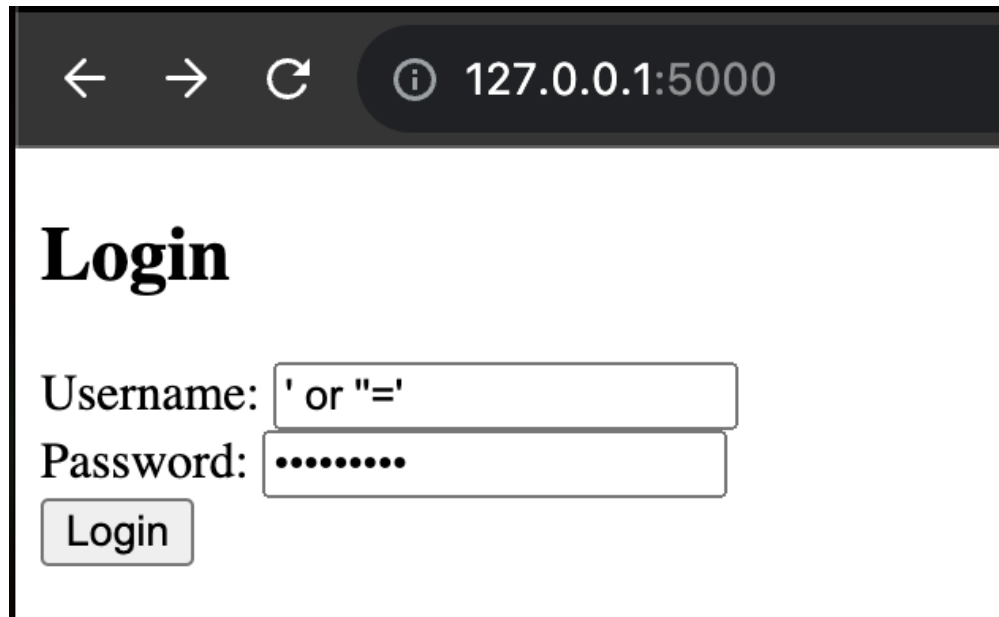


Figure 11: A successful login with credentials matched from db.



A screenshot of a web browser window. The address bar shows the URL `127.0.0.1:5000`. The page title is "Login". Below the title, there are two input fields. The "Username:" field contains the text `' or "='`. The "Password:" field contains a series of dots. Below the password field is a "Login" button.

Figure 12: Instead of real user data, here's we try to do an SQL injection



A screenshot of a web browser window. The address bar shows the URL `127.0.0.1:5000/login`. The page content displays the message: `Welcome, [('John Doe', 'password123'), ('Jane Smith', 'securepass456'), ('Bob Johnson', 'secretword789')]`!

Figure 13: After a successful attack, db returns matches the entire `Employee` table.