Project Report : CS3543 - Computer Networks 2
Mini-Nmap

Harsh Agarwal `cs15btech11019`

S. Vishwak `cs15btech11043`

Contents

1 Objective and Motivation

Nmap has been a very-effective tool for network discovery and security analysis for about 20 years now. The goal of the project was to implement certain network discovery techniques which are available in Nmap. A full implementation of Nmap would take quite a long time of development, hence in this project, we have decided to stick with certain techniques that are popularly used.

The motivation of the project was to get an idea of how such network discovery tools are implemented, and to objectively see how hard implementing a suite like Nmap would actually be.

2 Project Specifications

The size of the project stands at around 2000 lines of code written in C++ and Python. The tools implemented as a part of the project are:

- Host discovery utility
- Port scanning techniques: `SYN`, `FIN`, `NULL`, `XMAS` and `Decoy` scan
- A TCP sniffer
- ARP Poisoner (with ARP Doctor)

A full documentation of the code is available to be auto-generated for user convenience. This is discussed later.

3 Code Structure, Design Decisions and Tools used

3.1 Code Structure and Organization

The code is organized in the following manner:

- `ping.h/.cpp` consists of ICMP ping preliminaries required for the host discovery.
- `discover.h/.cpp` consists of code required for host discovery using ICMP messages.
- `packet.h/.cpp` consists of code which enables socket creation, packet processing (e.g., checksum calculation), layer-wise packet assembly.
- `scan.h/.cpp` consists of code required for scanning techniques.
- `sniff.h/.cpp` consists of code required for the TCP sniffing utility
- `error.h/.cpp` consists of error handling utilities.
- `logger.h/.cpp` consists of logging utilities.
- `arpPoison.py` is a Python module consists of code required for a Man-in-the-Middle Attack using ARP poisoning.

3.2 What happens in each of the files above?

3.2.1 `discover.h/.cpp`

The main host discovery takes place with the `discover_host()` function. The algorithm followed by the function is as follows:

Algorithm 1 Algorithm for Round-Robin based host discovery

```
 1: Get CIDR string and validate CIDR
 2: IPaddr, Mask = CIDR_string
 3: Get list of IPaddrs using IPaddr and Mask
 4: Populate a Queue with requests
 5: active = $\phi$
 6: while Queue $\neq \phi$ do
 7:     Open request and get the IPaddr
 8:     Ping the IPaddr and await reply
 9:     if Reply received then
10:         If IPaddr replies, active = active $\cup$ IPaddr
11:     else
12:         If non-zero trials for the request left, add to the queue. Otherwise, don't.
13:     end if
14: end while
15: return active
```

3.2.2 `packet.h/.cpp`

This a utility file. This helps create socket descriptors, assemble packets at a header level (`TCP`, `IP`), and calculate checksums.

The source code has been adapted and modified to requirements based on other code available online (Linux kernel). These links are available in the references and the function descriptions.

3.2.3 `ping.h/.cpp`

This is a utility file to handle `ICMP` packets. This file consists of functions to create sockets, assemble `ICMP` packets, and handling sending and receiving of ping messages.

3.2.4 `scan.h/.cpp`

Here is where the port scanning techniques take place. The most important function is `scan`. Here, too, we follow a round robin approach with requests, hence the algorithm is very similar to that specified above. Instead of `CIDR`, we will instead have a user specified range of ports, and instead of pinging we would be sending specialized packets pertaining to each technique. We have incorporated parallelism by chunking the ports to be scanned evenly across multiple threads. The scanning techniques are briefly explained below:

- SYN Scan: In SYN scanning, the attacking client attempts to set up a `TCP/IP` connection with a server at every possible port. This is done by sending a `SYN` (synchronization) packet, to mimic initiating a three-way handshake, to every port on the server. If the server responds with a `SYN/ACK` (synchronization acknowledged) packet from a particular port, it means the port is open. If the server responds with a `RST` (reset) packet from a particular port, it means the port is closed.

- `FIN` Scan: In FIN scanning, the specifications of RFC 793 are exploited. RFC 793 states that: "Traffic to a closed port should always return `RST`" and "If neither of the `SYN` or `RST` bits is set then drop the segment and return". Now, if you send a packet with a `FIN` bit set to 1, then for closed ports you will receive a `RST`, but for open ports / filtered ports you will not receive anything. Thus, if there is no firewall, then there is a good chance that timeout indicates activity of the port.

- `NULL, XMAS`: Both these work in the same way as FIN Scan, thereby exploiting the specifications of RFC 793.

- `Decoy` Scan: To prevent a filtering of the attacker's side, there is a utility called Decoy Scan. This sends spoofed packets mimicking random `IP` address to the victim `IP` address, thus making it hard to pinpoint the attacker.

3.2.5 `sniff.h/.cpp`

This is a sniffing utility. Sent and Received packets are analyzed in a `process_packet` function. This sniffer only analyzes `TCP` packets, since most of our scans are designed to work with the `TCP` header.

3.2.6 `arpPoison.py`

This has two key functions - `poisoner` and `doctor`. `poisoner` will tell the victim that the MAC address of the gateway is its own, and will tell the gateway that the victims's MAC address is its own. Now packets inbound to the victim from anywhere will be intercepted by the attacker. Messages from the victim to the gateway, will be intercepted by the attacker as well.

To revert the changes made, the "doctor" broadcasts messages as the gateway and victim asking for the MAC addresses of the victim and gateway respectively, thus restoring the ARP tables.

This is a simple man-in-the-middle attack, which is why we decided to add it in this suite. The network library ScaPy was used for this module alone. The main file will execute the attack for a given duration, after which the attacked reprises the role of a "doctor" and reverts the changes

3.3 Design Decisions

Certain design decisions were made as a part of the project. We chose C++ over Python, despite the huge module support from Python, since we believed that C++ was closer to the Linux Network API than Python, thus leaving room for less error. We have also made use of threads for parallel port scanning, and Python's threading module is not as efficient as C++, which is also another reason to migrate to C++.

However, for the ARP poisoning tool, we decided to go with Python instead of C++, since it would require dealing with the link-layer packets, which could cause a lot of pointer manipulation, which is tedious and error-prone.

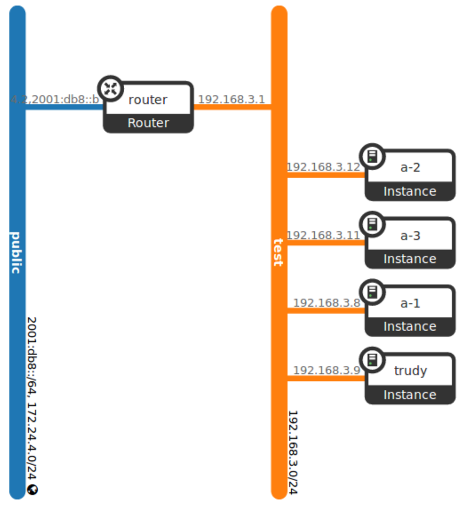
We have also effectively modularized our code, so that the utilities designed could also be used for other purposes than their intended usage as well. For example, we have a ICMP message sender which can be used in place of the `iputils` utility `ping`.

In the host discovery and port scanning techniques, we have decided to use a Round-robin technique. Here, we create a queue of "requests", and these "requests" are processed in a round-robin fashion. If a "request" is successfully completed or has run out of trials, then we remove it from the queue. Other alternatives were to wait until a reply arrived for a given request - which is susceptible to timeouts. This round-robin scheme allows us to pipeline the process as well.

We have also made use of raw sockets (specified using `SOCK_RAW`). This helps us modify the properties of the socket to our will (for example: sending spoofed packets), which is why we decided to use them over traditional stream or datagram sockets. Using raw sockets, we were able to populate the header appropriately as per requirements (for TCP, IP, and ICMP).

3.4 Tools used

We have made use of `OpenStack`. Using this, we have created a private subnet, which behaves as the test-bed for our tools. The setup can be seen to the right.



We have also made use of `tcpdump` and `WireShark` for monitoring the network traffic during the time of the attack.

For specifying input to the executable, we have used a JSON (Javascript Object Notation) parser designed by Niels Lohmann. The file `json.h` is code for the aforementioned parser.

The documentation in generated using Doxygen.

4 Contributions

Our contributions are listed below:

- Port Scanning techniques and the TCP Sniffer were implemented by Harsh Agarwal.
- Host discovery utility and ARP Poisoner (with Doctor) were implemented by Vishwak Srinivasan.

In addition, the `OpenStack` setup was made by Harsh Agarwal.

5 Code tutorial

5.1 Documentation generation

To obtain the documentation for the project, one can go the home folder of the project, and perform

```
make docs
```

This will create a `docs` directory in the same directory of the project. Opening `index.html`, will help view the project from a GUI.

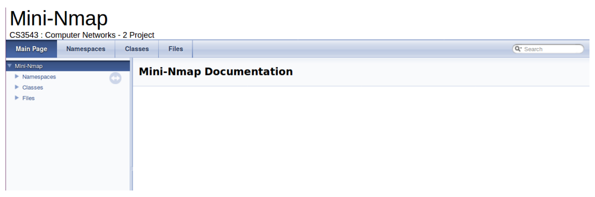
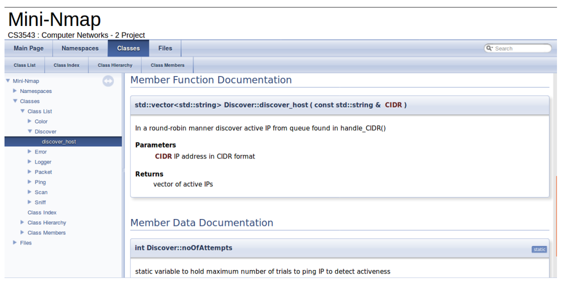


Figure 1: Home page of generated docs. View the taskbar on the left.



Member Function Documentation

std::vector<std::string> Discover::discover_host (const std::string & CIDR)

In a round-robin manner discover active IP from queue found in handle_CIDR()

Parameters
CIDR IP address in CIDR format

Returns
vector of active IPs

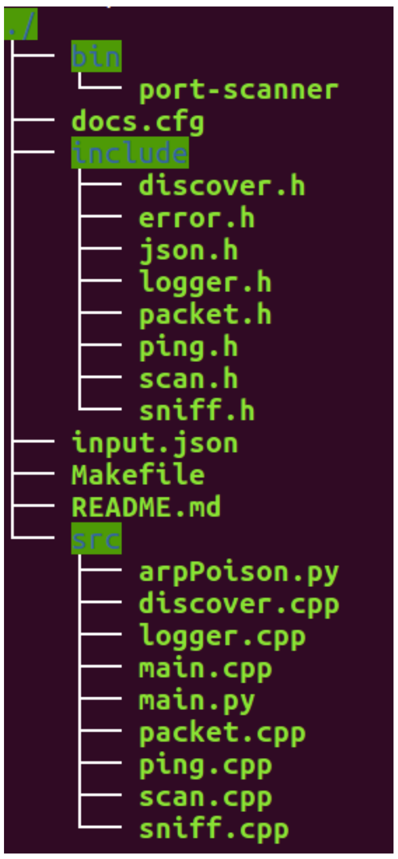
Member Data Documentation

int Discover::noOfAttempts

static variable to hold maximum number of trials to ping IP to detect activeness

Figure 2: One can view member functions and data as well

5.2 Building the project



```
./
├── bin
│   └── port-scanner
├── docs.cfg
├── include
│   ├── discover.h
│   ├── error.h
│   ├── json.h
│   ├── logger.h
│   ├── packet.h
│   ├── ping.h
│   ├── scan.h
│   └── sniff.h
├── input.json
├── Makefile
├── README.md
└── src
    ├── arpPoison.py
    ├── discover.cpp
    ├── logger.cpp
    ├── main.cpp
    ├── main.py
    ├── packet.cpp
    ├── ping.cpp
    ├── scan.cpp
    └── sniff.cpp
```

Again, using the `Makefile`, one can perform

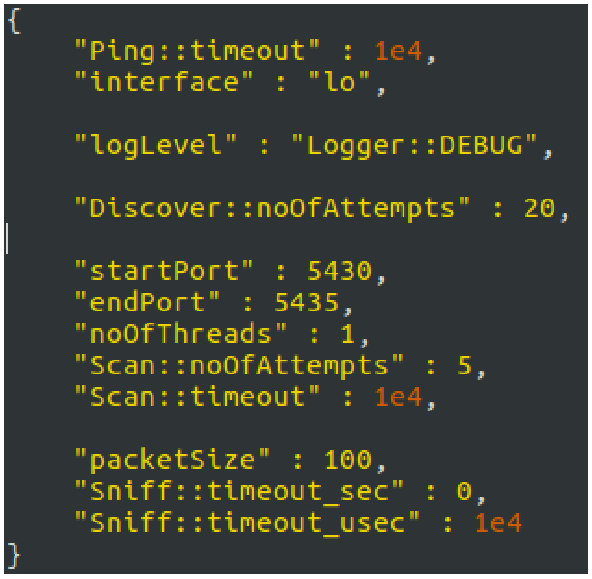
```
make all
```

This will build the project. The compiled binary named `port-scanner` will be placed in `./bin`, after compilation. Note the tree structure for more information.

5.3 Running the generated binary

The executable takes a configuration file supplied in a JSON format, the CIDR to get the list of IPs to test for activity and the scanning technique - one of `SYN`, `FIN`, `NULL`, `XMAS` and `Decoy`, in that order.

In the `JSON` file, you have to specify parameters based on which your port scanner will work. Beside is a sample `JSON` file. This file is also provided in the code.

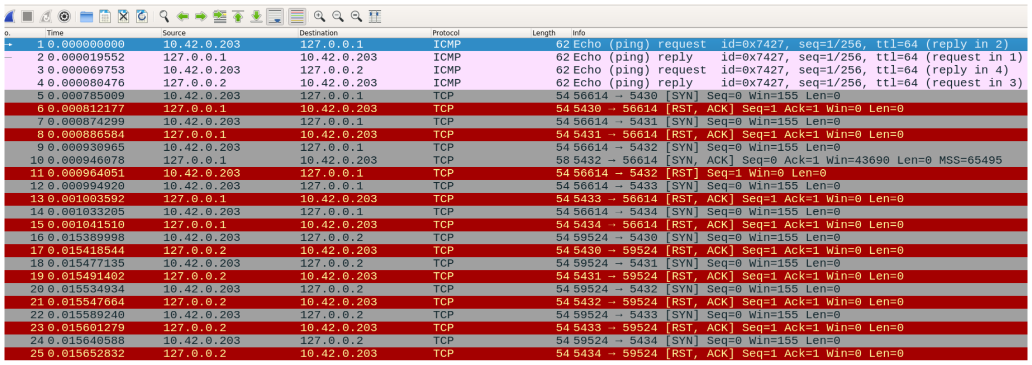


```
{
    "Ping::timeout" : 1e4,
    "interface" : "lo",

    "logLevel" : "Logger::DEBUG",

    "Discover::noOfAttempts" : 20,

    "startPort" : 5430,
    "endPort" : 5435,
    "noOfThreads" : 1,
    "Scan::noOfAttempts" : 5,
    "Scan::timeout" : 1e4,

    "packetSize" : 100,
    "Sniff::timeout_sec" : 0,
    "Sniff::timeout_usec" : 1e4
}
```

6 Results

6.1 SYN Scan

Below is the screenshot of a `Wireshark` trace of a successful `SYN` Scan. In the first 4 packets, you can see the IP discovery utility in action. After this, the `SYN` scan starts, indicated by a number of `SYN` packets.

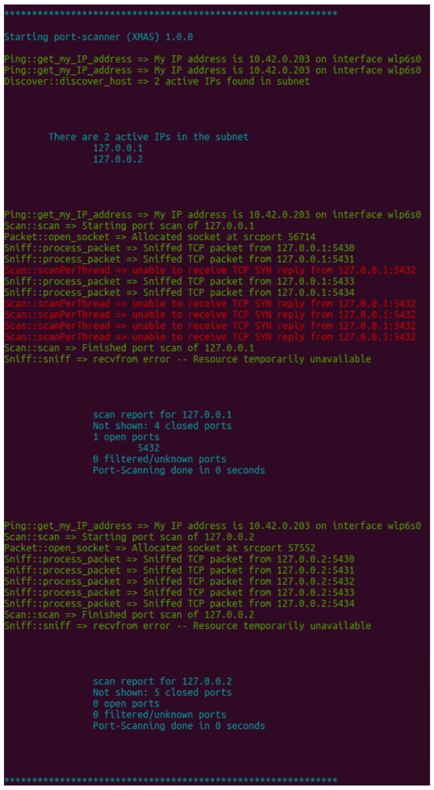


o.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	10.42.0.203	127.0.0.1	ICMP	62	Echo (ping) request id=0x7427, seq=1/256, ttl=64 (reply in 2)
2	0.000019552	127.0.0.1	10.42.0.203	ICMP	62	Echo (ping) reply id=0x7427, seq=1/256, ttl=64 (request in 1)
3	0.000069753	10.42.0.203	127.0.0.2	ICMP	62	Echo (ping) request id=0x7427, seq=1/256, ttl=64 (reply in 4)
4	0.000080476	127.0.0.2	10.42.0.203	ICMP	62	Echo (ping) reply id=0x7427, seq=1/256, ttl=64 (request in 3)
5	0.000785009	10.42.0.203	127.0.0.1	TCP	54	56614 → 5430 [SYN] Seq=0 Win=155 Len=0
6	0.000812177	127.0.0.1	10.42.0.203	TCP	54	5430 → 56614 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
7	0.000874299	10.42.0.203	127.0.0.1	TCP	54	56614 → 5431 [SYN] Seq=0 Win=155 Len=0
8	0.000886584	127.0.0.1	10.42.0.203	TCP	54	5431 → 56614 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
9	0.000930965	10.42.0.203	127.0.0.1	TCP	54	56614 → 5432 [SYN] Seq=0 Win=155 Len=0
10	0.000946078	127.0.0.1	10.42.0.203	TCP	58	5432 → 56614 [SYN, ACK] Seq=0 Ack=1 Win=43690 Len=0 MSS=65495
11	0.000964051	10.42.0.203	127.0.0.1	TCP	54	56614 → 5432 [RST] Seq=1 Win=0 Len=0
12	0.000994920	10.42.0.203	127.0.0.1	TCP	54	56614 → 5433 [SYN] Seq=0 Win=155 Len=0
13	0.001003592	127.0.0.1	10.42.0.203	TCP	54	5433 → 56614 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
14	0.001033205	10.42.0.203	127.0.0.1	TCP	54	56614 → 5434 [SYN] Seq=0 Win=155 Len=0
15	0.001041510	127.0.0.1	10.42.0.203	TCP	54	5434 → 56614 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
16	0.015389998	10.42.0.203	127.0.0.2	TCP	54	59524 → 5430 [SYN] Seq=0 Win=155 Len=0
17	0.015418544	127.0.0.2	10.42.0.203	TCP	54	5430 → 59524 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
18	0.015477135	10.42.0.203	127.0.0.2	TCP	54	59524 → 5431 [SYN] Seq=0 Win=155 Len=0
19	0.015491402	127.0.0.2	10.42.0.203	TCP	54	5431 → 59524 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
20	0.015534934	10.42.0.203	127.0.0.2	TCP	54	59524 → 5432 [SYN] Seq=0 Win=155 Len=0
21	0.015547664	127.0.0.2	10.42.0.203	TCP	54	5432 → 59524 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
22	0.015589240	10.42.0.203	127.0.0.2	TCP	54	59524 → 5433 [SYN] Seq=0 Win=155 Len=0
23	0.015601279	127.0.0.2	10.42.0.203	TCP	54	5433 → 59524 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
24	0.015640588	10.42.0.203	127.0.0.2	TCP	54	59524 → 5434 [SYN] Seq=0 Win=155 Len=0
25	0.015652832	127.0.0.2	10.42.0.203	TCP	54	5434 → 59524 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0

Figure 3: Note that there is a `SYN/ACK` for the port 5432, which we had open in the setup.

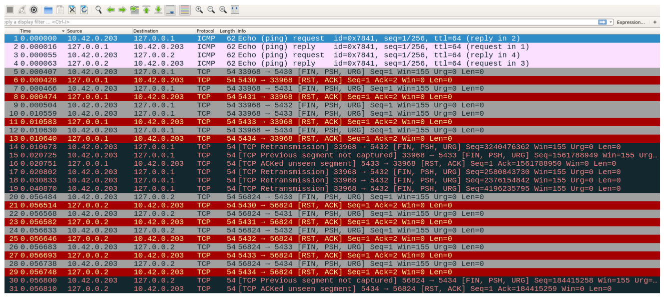
6.2 XMAS Scan

Below are some screenshot of the testing. As specified earlier, for those ports which are either filtering / open, there will be a timeout, and the closed ports will send a `RST` no matter what. The number of trials is 5, which is why you can see 4 re-transmissions.



```
************************************************************

Starting port-scanner (XMAS) 1.0.0

Ping::get_my_IP_address => My IP address is 10.42.0.203 on interface wlp6s0
Ping::get_my_IP_address => My IP address is 10.42.0.203 on interface wlp6s0
Discover::discover_host => 2 active IPs found in subnet



        There are 2 active IPs in the subnet
                127.0.0.1
                127.0.0.2



Ping::get_my_IP_address => My IP address is 10.42.0.203 on interface wlp6s0
Scan::scan => Starting port scan of 127.0.0.1
Packet::open_socket => Allocated socket at srcport 56714
Sniff::process_packet => Sniffed TCP packet from 127.0.0.1:5430
Sniff::process_packet => Sniffed TCP packet from 127.0.0.1:5431
Scan::scanPerThread => unable to receive TCP SYN reply from 127.0.0.1:5432
Sniff::process_packet => Sniffed TCP packet from 127.0.0.1:5433
Sniff::process_packet => Sniffed TCP packet from 127.0.0.1:5434
Scan::scanPerThread => unable to receive TCP SYN reply from 127.0.0.1:5432
Scan::scanPerThread => unable to receive TCP SYN reply from 127.0.0.1:5432
Scan::scanPerThread => unable to receive TCP SYN reply from 127.0.0.1:5432
Scan::scanPerThread => unable to receive TCP SYN reply from 127.0.0.1:5432
Scan::scan => Finished port scan of 127.0.0.1
Sniff::sniff => recvfrom error -- Resource temporarily unavailable



                scan report for 127.0.0.1
                Not shown: 4 closed ports
                1 open ports
                        5432
                0 filtered/unknown ports
                Port-Scanning done in 0 seconds



Ping::get_my_IP_address => My IP address is 10.42.0.203 on interface wlp6s0
Scan::scan => Starting port scan of 127.0.0.2
Packet::open_socket => Allocated socket at srcport 57552
Sniff::process_packet => Sniffed TCP packet from 127.0.0.2:5430
Sniff::process_packet => Sniffed TCP packet from 127.0.0.2:5431
Sniff::process_packet => Sniffed TCP packet from 127.0.0.2:5432
Sniff::process_packet => Sniffed TCP packet from 127.0.0.2:5433
Sniff::process_packet => Sniffed TCP packet from 127.0.0.2:5434
Scan::scan => Finished port scan of 127.0.0.2
Sniff::sniff => recvfrom error -- Resource temporarily unavailable



                scan report for 127.0.0.2
                Not shown: 5 closed ports
                0 open ports
                0 filtered/unknown ports
                Port-Scanning done in 0 seconds



************************************************************
```

Figure 4: The terminal output.

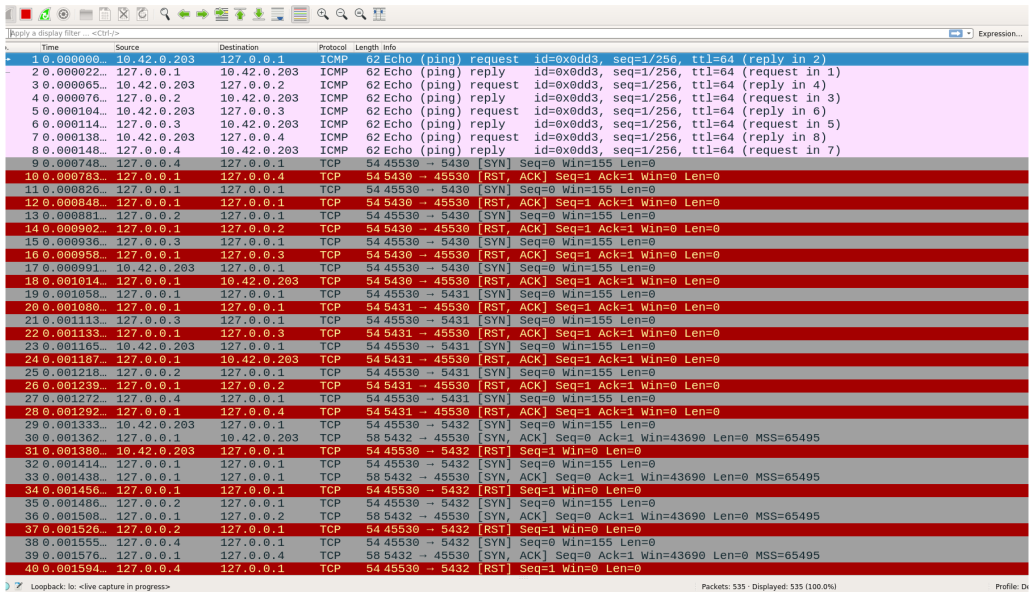


```
Apply a display filter ... <Ctrl-/>                                                              Expression...  +
    Time       Source         Destination    Protocol Length Info
  1 0.000000   10.42.0.203    127.0.0.1      ICMP     62 Echo (ping) request  id=0x7841, seq=1/256, ttl=64 (reply in 2)
  2 0.000016   127.0.0.1      10.42.0.203    ICMP     62 Echo (ping) reply    id=0x7841, seq=1/256, ttl=64 (request in 1)
  3 0.000055   10.42.0.203    127.0.0.2      ICMP     62 Echo (ping) request  id=0x7841, seq=1/256, ttl=64 (reply in 4)
  4 0.000063   127.0.0.2      10.42.0.203    ICMP     62 Echo (ping) reply    id=0x7841, seq=1/256, ttl=64 (request in 3)
  5 0.000407   10.42.0.203    127.0.0.1      TCP      54 33968 → 5430 [FIN, PSH, URG] Seq=1 Win=155 Urg=0 Len=0
  6 0.000428   127.0.0.1      10.42.0.203    TCP      54 5430 → 33968 [RST, ACK] Seq=1 Ack=2 Win=0 Len=0
  7 0.000466   10.42.0.203    127.0.0.1      TCP      54 33968 → 5431 [FIN, PSH, URG] Seq=1 Win=155 Urg=0 Len=0
  8 0.000474   127.0.0.1      10.42.0.203    TCP      54 5431 → 33968 [RST, ACK] Seq=1 Ack=2 Win=0 Len=0
  9 0.000504   10.42.0.203    127.0.0.1      TCP      54 33968 → 5432 [FIN, PSH, URG] Seq=1 Win=155 Urg=0 Len=0
 10 0.010559   10.42.0.203    127.0.0.1      TCP      54 33968 → 5433 [FIN, PSH, URG] Seq=1 Win=155 Urg=0 Len=0
 11 0.010583   127.0.0.1      10.42.0.203    TCP      54 5433 → 33968 [RST, ACK] Seq=1 Ack=2 Win=0 Len=0
 12 0.010630   10.42.0.203    127.0.0.1      TCP      54 33968 → 5434 [FIN, PSH, URG] Seq=1 Win=155 Urg=0 Len=0
 13 0.010640   127.0.0.1      10.42.0.203    TCP      54 5434 → 33968 [RST, ACK] Seq=1 Ack=2 Win=0 Len=0
 14 0.010673   10.42.0.203    127.0.0.1      TCP      54 [TCP Retransmission] 33968 → 5432 [FIN, PSH, URG] Seq=3240476362 Win=155 Urg=0 Len=0
 15 0.020725   10.42.0.203    127.0.0.1      TCP      54 [TCP Previous segment not captured] 33968 → 5433 [FIN, PSH, URG] Seq=1561788949 Win=155 Urg…
 16 0.020751   127.0.0.1      10.42.0.203    TCP      54 [TCP ACKed unseen segment] 5433 → 33968 [RST, ACK] Seq=1 Ack=1561788950 Win=0 Len=0
 17 0.020802   10.42.0.203    127.0.0.1      TCP      54 [TCP Retransmission] 33968 → 5432 [FIN, PSH, URG] Seq=2580843730 Win=155 Urg=0 Len=0
 18 0.030833   10.42.0.203    127.0.0.1      TCP      54 [TCP Retransmission] 33968 → 5432 [FIN, PSH, URG] Seq=2376154842 Win=155 Urg=0 Len=0
 19 0.040870   10.42.0.203    127.0.0.1      TCP      54 [TCP Retransmission] 33968 → 5432 [FIN, PSH, URG] Seq=4196235795 Win=155 Urg=0 Len=0
 20 0.056484   10.42.0.203    127.0.0.2      TCP      54 56824 → 5430 [FIN, PSH, URG] Seq=1 Win=155 Urg=0 Len=0
 21 0.056514   127.0.0.2      10.42.0.203    TCP      54 5430 → 56824 [RST, ACK] Seq=1 Ack=2 Win=0 Len=0
 22 0.056568   10.42.0.203    127.0.0.2      TCP      54 56824 → 5431 [FIN, PSH, URG] Seq=1 Win=155 Urg=0 Len=0
 23 0.056582   127.0.0.2      10.42.0.203    TCP      54 5431 → 56824 [RST, ACK] Seq=1 Ack=2 Win=0 Len=0
 24 0.056633   10.42.0.203    127.0.0.2      TCP      54 56824 → 5432 [FIN, PSH, URG] Seq=1 Win=155 Urg=0 Len=0
 25 0.056646   127.0.0.2      10.42.0.203    TCP      54 5432 → 56824 [RST, ACK] Seq=1 Ack=2 Win=0 Len=0
 26 0.056683   10.42.0.203    127.0.0.2      TCP      54 56824 → 5433 [FIN, PSH, URG] Seq=1 Win=155 Urg=0 Len=0
 27 0.056693   127.0.0.2      10.42.0.203    TCP      54 5433 → 56824 [RST, ACK] Seq=1 Ack=2 Win=0 Len=0
 28 0.056738   10.42.0.203    127.0.0.2      TCP      54 56824 → 5434 [FIN, PSH, URG] Seq=1 Win=155 Urg=0 Len=0
 29 0.056748   127.0.0.2      10.42.0.203    TCP      54 5434 → 56824 [RST, ACK] Seq=1 Ack=2 Win=0 Len=0
 30 0.056800   10.42.0.203    127.0.0.2      TCP      54 [TCP Previous segment not captured] 56824 → 5434 [FIN, PSH, URG] Seq=184415258 Win=155 Urg=…
 31 0.056810   127.0.0.2      10.42.0.203    TCP      54 [TCP ACKed unseen segment] 5434 → 56824 [RST, ACK] Seq=1 Ack=184415259 Win=0 Len=0
```

Figure 5: `Wireshark` output for `XMAS` scan

6.3 `Decoy` Scan

Below is a screenshot of the `Decoy` scan in action. Note that the IP address to which `SYN` packets are sent are not one but many, thus fooling the system. Also the `RST` is sent from the OS, which will prevent a seemingly `DOS` attack.

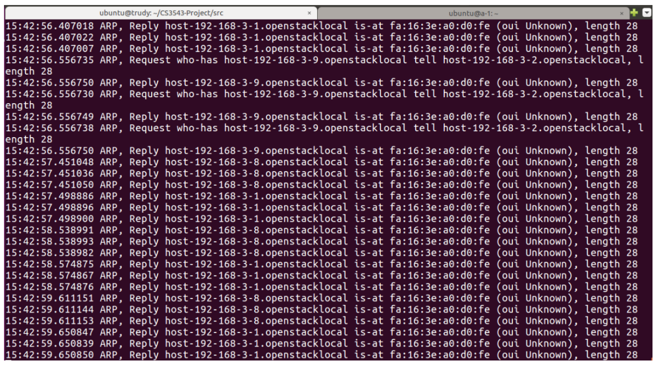


```
Apply a display filter ... <Ctrl-/>                                                    Expression...
    Time        Source         Destination    Protocol Length Info
  1 0.000000…   10.42.0.203    127.0.0.1      ICMP     62 Echo (ping) request  id=0x0dd3, seq=1/256, ttl=64 (reply in 2)
  2 0.000022…   127.0.0.1      10.42.0.203    ICMP     62 Echo (ping) reply    id=0x0dd3, seq=1/256, ttl=64 (request in 1)
  3 0.000065…   10.42.0.203    127.0.0.2      ICMP     62 Echo (ping) request  id=0x0dd3, seq=1/256, ttl=64 (reply in 4)
  4 0.000076…   127.0.0.2      10.42.0.203    ICMP     62 Echo (ping) reply    id=0x0dd3, seq=1/256, ttl=64 (request in 3)
  5 0.000104…   10.42.0.203    127.0.0.3      ICMP     62 Echo (ping) request  id=0x0dd3, seq=1/256, ttl=64 (reply in 6)
  6 0.000114…   127.0.0.3      10.42.0.203    ICMP     62 Echo (ping) reply    id=0x0dd3, seq=1/256, ttl=64 (request in 5)
  7 0.000138…   10.42.0.203    127.0.0.4      ICMP     62 Echo (ping) request  id=0x0dd3, seq=1/256, ttl=64 (reply in 8)
  8 0.000148…   127.0.0.4      10.42.0.203    ICMP     62 Echo (ping) reply    id=0x0dd3, seq=1/256, ttl=64 (request in 7)
  9 0.000748…   127.0.0.4      127.0.0.1      TCP      54 45530 → 5430 [SYN] Seq=0 Win=155 Len=0
 10 0.000783…   127.0.0.1      127.0.0.4      TCP      54 5430 → 45530 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
 11 0.000826…   127.0.0.1      127.0.0.1      TCP      54 45530 → 5430 [SYN] Seq=0 Win=155 Len=0
 12 0.000848…   127.0.0.1      127.0.0.1      TCP      54 5430 → 45530 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
 13 0.000881…   127.0.0.2      127.0.0.1      TCP      54 45530 → 5430 [SYN] Seq=0 Win=155 Len=0
 14 0.000902…   127.0.0.1      127.0.0.2      TCP      54 5430 → 45530 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
 15 0.000936…   127.0.0.3      127.0.0.1      TCP      54 45530 → 5430 [SYN] Seq=0 Win=155 Len=0
 16 0.000958…   127.0.0.1      127.0.0.3      TCP      54 5430 → 45530 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
 17 0.000991…   10.42.0.203    127.0.0.1      TCP      54 45530 → 5430 [SYN] Seq=0 Win=155 Len=0
 18 0.001014…   127.0.0.1      10.42.0.203    TCP      54 5430 → 45530 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
 19 0.001058…   127.0.0.1      127.0.0.1      TCP      54 45530 → 5431 [SYN] Seq=0 Win=155 Len=0
 20 0.001080…   127.0.0.1      127.0.0.1      TCP      54 5431 → 45530 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
 21 0.001113…   127.0.0.3      127.0.0.1      TCP      54 45530 → 5431 [SYN] Seq=0 Win=155 Len=0
 22 0.001133…   127.0.0.1      127.0.0.3      TCP      54 5431 → 45530 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
 23 0.001165…   10.42.0.203    127.0.0.1      TCP      54 45530 → 5431 [SYN] Seq=0 Win=155 Len=0
 24 0.001187…   127.0.0.1      10.42.0.203    TCP      54 5431 → 45530 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
 25 0.001218…   127.0.0.2      127.0.0.1      TCP      54 45530 → 5431 [SYN] Seq=0 Win=155 Len=0
 26 0.001239…   127.0.0.1      127.0.0.2      TCP      54 5431 → 45530 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
 27 0.001272…   127.0.0.4      127.0.0.1      TCP      54 45530 → 5431 [SYN] Seq=0 Win=155 Len=0
 28 0.001292…   127.0.0.1      127.0.0.4      TCP      54 5431 → 45530 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
 29 0.001333…   10.42.0.203    127.0.0.1      TCP      54 45530 → 5432 [SYN] Seq=0 Win=155 Len=0
 30 0.001362…   127.0.0.1      10.42.0.203    TCP      58 5432 → 45530 [SYN, ACK] Seq=0 Ack=1 Win=43690 Len=0 MSS=65495
 31 0.001380…   10.42.0.203    127.0.0.1      TCP      54 45530 → 5432 [RST] Seq=1 Win=0 Len=0
 32 0.001414…   127.0.0.1      127.0.0.1      TCP      54 45530 → 5432 [SYN] Seq=0 Win=155 Len=0
 33 0.001438…   127.0.0.1      127.0.0.1      TCP      58 5432 → 45530 [SYN, ACK] Seq=0 Ack=1 Win=43690 Len=0 MSS=65495
 34 0.001456…   127.0.0.1      127.0.0.1      TCP      54 45530 → 5432 [RST] Seq=1 Win=0 Len=0
 35 0.001486…   127.0.0.2      127.0.0.1      TCP      54 45530 → 5432 [SYN] Seq=0 Win=155 Len=0
 36 0.001508…   127.0.0.1      127.0.0.2      TCP      58 5432 → 45530 [SYN, ACK] Seq=0 Ack=1 Win=43690 Len=0 MSS=65495
 37 0.001526…   127.0.0.2      127.0.0.1      TCP      54 45530 → 5432 [RST] Seq=1 Win=0 Len=0
 38 0.001555…   127.0.0.4      127.0.0.1      TCP      54 45530 → 5432 [SYN] Seq=0 Win=155 Len=0
 39 0.001576…   127.0.0.1      127.0.0.4      TCP      58 5432 → 45530 [SYN, ACK] Seq=0 Ack=1 Win=43690 Len=0 MSS=65495
 40 0.001594…   127.0.0.4      127.0.0.1      TCP      54 45530 → 5432 [RST] Seq=1 Win=0 Len=0
Loopback: lo: <live capture in progress>                      Packets: 535 · Displayed: 535 (100.0%)            Profile: De
```

6.4 ARP Poisoner

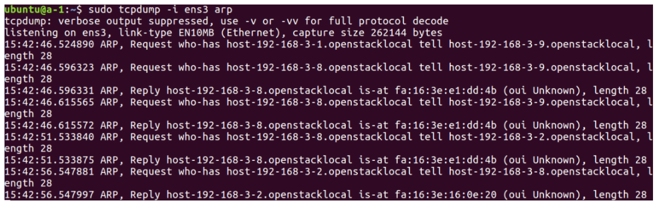
The poisoner successfully sends packets whilst trying to spoof the ARP entries. Unfortunately, these packets were not able to be detected at the hosts on OpenStack. Digging into the issue led us to a patch in OpenStack that prevents this behaviour. We didn't want to test this on a live network fearing it might cause issues to multiple-systems.

Below are the screenshots of the ARP Poisoner at work. These are outputs from the tcpdump utility:



```
15:42:56.407018 ARP, Reply host-192-168-3-1.openstacklocal is-at fa:16:3e:a0:d0:fe (oui Unknown), length 28
15:42:56.407022 ARP, Reply host-192-168-3-1.openstacklocal is-at fa:16:3e:a0:d0:fe (oui Unknown), length 28
15:42:56.407007 ARP, Reply host-192-168-3-1.openstacklocal is-at fa:16:3e:a0:d0:fe (oui Unknown), length 28
15:42:56.556735 ARP, Request who-has host-192-168-3-9.openstacklocal tell host-192-168-3-2.openstacklocal, l
ength 28
15:42:56.556750 ARP, Reply host-192-168-3-9.openstacklocal is-at fa:16:3e:a0:d0:fe (oui Unknown), length 28
15:42:56.556730 ARP, Request who-has host-192-168-3-9.openstacklocal tell host-192-168-3-2.openstacklocal, l
ength 28
15:42:56.556749 ARP, Reply host-192-168-3-9.openstacklocal is-at fa:16:3e:a0:d0:fe (oui Unknown), length 28
15:42:56.556738 ARP, Request who-has host-192-168-3-9.openstacklocal tell host-192-168-3-2.openstacklocal, l
ength 28
15:42:56.556750 ARP, Reply host-192-168-3-9.openstacklocal is-at fa:16:3e:a0:d0:fe (oui Unknown), length 28
15:42:57.451048 ARP, Reply host-192-168-3-8.openstacklocal is-at fa:16:3e:a0:d0:fe (oui Unknown), length 28
15:42:57.451036 ARP, Reply host-192-168-3-8.openstacklocal is-at fa:16:3e:a0:d0:fe (oui Unknown), length 28
15:42:57.451050 ARP, Reply host-192-168-3-8.openstacklocal is-at fa:16:3e:a0:d0:fe (oui Unknown), length 28
15:42:57.498886 ARP, Reply host-192-168-3-1.openstacklocal is-at fa:16:3e:a0:d0:fe (oui Unknown), length 28
15:42:57.498896 ARP, Reply host-192-168-3-1.openstacklocal is-at fa:16:3e:a0:d0:fe (oui Unknown), length 28
15:42:57.498900 ARP, Reply host-192-168-3-1.openstacklocal is-at fa:16:3e:a0:d0:fe (oui Unknown), length 28
15:42:58.538991 ARP, Reply host-192-168-3-8.openstacklocal is-at fa:16:3e:a0:d0:fe (oui Unknown), length 28
15:42:58.538993 ARP, Reply host-192-168-3-8.openstacklocal is-at fa:16:3e:a0:d0:fe (oui Unknown), length 28
15:42:58.538982 ARP, Reply host-192-168-3-8.openstacklocal is-at fa:16:3e:a0:d0:fe (oui Unknown), length 28
15:42:58.574875 ARP, Reply host-192-168-3-1.openstacklocal is-at fa:16:3e:a0:d0:fe (oui Unknown), length 28
15:42:58.574867 ARP, Reply host-192-168-3-1.openstacklocal is-at fa:16:3e:a0:d0:fe (oui Unknown), length 28
15:42:58.574876 ARP, Reply host-192-168-3-1.openstacklocal is-at fa:16:3e:a0:d0:fe (oui Unknown), length 28
15:42:59.611151 ARP, Reply host-192-168-3-8.openstacklocal is-at fa:16:3e:a0:d0:fe (oui Unknown), length 28
15:42:59.611144 ARP, Reply host-192-168-3-8.openstacklocal is-at fa:16:3e:a0:d0:fe (oui Unknown), length 28
15:42:59.611153 ARP, Reply host-192-168-3-8.openstacklocal is-at fa:16:3e:a0:d0:fe (oui Unknown), length 28
15:42:59.650847 ARP, Reply host-192-168-3-1.openstacklocal is-at fa:16:3e:a0:d0:fe (oui Unknown), length 28
15:42:59.650839 ARP, Reply host-192-168-3-1.openstacklocal is-at fa:16:3e:a0:d0:fe (oui Unknown), length 28
15:42:59.650850 ARP, Reply host-192-168-3-1.openstacklocal is-at fa:16:3e:a0:d0:fe (oui Unknown), length 28
```

Figure 6: Trudy is successfully spoofing the network



```
ubuntu@a-1:~$ sudo tcpdump -i ens3 arp
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on ens3, link-type EN10MB (Ethernet), capture size 262144 bytes
15:42:46.524890 ARP, Request who-has host-192-168-3-1.openstacklocal tell host-192-168-3-9.openstacklocal, l
ength 28
15:42:46.596323 ARP, Request who-has host-192-168-3-8.openstacklocal tell host-192-168-3-9.openstacklocal, l
ength 28
15:42:46.596331 ARP, Reply host-192-168-3-8.openstacklocal is-at fa:16:3e:e1:dd:4b (oui Unknown), length 28
15:42:46.615565 ARP, Request who-has host-192-168-3-8.openstacklocal tell host-192-168-3-9.openstacklocal, l
ength 28
15:42:46.615572 ARP, Reply host-192-168-3-8.openstacklocal is-at fa:16:3e:e1:dd:4b (oui Unknown), length 28
15:42:51.533840 ARP, Request who-has host-192-168-3-8.openstacklocal tell host-192-168-3-2.openstacklocal, l
ength 28
15:42:51.533875 ARP, Reply host-192-168-3-8.openstacklocal is-at fa:16:3e:e1:dd:4b (oui Unknown), length 28
15:42:56.547881 ARP, Request who-has host-192-168-3-2.openstacklocal tell host-192-168-3-8.openstacklocal, l
ength 28
15:42:56.547997 ARP, Reply host-192-168-3-2.openstacklocal is-at fa:16:3e:16:0e:20 (oui Unknown), length 28
```

Figure 7: A-1 (a victim) is unable to receiving the spoofed ARP messages

7 Scope for improvement

We had a working prototype a firewall, which will be able to detect port-scanners. We were able to test it completely, which is why we haven't submitted it as a part of the project.

Furthermore, we also would like improve this by adding an OS fingerprinting technique, wherein specifics of the networking APIs of multiple OSes are exploited to possibly find of the OS operating at each port.

Bibliography

Linux TCP Header source code. Link.

Transmission Control Protocol. RFC 793, 1981.

Computing the Internet checksum. RFC 1071, 1988.

Marco de Vivo, Le Ke, Germinal Isern, and Gabriela O. de Vivo. A review of port scanning techniques. 1999.

Niels Lohmann. JSON parser. Link.

Gordon Fyodor Lyon. *Nmap Network Scanning: The Official Nmap Project Guide to Network Discovery and Security Scanning*. 2009.