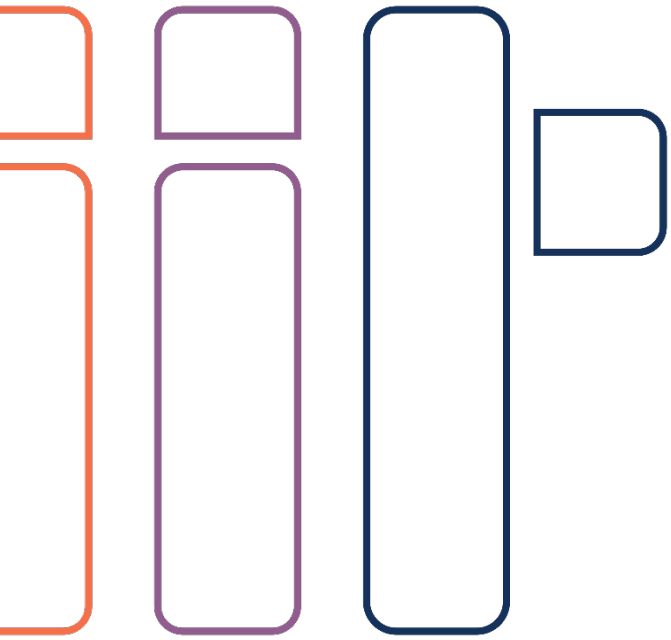


TECHNICAL REPORT

IIT TR-07/2023

Controlling and monitoring Ethernet-based network infrastructures: practical implementations using scapy

A. Gebrehiwot, F. Lauria

ISTITUTO DI INFORMATICA E TELEMATICA

Consiglio Nazionale delle Ricerche

Controlling and monitoring Ethernet-based network infrastructures: practical implementations using scapy

Abraham Gebrehiwot, Filippo Maria Lauria
firstname.lastname@iit.cnr.it

Technology Unit Computer and Communication Networks
Institute of Informatics and Telematics — Italian National Research Council
via G. Moruzzi, 1 — 56124 Pisa, Italy

Abstract
This document explores control and monitoring mechanisms commonly employed in Ethernet-based network infrastructures, aiming to provide a comprehensive understanding of their functionality. It presents practical script examples that utilize scapy, a powerful and user-friendly Python library for sensing and manipulating network packets. The showcased scripts focus on essential functionalities such as ARP monitoring, IPv4 collision detection, and rogue DHCP server detection. By examining these examples, readers can gain a comprehensive understanding of how these mechanisms contribute to network control and maintenance. The main objective is to offer valuable insights and practical applications of these mechanisms within Ethernet-based network infrastructures.

Introduction
Ethernet-based network infrastructures are widely used and essential for communication and resource sharing within a local environment. [1] However, ensuring effective control and monitoring of these type of networks is crucial to identify and address potential security and management issues.

This document provides a comprehensive overview of various control and monitoring mechanisms used in ethernet networks. To illustrate these concepts practically, it focuses on utilizing scapy, a powerful Python library for sensing and manipulating network packets. Scapy offers a wide range of functionalities that facilitate the implementation of network monitoring and control scripts. [2] Throughout the document, three specific scripts are exemplified: ARP monitoring, IP collision detection, and rogue DHCP server detection. These scripts serve as practical implementations of the discussed mechanisms.

Moreover, not only has been emphasized the practical implementation but also the understanding of how these mechanisms function. By exploring the underlying principles and protocols, readers can gain insight into the inner workings of the presented mechanisms and develop a deeper understanding of ethernet networks control and maintenance.

In summary, this document serves as a comprehensive guide to explore control and monitoring mechanisms in Ethernet-based network infrastructures, utilizing scapy as a practical tool for implementing network monitoring and control scripts. It provides a balance of theoretical explanations and practical examples to enhance readers' understanding and proficiency in network control and monitoring.

Sensing packets with scapy

One of the fundamental concepts in using scapy for network monitoring is packet sensing (more commonly called packet sniffing [3]). Scapy provides the `sniff` function, which allows capturing packets from the network and returning them as a packet list.

The `sniff` function accepts several input parameters. The ones of interest to us are described as follows (taken directly from the official documentation [2]):

- `count` specifies the number of packets to capture. Setting it to `0` means capturing packets indefinitely;
- `store` determines whether to store sniffed packets or discard them. As per our case, when monitoring the network continuously, it is recommended to set `store` to `0` to prevent memory consumption;
- `filter`: specifies a *Berkeley Packet Filter (BPF)* [4] to apply for capturing specific types of packets. This allows you to filter out unnecessary packets and focus on the ones relevant to your monitoring needs.
- `prn`: represents the function to apply to each captured packet. If something is returned from the function, it will be displayed. For example, you can use the lambda function `prn = lambda x: x.summary()` to print a summary of each captured packet[1].

By leveraging the `sniff` function with appropriate parameters, packet sniffing capabilities can be implemented to create powerful network monitoring tools. This foundation enables the capturing, analysis, and processing of network packets to detect a wide range of network events and anomalies.

In the following sections, we will introduce specific ideas on how to build some tool that utilizes the packet sniffing capabilities of scapy to address different aspects of network monitoring and security.

[1] with the `prn` parameter, a callback function is effectively passed to the `sniff` function. This callback (asynchronous in this case) will be invoked every time a new packet is sniffed, and it will receive the packet itself as its parameter.

ARP monitoring

In a switched Ethernet-based network infrastructures[2], the protocol used by IPv4 hosts to obtain the MAC address of another host to communicate with is ARP (Address Resolution Protocol [5]). ARP essentially associates a physical address (MAC address) with a logical address (IPv4 address), and vice versa, both belonging to the same host. The protocol operates on a request/response mechanism: when host A (with MAC address A_{MAC} and IPv4 address A_{IP}) wants to determine the physical address of host B (knowing its B_{IP}), it sends an ARP request message (referred to as `who-has` in scapy) to the *ethernet broadcast address* and waits for a response (a `is-at` message in scapy).

```
>>> request = ARP(pdst='10.100.1.254', op='who-has')
>>> request.show()
...
  op = who-has
  hwsrc = 08:00:27:00:00:0A  # A_MAC
  psrc = 10.100.1.23  # A_IP
  hwdst = 00:00:00:00:00:00  # B_MAC (unknown)
  pdst = 10.100.1.254  # B_IP
```

Through this process, host A learns the MAC address of host B. The mechanism is based on trust, where the requester (in the above example, A) trusts that the responder (in this case, B) is indeed who they claim to be.

```
>>> results = sr(request)
>>> reply = results[0][0][1]
>>> reply.show()
...
  op = is-at
  hwsrc = 08:00:27:00:00:0A
  psrc = 10.100.1.254
  hwdst = 08:00:27:00:00:0B  # B_MAC
  pdst = 10.100.1.23
```

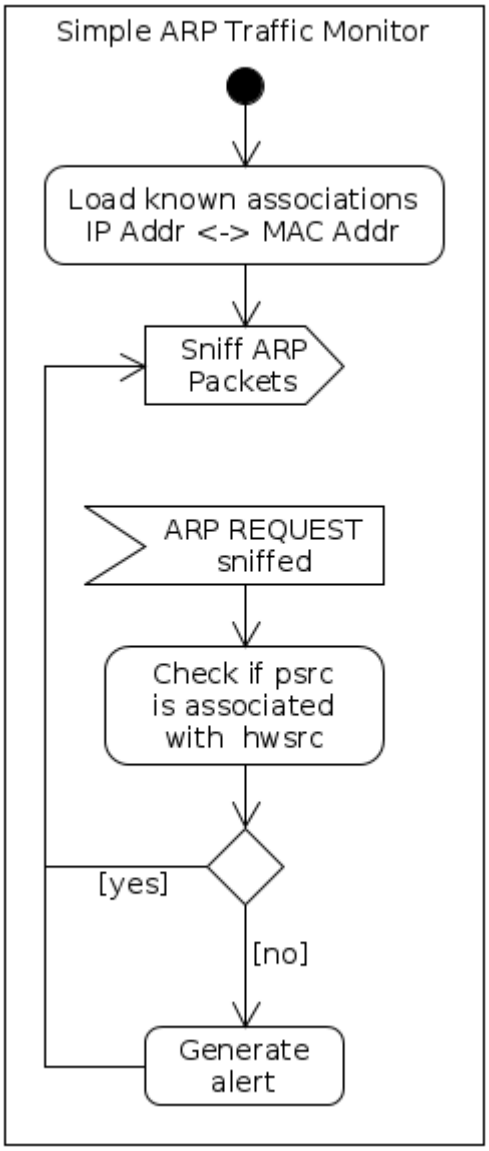
This may pose the risk of *Man-in-the-Middle attacks* in the network by malicious users. [7] It is also possible for non-malicious users to mistakenly assign their host, for example, the IPv4 address of the default gateway, which in this case would result in the disruption of outgoing traffic from the network. [6] To mitigate the latter issue, it is possible to use a tool for monitoring all ARP requests in the network[3].

The tool should detect when an IPv4 address, which has been officially assigned by the network administrator to an interface identified by a MAC address[4], is being used[5] by another host[6]. In other words, the purpose of this tool is to alert the network administrator when a non-authorized user configures, on their host, an IPv4 address that is already assigned to and in use by a known network device[7].

---

[2] such as Ethernet-based networks, e.g. university campus networks, etc.
[3] it is important to understand that this type of traffic is limited to broadcast requests only
[4] e.g. of the default gateway or another network device
[5] e.g. due to misconfiguration
[6] i.e. an host with a different MAC address
[7] e.g. the default gateway or a web server hosting the company's intranet site

Simple ARP traffic monitor



The simplified algorithm that the tool implement is outlined as follows.

The administrator should provide the monitoring tool with the correct IPv4⇔MAC address associations. This can be done by loading them from a configuration file, where each line represents an IPv4⇔MAC association. Once these associations are obtained, the tool initiates the monitoring of network traffic, specifically focusing on ARP traffic (using `arp` as BPF [4]). For every detected ARP request, the tool checks if the sender of the request has not "taken" an IPv4 address (`psrc`) that is assigned to a specific MAC address (`hwsrc`), as designated by the administrator.

If the `psrc` field contains an IP address that the tool recognizes as a monitored IP and the corresponding MAC address in the `hwsrc` field matches the associated MAC address, the tool does not take any action. However, if the `psrc` field contains a monitored IP, but the `hwsrc` field contains a MAC address that does not correspond to the associated MAC address, the tool generates an alert in the form of a message.

A flowchart illustrating the simplified algorithm discussed above is depicted in the figure on the left.

Simple ARP traffic monitor: a possible implementation

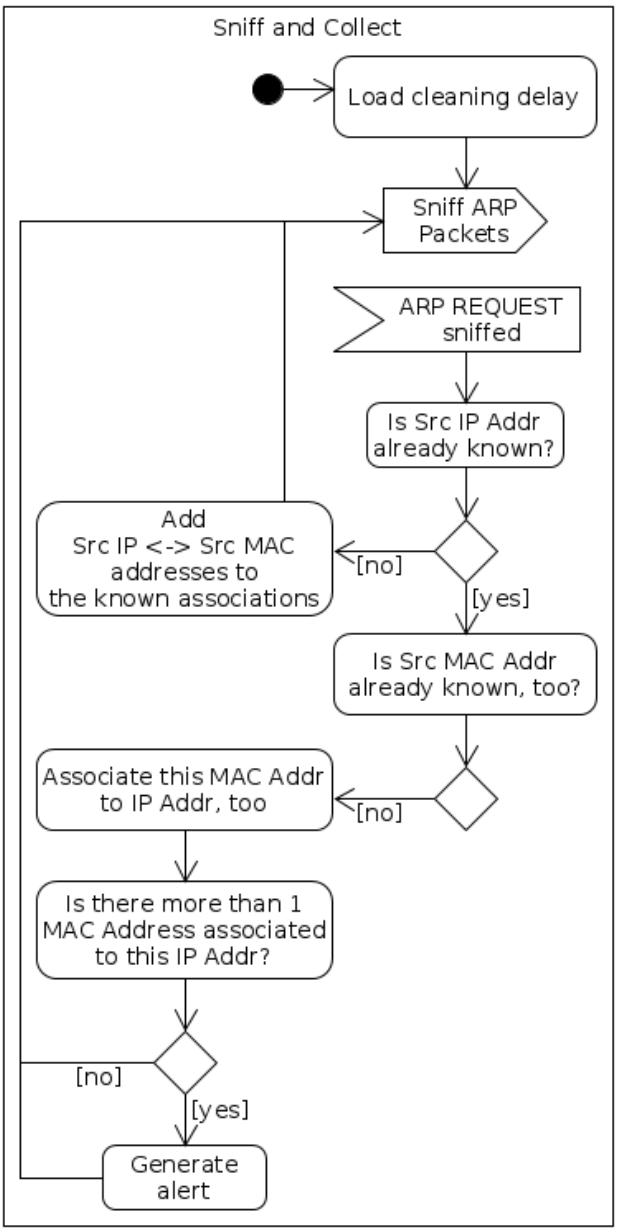
In the following Python code, we have implemented a possible version of the aforementioned ARP monitoring tool. One key aspect to note is that the code has been extensively commented to ensure a clear understanding of its functionality and usage.

Additionally, it's important to note that we have utilized the following libraries, in addition to scapy, to build this tool:

- `argparse`: by utilizing this library, we can easily parse and handle command-line arguments and options;
- `netaddr`: with the help of this library, we can work with IP and MAC addresses conveniently.

```
#!/usr/bin/env python
from scapy.all import *
import argparse
from netaddr import EUI, IPAddress
import os.path
import sys

# Callback function to handle captured packets
def packet_callback(p):
    arp = p[ARP]
    if arp.op != 1:
        return

    psrc = IPAddress(arp.psrc)
    hwsrc = EUI(arp.hwsrc)

    # Check if the source IP is in the dictionary of associations
    if psrc not in pairs:
        return

    # Compare the source MAC address with the associated MAC address
    if hwsrc != pairs[psrc]:
        print("ALERT! Wrong association {0} <--> {1} should be {0} <--> {2}!".format(
              psrc, hwsrc, pairs[psrc]))

# Parse command-line arguments
parser = argparse.ArgumentParser(description='ARP-Mon')
parser.add_argument("--interface", "-i", nargs="?", default="eth0", metavar="interface",
                    type=str, help="network interface name")
parser.add_argument("dictionary", type=str, metavar="dictionary",
                    help="an ASCII file containing a list of associations IP Addr <-> MAC Addr")

args = parser.parse_args()
iface = args.interface
fname = args.dictionary

# Check if the dictionary file exists
if not os.path.isfile(fname):
    print("The given file is not valid.")
    sys.exit(-1)

pairs = {}

# Load IP-MAC address associations from the dictionary file
with open(fname) as fd:
    for line in fd:
        pair = line.split()
        ip = IPAddress(pair[0])
        mac = EUI(pair[1])
        pairs[ip] = mac
        print("Loaded {0} <---> {1}".format(ip, mac))

print("Loaded {} associations.".format(len(pairs)))
print("Starting ARP-Mon...")

# Start sniffing ARP traffic on the specified interface
sniff(iface=iface, prn=packet_callback, filter="arp", store=0, count=0)
```

Simple IPv4 collision detector



The purpose of this tool is to alert the network administrator when two or more hosts within the same network are using the same IP address. This situation typically occurs due to misconfiguration or it could be a deliberate and malicious act by one of the hosts. The tool consists of two execution flows. The main execution flow of the tool is outlined below. Additionally, the tool needs to store a one-to-many association in a Python data structure (dictionary):

```
IP Address => [
    MAC Address 1,
    MAC Address 2,
    ...,
    MAC Address N
]
```

The second execution flow periodically and completely clears (using the `clear()` method) the dictionary containing the previously seen associations.

The main execution flow begins by loading a delay value, which represents the number of seconds between each clear operation. When the tool captures an ARP packet, it first checks if the source IP address (`psrc`) is already known.

If the IP address is not recognized, a new association is created between the source IP address (`psrc`) and the corresponding MAC address (`hwsrc`). However, if the IP address is already known, the tool then checks if the source MAC address (`hwsrc`) is also recorded. If the MAC address is not found, it is added to the list of MAC addresses associated with that particular IP address (`psrc`). Subsequently, the tool examines whether there are multiple MAC addresses associated with the same IP address. If such a case is detected, an alert is generated to notify the network administrator. An explicit ARP request of the detected IP address (`psrc`) may also be done to confirm the presence of multiple hosts, with different MAC addresses (`hwsrc`), in the network.

Simple IPv4 collision detector: a possible implementation

The code begins with importing necessary modules and defining variables and data structures:

- `lock` is a threading lock used to synchronize access to the associations dictionary;
- `associations` is a dictionary that stores the associations between IP addresses and MAC addresses;
- the `CleanerThread` class represents a secondary execution flow responsible for periodically clearing the associations dictionary;
- the `run` method of `CleanerThread` is overridden to define the execution logic for the thread.

Within the `run` method, the dictionary is cleared periodically if it contains any associations.

```
from scapy.all import ARP, sniff
from netaddr import EUI, IPAddress
import argparse, threading, time

lock = threading.RLock()
associations = {}

class CleanerThread(threading.Thread):
    def __init__(self, group=None, target=None, name='Cleaner',
                 args=(), kwargs=None, verbose=None):
        threading.Thread.__init__(self, group=group, target=target,
                                  name=name, verbose=verbose)
        self.args = args
        self.kwargs = kwargs
        return

    def run(self):
        while True:
            lock.acquire()
            try:
                l = len(associations)
                if l > 0:
                    print("Cleaning {0} associations..".format(l))
                    associations.clear()
            finally:
                lock.release()
            time.sleep(cleaning_time)
        return
```

The `packet_callback` function represents the main execution flow responsible for handling sniffed ARP packets:

- it extracts the source IP address (`psrc`) and source MAC address (`hwsrc`) from the ARP packet, then the `lock` is acquired to ensure thread-safe access to the associations dictionary;
- if the `psrc` is not present in the dictionary, it creates a new entry with the source IP address and a list containing the source MAC address; otherwise, it checks if the source MAC address is not already associated with the IP address. If it is not, the MAC address is appended to the list;
- if multiple MAC addresses are associated with the same IP address, an alert is printed indicating the potential IP address collision.

```
def packet_callback(p):
    arp = p[ARP]
    if arp.op != 1:
        return
    psrc = IPAddress(arp.psrc)
    hwsrc = EUI(arp.hwsrc)
    lock.acquire()
    try:
        if psrc not in associations:
            associations[psrc] = [hwsrc]
        else:
            if len(associations[psrc]) > 0 and hwsrc not in associations[psrc]:
                associations[psrc].append(hwsrc)
                s = ", ".join(str(a) for a in associations[psrc]
                print("ALERT! " +
                      "found {0} different MAC Addresses ({1}) " +
                      "associated to the same IP Address ({2})").format(
                          len(associations[psrc]), s), psrc)
    finally:
        lock.release()
```

The script uses the `argparse` module to handle command-line arguments. It defines two arguments: `--interface` to specify the network interface and `--cleaning_time` to set the interval for dictionary clearing.

```
parser = argparse.ArgumentParser(description='Simple IP Collision Detector')
parser.add_argument("--interface", "-i", nargs="?", default="eth0", type=str,
                    help="network interface name", metavar="interface")
parser.add_argument("--cleaning_time", "-c", nargs="?", default="5", type=float,
                    help="seconds elapsed between a dictionary clear and another one",
                    metavar="cleaning_time")

args = parser.parse_args()
iface=args.interface
cleaning_time=args.cleaning_time

cleaner_thread = CleanerThread()
cleaner_thread.daemon = True
cleaner_thread.start()

print("Starting Simple IP Collision Detector...")
sniff(iface=iface, prn=packet_callback, filter="arp", store=0, count=0)
```

Rogue DHCP detector

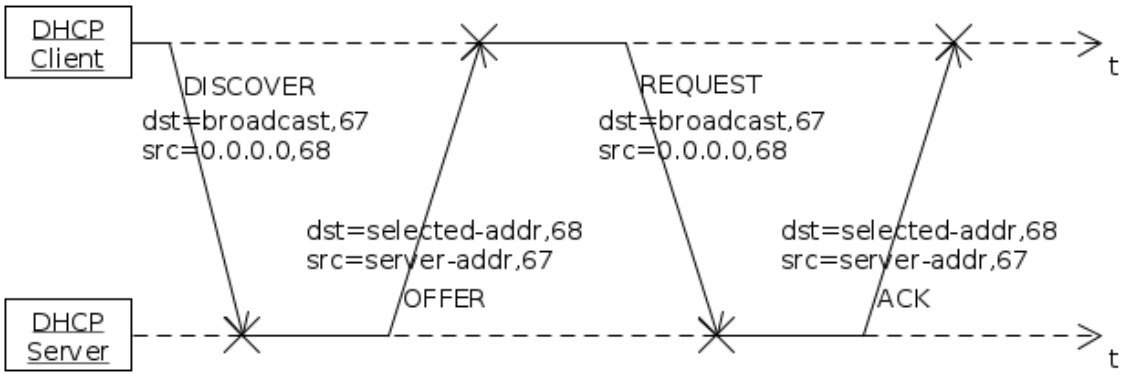
Before diving into the algorithm behind this tool, let's briefly discuss some aspects of the BOOTP and DHCP protocols. [8], [9]

```
>>> ls(BOOTP)
op          : ByteEnumField                     = (1)
htype       : ByteField                         = (1)
hlen        : ByteField                         = (6)
hops        : ByteField                         = (0)
xid         : IntField                          = (0)
secs        : ShortField                        = (0)
flags       : FlagsField (16 bits)              = (0)
ciaddr      : IPField                           = ('0.0.0.0')
yiaddr      : IPField                           = ('0.0.0.0')
siaddr      : IPField                           = ('0.0.0.0')
giaddr      : IPField                           = ('0.0.0.0')
chaddr      : Field                             = ('')
sname       : Field                             = ('')
file        : Field                             = ('')
options     : StrField                          = ('')
>>> ls(DHCP)
options     : DHCPOptionsField                  = ('')
>>>
```

Here is a list of relevant fields:

- `op`: the operation field can be `1` (Boot Request) or `2` (Boot Reply);
- `htype`: the hardware type indicates the type of hardware being used, for example, for Ethernet cards, this field is set to `0x01`;
- `xid`: represents the transaction ID. It will be set using the `RandInt()` function;
- `chaddr`: represents the client's MAC Address (in hexadecimal, not as a string!);
- `options`: consists of a list of options to be requested to the DHCP server. [9] In this list, we highlight 4 specific options, marked by an 8-bit ID (from 0 to 255):
    - `3` (Router): this ID is accompanied by a list consisting of 1 or more elements, each of which is 4 bytes in size and semantically represents the IPv4 address of a gateway;
    - `53` (Message type): It can be accompanied by the DHCP message type. This option is 1 byte in size. Common message types can be `1` (`DISCOVER`), `2` (`OFFER`), `3` (`REQUEST`), ...
    - `55` (Parameter request list): This ID is accompanied by a list consisting of 1 or more elements, each of which is 1 byte in size and semantically represents another DHCP option ID. In the response (e.g., the response to a DISCOVER can be an OFFER), the requested options can be populated;
    - `255` (End): An option that delimits the end of the options list.

The message exchange that takes place between the *DHCP Client* and *DHCP Server* is shown below. It should be known that DHCP uses UDP as the transport protocol, with DHCP servers listening on port 67, while clients listen on port 68.



The tool's main objective is to initiate a DHCP DISCOVER message and await responses (in the form of DHCP OFFER messages) from DHCP servers present on the network. The tool must have prior knowledge of the DHCP servers[8] that are authorized to offer IPv4 addresses within the network. Moreover, since these DHCP servers can offer additional configuration options alongside IPv4 addresses, the tool should maintain a list of all valid options assigned by the servers. Specifically, as an example of a valid option, it is assumed that the tool is already aware of the "legitimate" default gateway. Consequently, the tool verifies whether the default gateway provided by the DHCP matches its preconfigured gateway. If a mismatch occurs, the tool promptly generates an alert to notify the network administrator. The presented algorithm can be further developed to verify every possible option that can be assigned by the DHCP server to ensure proper network connectivity and security.

Rogue DHCP detector: a possible implementation

This code is a Python script that detects rogue DHCP servers on a network. A rogue DHCP server is an unauthorized device that is offering IPv4 addresses on a network, potentially causing network connectivity issues and security risks. The script uses the scapy library to send DHCP DISCOVER messages and to capture and analyze DHCP OFFER messages.

The script takes command-line arguments for the network interface to use, the delay between sending discover messages, and the file containing a list of associations between DHCP server IPv4 addresses and "legitimate" default gateway IP addresses. It starts by creating a thread (`DiscoverThread`) that periodically sends DHCP DISCOVER messages at the specified interval.

The `packet_callback` function is the callback function that is called for each captured packet. It checks if the packet contains DHCP and BOOTP layers, verifies if the operation field indicates an OFFER, and compares the gateway IP address received from the DHCP server with the expected gateway IP address associated with that DHCP server. If any discrepancies are found, it prints an alert message indicating the presence of a rogue DHCP server.

[8] e.g. by loading them from a configuration file, etc.

The script also loads the associations from the specified file and prints the loaded associations. It then starts the packet capture using the sniff function from scapy, filtering only DHCP traffic, and calling the `packet_callback` function for each captured packet. Overall, this script provides a simple but effective way to detect rogue DHCP servers on a network by actively sending `DHCP DISCOVER` messages and analyzing the responses.

```
#!/usr/bin/env python
from scapy.all import *
import argparse
from netaddr import IPAddress
import os.path
import sys
import threading
import time

# Filter for DHCP traffic
dhcp_filter = "(udp src port 67 and udp dst port 68) " + "or (udp src port 68 and udp dst port 67)"
pairs = {}

def create_dhcp_discover(iface):
    myMac = get_if_hwaddr(iface)
    myMAC_hex = myMac.replace(':', '').decode('hex')
    eth = Ether(dst='ff:ff:ff:ff:ff:ff', type=0x0800)
    ip = IP(src='0.0.0.0', dst='255.255.255.255', flags='DF', proto='udp', id=RandShort())
    udp = UDP(dport=67, sport=68)
    bootp = BOOTP(chaddr=myMAC_hex, xid=RandInt(), htype=0x01, op=1)
    dhcp = DHCP(options=[('message-type', 'discover'), ('param_req_list', '\\x03'), 'end'])
    discover = eth/ip/udp/bootp/dhcp
    return discover

# Command-line arguments parser
parser = argparse.ArgumentParser(description='Rogue DHCP-Server Discover')
parser.add_argument("--interface", "-i", nargs="?", default="eth0", type=str, help="network interface
name", metavar="interface")
parser.add_argument("--discover_delay", "-d", nargs="?", default="30", type=int, help="seconds elapsed
between a discover message and another one", metavar="discover_delay")
parser.add_argument("servers", type=str, help="an ASCII file containing a list of associations DHCP
Server IP Addr <-> Default Gateway IP Addr", metavar="servers")
args = parser.parse_args()
iface = args.interface
fname = args.servers
discover_delay = args.discover_delay

# Check if the file exists
if not os.path.isfile(fname):
    print("The given file is not valid.")
    sys.exit(-1)

# Thread for sending DHCP DISCOVER messages
class DiscoverThread(threading.Thread):
    def __init__(self, group=None, target=None, name='Discover',
                 args=(), kwargs=None, verbose=None):
        threading.Thread.__init__(self, group=group, target=target,
                                  name=name, verbose=verbose)
        self.args = args
        self.kwargs = kwargs
        self.iface = kwargs['iface']

    def run(self):
        discover = create_dhcp_discover(self.iface)
        while True:
            discover[IP].id = RandShort()
            sendp(discover, verbose=0)
            time.sleep(discover_delay)
```

```
# Callback function for processing captured packets
def packet_callback(p):
    if BOOTP not in p or DHCP not in p:
        return

    bootp = p[BOOTP]
    dhcp = p[DHCP]
    server_ip = IPAddress(p[IP].src)

    if bootp.op != 2:
        return

    if server_ip not in pairs:
        print("ALERT! Rogue DHCP-Server Detected! (addr: {})".format(server_ip))
        return

    gateway = ''
    message_type = ''

    for option in dhcp.options:
        if type(option) is not tuple:
            continue

        if option[0] == 'message-type':
            if option[1] != 2:
                message_type = 'not_offer'
                break
            message_type = 'offer'
        elif option[0] == 'router':
            gateway = IPAddress(option[1])

    if message_type != 'offer':
        return

    if gateway == '':
        print("INFO! Legal DHCP-Server is working! (addr: {0}, no gateway options
provided)".format(server_ip))
        return

    if gateway == pairs[server_ip]:
        print("INFO! Legal DHCP-Server is working! (addr: {0}, gateway: {1})".format(server_ip, gateway))
        return

    print("ALERT! Rogue DHCP-Server Detected! [addr: {0}, gateway: {1} (should be {2})
]".format(server_ip, gateway, pairs[server_ip]))

with open(fname) as fd:
    for line in fd:
        pair = line.split()
        dhcp_server = IPAddress(pair[0])
        gateway = IPAddress(pair[1])
        pairs[dhcp_server] = gateway
        print("Loaded {0} <---> {1}".format(dhcp_server, gateway))

print("Loaded {} associations.".format(len(pairs)))

# Start the DiscoverThread to send DHCP DISCOVER messages
discover_thread = DiscoverThread(kwargs={'iface': iface})
discover_thread.daemon = True
discover_thread.start()

print("Starting Rogue DHCP-Server Discover...")
# Start sniffing DHCP traffic and process packets with the packet_callback function
sniff(iface=iface, prn=packet_callback, filter=dhcp_filter, store=0, count=0)
```

Conclusion

In conclusion, this document has provided systematic approaches for controlling and monitoring Ethernet-based network infrastructures. Practical script examples, such as ARP monitor, IP collision detector, and rogue DHCP detector, have been discussed and showcased, illustrating their practical applications. By understanding and implementing these approaches, network administrators can effectively identify and address potential security and management issues within their network infrastructures.

Furthermore, the primary objective of this document is to enhance the understanding of network control and maintenance by demonstrating the functionality and practicality of the presented approaches. These approaches can serve as the basis for the development of more robust and user-friendly (e.g. with web-based graphical user interface) network management systems.

Ultimately, by consistently leveraging these tools and techniques, network administrators can ensure the stability and security of their Ethernet-based network infrastructures.

References

[1] Ethernet networks - https://www.ibm.com/docs/en/i/7.1?topic=standards-ethernet-networks
[2] Scapy's documentation - https://scapy.readthedocs.io/en/latest/
[3] Packet analyzer - https://en.wikipedia.org/wiki/Packet_analyzer
[4] Berkeley Packet Filter - https://en.wikipedia.org/wiki/Berkeley_Packet_Filter
[5] RFC 826 - An Ethernet Address Resolution Protocol - https://www.rfc-editor.org/rfc/rfc826.html
[6] Internet Protocol version 4 - https://en.wikipedia.org/wiki/Internet_Protocol_version_4
[7] Man-in-the-middle attack - https://en.wikipedia.org/wiki/Man-in-the-middle_attack
[8] Bootstrap Protocol - https://en.wikipedia.org/wiki/Bootstrap_Protocol
[9] Dynamic Host Configuration Protocol - https://en.wikipedia.org/wiki/Dynamic_Host_Configuration_Protocol