Practical Network Defense

First Assignment - University "La Sapienza"

Group 27: Nicola Bartoloni ******* - Valerio Trenta 1856471

2020-13-04

1 Scope of the assignment

1.1 Our network

The target network belongs to the fictitious **ACME co.**, which requires a series of tools and security checks to be implemented in the hosts of the network itself.
All the hosts can be reached through a **VPN**. Though the network is quite large and comprehends a number of more than ten hosts, in this paper we will focus only on a specific subnet, that is the one where the security measures are to be implemented according to the scope of this assignment.
Thus, from now on we will not refer to the topology of the whole network, but only on the topology of the **Clients network**.

1.2 Our scope

The aforementioned **Clients network** is pictured in **Figure 1**, and is assigned the corresponding IP address **100.100.2.0/24**.

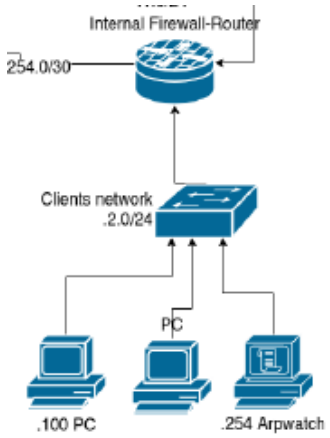

Figure 1: The **Clients network**, scope of the assignment.

The scope of the assignment is to enable the **DHCP service** on the *Internal Firewall-Router*, in order to provide the clients of the target network with dynamic IPv4 addresses. Furthermore, the clients should be protected against **link-local attacks** by implementing security measures and tools on the **Arpwatch** machine with IP address **100.100.2.254**.

1.3 Goals

How do we plan to achieve the goals of this assignment? General ideas.

2 DHCP Setup

This section describes how the **DHCP service** was set up in the *Internal router* machine, the one reachable at address **100.100.2.1**.
The router was accessed via web browser in the **Kali machine** (**100.100.2.100** as static IP address) through the given credentials, and then the desired service was configured through the **OPNSense** administration panel.
The following steps were taken to give the service the proper configuration provided by the assignment:

- the service was enabled on the router, as pictured in **Figure 2**, with the option **"Deny unknown clients"** checked in order to prevent undesired clients (the ones with a MAC address that is different from the ones specified and thus without ARP entries registered) from obtaining a dynamic IP address in the network. Keep also in mind that the subnet mask was already specified in the **CLIENTS** interface for the router, and the same goes for other services - such as **Unbound DNS**, so that these settings were not required to be configured again;
- the address range was initially specified as **100.100.2.101 - 100.100.2.253** so to avoid a re-assignment of known static IP addresses in the network - that is to say, those belonging to the Kali machine, the Arpwatch machine and, of course, the router itself. Later on, as pictured in **Figure 3**, a new pool of addresses was specified in the range **100.100.2.2 - 100.100.2.99** so to comprehend all the addresses that are not known to be assigned in the network;
- as pictured in **Figure 4**, the partial **MAC addresses** pointed out in the assignment as allowed to make use of the service were specified in the configuration, so that the only machines able to receive DHCP offers from the router will be the ones exhibiting these MAC addresses;
- as pictured in **Figure 5**, the six desired MAC addresses were then specified in the **DHCP Static Mappings** and a **Static ARP entry** was created for each of them so to remember their MAC address and mark them as known clients.

Once the **DHCP service** was confirmed running on the router, it was tested directly from the **Kali machine** by deleting the IP address set by default on **eth0** and thus trying to obtain a new one through the **DHCP service** of the router. First, as pictured in **Figure 6**, the **dhclient** command was run so that the machine could obtain a new dynamic IP address on the interface **eth0** - the machine has by default a **MAC address** on interface **eth0** which falls under the allowed addresses specified in the configuration. A **DHCP offer** was received by the router, and a new IP address was obtained, positively assessing the functioning of the service so far.
Again from the **Kali machine**, the **MAC address** of **eth0** was then changed to one that should not be allowed to request a new IP to the service, as pictured in **Figure 7**. The **dhclient** command was run again, and this time no offer was received, thus

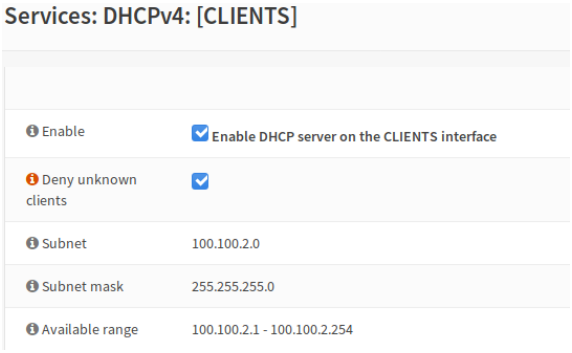

Figure 2: Enabling the DHCP service on the Internal Router.

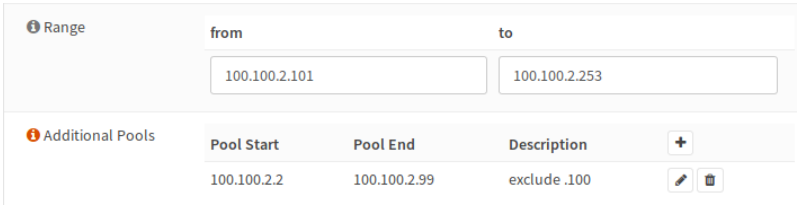

Figure 3: The specified range for assignable addresses on the DHCP service.

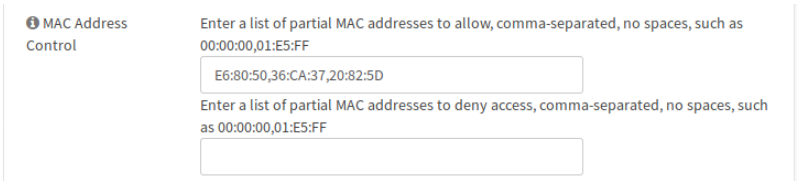

Figure 4: The partial MAC addresses specified in the configuration.

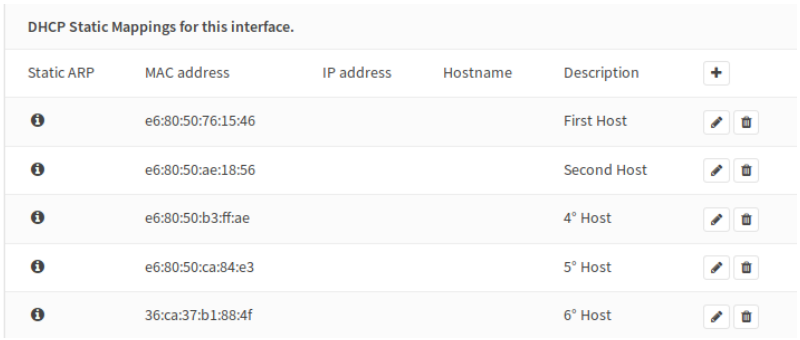

Figure 5: DHCP Static Mappings for the desired addresses - one is missing due to screen resolution on openVNC.

confirming that the service is only leasing IP addresses to machines having the desired **MAC addresses** on their interfaces. Since the machine had already received a lease by the service in the previous step, though, it was able to re-use the one previously obtained, but if it hadn't then no IP address would have been obtained.
Same test was run by slightly changing the original **MAC address** maintaining the first three bytes (**e6:80:50**), so to obtain a partial match of the address; as we can see from **Figure 8**, again no DHCP offer was received, confirming the desired behavior of the service.

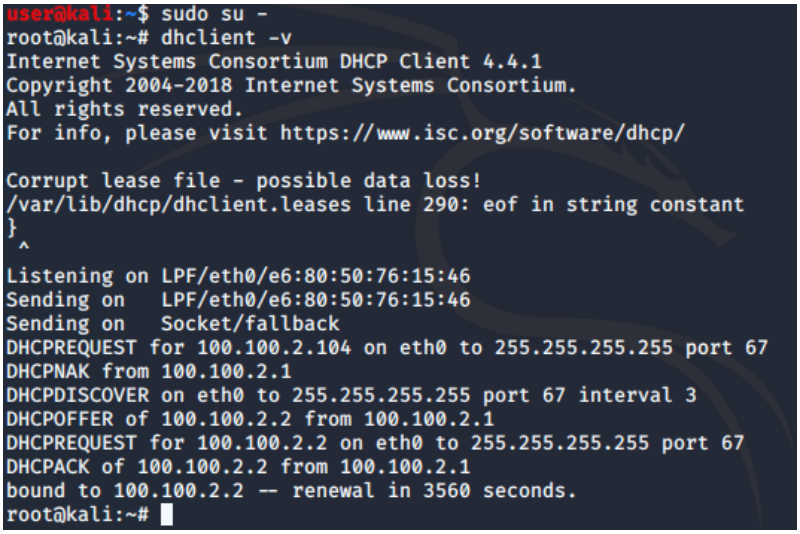

Figure 6: Kali machine with right MAC address on eth0 receiving a new IP address.

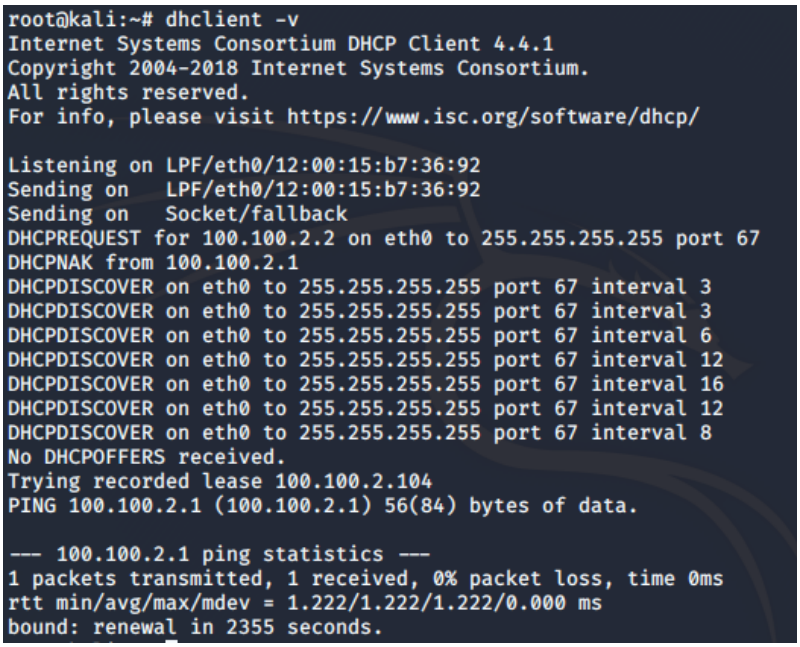

Figure 7: Kali machine unable to receive a DHCP offer due to its modified MAC address.

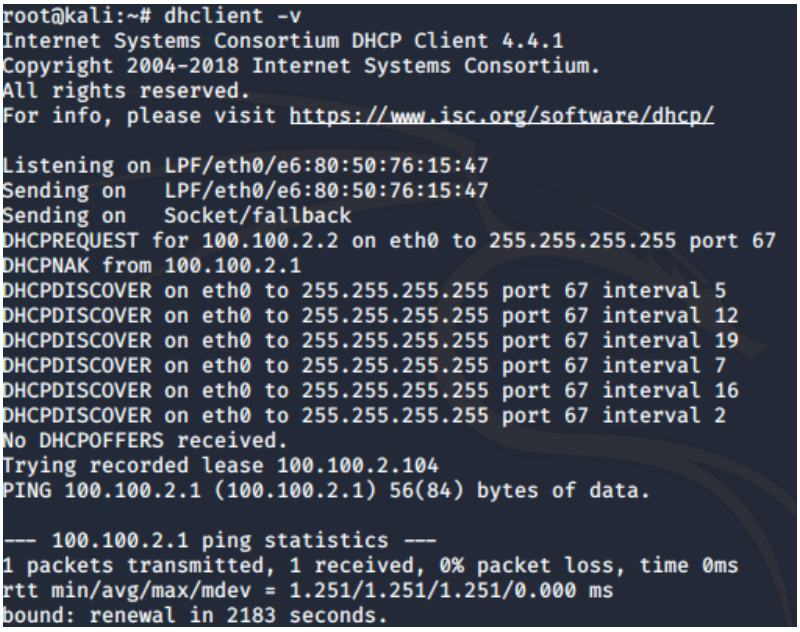

Figure 8: Kali machine unable to receive a DHCP offer due to its slightly modified MAC address.

3 Arpwatch tool configuration

This section is about how the **Arpwatch** tool was configured in the corresponding machine.
It is composed of two subparagraphs: the first one briefly describes the tool's configuration, while the scripts that were written in order to monitor the target network are described in the second one. Note that the security issues that we had to face and the countermeasures taken for this configuration to be actually effective are described in the next section as part of additional measures that were not required by the assignment, but somehow needed to secure the whole **Clients network**.

3.1 Arpwatch configuration

The **Arpwatch** tool was not installed by the default in the corresponding machine, so it had to be downloaded and installed via command *apt install arpwatch*. The empty file **eth0.iface** had also to be created in the tool's directory */etc/arpwatch/* in order for **Arpwatch** to be able to listen through the machine's interface linked to the **Clients network**'s LAN.
With the default configuration, the tool is already able to detect possible changes in the network when launched, and mail them to the **root** - and this function is still enabled at the end of the whole configuration, mails are stored under */mail/root/*.
In order for the configuration to be effective, it is enough to launch the tool in our scripts via command *arpwatch -i eth0 -n 100.100.2.0/24* so that it will listen to the changes in the *whole* network, meaning if any machine in the **Clients network** pings any other machine (not just the arpwatch one) in the aforementioned network, the tool will be able to eventually capture potential spoofing attempts and send them an alert.
Arpwatch usually logs any captured event in a Unix predefined log file under */var/log/messages*, but this may be inconvenient for us since this file holds every single log that was generated by the machine, and we are only interested in monitoring **Arpwatch**-generated logs. This is the reason why we chose to define a new file where the tool stores its own logs, by configuring the **rsyslog** tool installed by the default to actually redirect all the logs generated by **Arpwatch** in their own file, which can be found under */var/log/arpwatch.log*.
The aforementioned configuration can be found under */etc/rsyslog.d/arpwatch.conf*.

3.2 Monitoring scripts

Three scripts were written to actually monitor our target LAN: the first one, *startmonitoring.sh* is located under */etc/profile.d/* and it is only needed to launch at login the second one, located at */home/bashshellscript.sh*. This script is actually the core of the whole monitoring system: it launches **Arpwatch** as previously described, then it tails its log file so that, whenever a new line is generated, if it is *interesting* - meaning it possibly detects an intrusion event in the monitored network, thus reporting a **MAC address** - it sends as

input to the third script the detected **MAC address** and a **payload** that is, as required by the assignment, the whole log's new line.
The third script is a **Python script** named *scapySendPacket.py* which exploits **ScaPy** module to generate a new ethernet packet to be sent to the detected address thanks to the input received. We have to highlight at least two items at this point:

- the packet has to be sent with a header **type** field that is *different* from **0101**, the one requested by the assignemnt: this is due to the fact that the implementation of the **ScaPy** module only allows to specify values for this field that are **¿= 0x0600**, otherwise they are interpreted as values for the **length** field. To overcome this problem, we decided to assign to this field the value **0x88b5**, defined by the **IEEE** as **Local Experimental Ethertype** - so still an experimental type, at least;
- this **Python** script is tghe only one that takes some values as input; furthermore, it is the only one actually performing network operations in our configuration, so we have to keep this in mind since it may be the only script that actually requires some kind of input validation in order to be considered as secure.

The execution of the last script obviously required to install **python3**, **pip3** and **scapy** module via **pip3** on the arpwatch machine.

4 Additional measures

Configuring **Arpwatch** as described in the previous section is obviously not enough to guarantee a certain level of security in the target network, so we decided to take some extra measures that were not required by the assignment.
As a best practice, we chose to change the default passwords of every account in the machines of the network, and also the password for the admin account for the Inernal Router has been changed. Strong, secure passwords on the **arpwatch machine** are a must-have in order for the monitoring system to actually be effective.
Apart from best practices, we needed to take into account possible threats such as **escalation of privilege** and **tampering** with the scripts' input in the **arpwatch machine**, so to make it difficult for an adversary to disrupt the monitoring system.

4.1 Escalation of privilege on the Arpwatch machine

This issue is pretty self-explanatory: in order for the script to be running, **root** has to be logged-in in the machine, meaning that an adversary could simply access it knowing its location (i.e., its IP address) and, being already logged-in as **root**, simply terminate the execution of the monitoring scripts, e.g. just by logging out of the account. Notice that this scenario could lead to even worse results for our system, such as the adversary being able to do basically everything he wants from the **arpwatch machine**. First measure we took was to actually launch the script at login such as, even when **root** or any other user logs out from bash, the monitoring script keeps running: this result was obtained through the **nohup** command being concatenated to the script when it is launched. This way, even if the user who executed the script logs out, it keeps running.
We do not need the **root** account to be logged in now to keep watching for possible spoofing attacks, but it is in general a best practice to execute scripts with the least possible privilege, so monitoring through **root** still seems a pretty bad idea. Indeed, if an adversary is granted access as **root**, he can still terminate the execution of the script, reboot or shut down the system. As a further security measure, a new account was created on the machine - **watchdog** - with no directory assigned and no privileges granted, so that it cannot create new files, edit old ones or execute files he does not own. This user basically doesn't own any file, but is given some specific permissions in order to perform the monitoring activity: first, he is able to *only read* the arpwatch logs under */var/log/arpwatch.log*. Thanks to the **sudo** tool (by modifying *visudo*), **watchdog** is able to launch from login the monitoring scripts - so that he can read and execute them, but not modify or delete them - and the **Python script** (so basically everything already existing under *home* directory), which has been granted the privilege to actually create sockets and send packets through the **setcap** tool.
Also, the file **arp.dat** where the tool stores the arp entries is usually located at */var/lib/arpwatch*, but when run from **watchdog** arpwatch actually runs from directory /, so a new file **arp.dat** was created and owned by **root** in this directory. We tested and found out that **arpwatch**, when run by **watchdog**, uses this file and reads from it: this approach should be more secure since **watchdog** user cannot write to this second **arp.dat**, but **arpwatch** can still edit it while running without saving it, so it is able to store all the arp

entries it needs there. When the script is eventually terminated by performing a rebot or a shut down from **root**, the file isn't saved, meaning the arp entries are lost and new stations will have to be re-discovered at new script startup, but this doesn't seem to affect badly the live detection, supposing a reboot of the machine will not occur frequently.
Notice that every other action in the machine requires **watchdog** to input its own password, which was changed, and he cannot create new files without *sudo* since he does not even own a directory - so, even if files executed in **Python** environment are actually able to send packets over the network, **watchdog** cannot create nor execute them: if he wants to, he needs to input its own password to be granted superuser privileges.
Furthermore, commands such as **reboot** and **shutdown** are disabled by default for **watchdog** without *sudo* command, and even if the adversary logs out from this account, the monitoring script will keep running as previously mentioned.
Please notice also that *sudo* is always requested to perform such actions - as, basically, every other action - since the *sudoers* file was modified so that the default value for the *timestamptimeout* was set to **0**.
At this point, the adversary cannot pose a threat to the machine that is worse than just killing the process related to the monitoring script, which at least can be prevented by logging in as **watchdog**, let the machine run the script and then log out - if passwords of the two accounts are not known to the adversary, the script's execution cannot be terminated.
Since the adversary is able to execute the monitoring scripts, he could provide **scapySendPacket.py** a bad input and exploit it to send packets over the network, which is an issue we deal with in the next subsection.

4.2 Input validation in scapySendPacket.py

The first, non-security related input validation phase requires the input to be at least two strings long. This is due to the fact that at least one string is required to match the target MAC address of the packet (the very first input string) and another one to be the payload, which we do not want to be empty. An error string will alert the user tampering with the input in this case, and the script will exit.
Another input validation mechanism was introduced in the Python script to actually prevent an adversary which gained access to a shell in the machine to abrirtarily send packets over the LAN exploiting this script itself.
Since we want *scapySendPacket.py* to be triggered by new log lines in the arpwatch log file, before actually sending the requested payload to the specified MAC address, the script reads the log file and verifies that the payload itself is actually contained in this file.
This means that an adversary will not be able to send packets exploiting this script with an arbitrary payload, since the payload **must** be contained in the log file, else the script will terminate before sending the packet. Obviously, an adversary could still send its own packets by just setting as their input log lines that are already present in the log file, but at least it would be obvious to the sysadmin, and maybe also to the users receiving those packers, that such an attack is being performed: all the log lines - and thus, the payloads of the packet - contain *date* and *time*, so if they do not correspond to the date and time

the packet was actually received, an user should still be able to tell that something's off.

5 Testing the security measures

We did test x test y and test z and everything is cool.

6 Final remarks and possible improvements

blablabla

References

[1] *Just a placeholder.*